Światły Programista

Grażyna Mirkowska
Andrzej Salwicki

Dombrova Research

Przedmowa

Felix qui potuit rerum cognoscere causas *Wirgiliusz*

Światły programista – co to ma znaczyć? Czy to żart? Absolutnie, nie. Zwrot "człowiek światły" ma, według słownika języka polskiego, 83 synonimy. Możesz sobie wybrać znaczenie, które Ci odpowiada. Każde z nich zastosowane do Ciebie powinno sprawić Ci przyjemność. Naszym celem jest pomóc Ci byś stał się programistą rozumnym, wykształconym, niegłupim...

Światły programista potrafi nie tylko napisać program. To w końcu nie jest takie trudne. Pomocą w napisaniu programu służy nam edytor lub zintegrowane środowisko programistyczne, takie jak np. Eclipse. No i kompilator – kompilator wyłapie wszystkie błędy składniowe i wiele innych błędów, które cudacznie nazywają się "błędami statycznej, analizy semantycznej". To też są błędy składniowe!

Natomiast, żaden kompilator $\mathcal{K}$, żaden program $\mathcal{W}$ wykrywający błędy, nie jest w stanie wykryć czy przedstawiony mu dowolny program P, zapętli się czy też zakończy obliczenia. Co więcej nigdy, nikt nie napisze takiego kompilatora. Nie załamuj się, to jest dobra wiadomość! To jest bowiem gwarancja dla Ciebie i dla zawodu programisty na wiele pokoleń. Żaden szef nie powie "*przykro mi, musimy Was zwolnić bo zakupiliśmy oprogramowanie, które wykona Waszą pracę.*". A jeśli się taki znajdzie to możesz go śmiało wyśmiać i odesłać na szkolną ławkę by się czegoś nauczył.
Jak trudne mogą być pytania o to czy pewien konkretny algorytm zakończy obliczenia? Udowodnienie własności stop nawet krótkiego programu może okazać się bardzo trudnym zadaniem. Przyjrzyjmy się dwu przykładom.

Ponad trzysta lat zajęło setkom matematyków udowodnienie, że następujący algorytm PF nie zakończy obliczeń por.[2].

PF:

```
x:=0 & y:=0 & z:=0 & n:=0 ;
dalej := true;
powtarzaj
    jeśli n > 2 ∧ x * y * z > 0
    to
        jeśli x^n + y^n = z^n to dalej := false fi;
    fi;
    weź następną czwórkę x, y, z, n;
dopóki dalej ;
```

Wszystko w tym programie jest proste i zrozumiałe. Polecenie "weź następną

czwórkę" też nie trudno napisać. Przypomnij sobie, że wszystkie pary liczb naturalnych można ustawić w ciąg bez powtórzeń. Można to samo zrobić z czwórkami liczb naturalnych.

Zadanie 0.1. *Napisz odpowiedni algorytm realizujący polecenie: weź następną czwórkę liczb naturalnych.*

I kolejny przykład. Od ponad 80 lat, tysiące matematyków i informatyków próbują udowodnić, że następujący, prosty algorytm *Col* zawsze (tzn. dla każdej liczby całkowitej, dodatniej n) zakończy obliczenia, por. [19]

Col:
powtarzaj
$\{$**jeśli** n jest parzyste **to** $n := n \div 2$ **inaczej** $n := 3 * n + 1$ $\}$
dopóki $n \neq 1$

Przy pomocy komputerów sprawdzono, że dla każdej liczby $n < 2^{60}$ program *Col* kończy obliczenie z $n = 1$. Sprawdzanie tej własności dla kolejnych n zabiera coraz więcej czasu i kosztuje wciąż więcej. Nie mamy pewności czy gdzieś wśród bardzo dużych liczb naturalnych nie kryje się kontrprzykład[1]. Nie umiemy jak dotąd udowodnić przypuszczeenia Lothara Collatza: *dla każdego n program Col zakończy obliczenie.* Łatwiej było wejść na Mt. Everest i polecieć na księżyc, niż znaleźć odpowiedź na pytanie: czy program *Col* zawsze zakończy obliczenie?

Własność stopu jest podstawową własnością semantyczną programu(-ów). A są jeszcze inne, równie ważne własności semantyczne. Będziemy o nich mówić w dalszych rozdziałach.

Ale z drugiej strony, mamy nadzieję, że w przyszłości powstaną narzędzia ułatwiające pracę ludzi analizujących programy. Już powstają biblioteki klas (dawniej biblioteki procedur). Oczekujemy że będą im towarzyszyć dowody poprawności względem odpowiednich specyfikacji. Specyfikacja to coś więcej niż interfejs (ang. interface), zawiera ona nie tylko wyliczenie funkcji (czyli metod), ale także zbiór podstawowych własności tj. aksjomatów (zobacz przykłady w drugiej części tej książki). Oczekujemy też, że oprócz specyfikacji S i samej klasy K pojawią się dowody poprawności klasy K względem specyfikacji S. Dowody takie będą gwarancją (wieczystą!) jakości oferowanego przez daną klasę oprogramowania. Gwarancja ta stanie się jednak zbędna, gdy porzucimy stosowanie klasy K. Jeśli zamiast klasy K zaczniemy stosować nową klasę K' to trzeba wyprodukować nowy dowód poprawności. Do tego tematu wrócimy w części drugiej *programuj z klasą*.

Światły programista rozumie jaki efekt przyniesie zastosowanie danej instrukcji w określonym miejscu programu. Potrafi sformułować zdanie lub więcej zdań, orzekających o własnościach semantycznych programu. Przykładem mogą tu służyć zdania typu:

1° *ten program P nie zapętli się*, lub

2° *jeśli dane dostarczone programowi Q spełnią warunek α, to program zakończy obliczenia i jego wyniki spełnią warunek β*,

[1] Znane są przykłady hipotez formułowanych przez uznanych naukowców, które okazały się błędne, gdy komputer zaczął sprawdzac dostatecznie duże dane.

i wiele innych podobnych zdań.
Ponadto, i to jest chyba najistotniejsze, potrafi on przekonać innych o słuszności swojej ekspertyzy.

Jeśli potrafisz przekonać innych ludzi do swojej opinii np. *ten program nie zawiera błędu i nie zapętli się*, to zapewne i komputer Cię posłucha. Brzmi to jak żart, ale wcale żartem nie jest. Jeśli dokonałaś ekspertyzy programu, przeanalizowałaś nie tylko jego strukturę składniową (syntaktyczną), ale także zbadałaś własności semantyczne programu i jego składników. Jeśli Twoje rozumowanie nie zawiera błędu to znaczy to tyle, że komputer właśnie tak się zachowa jak to przewidziałaś.
No dobrze, ale co ja z tego będę miał(a)? – zapytasz. Pieniądze. Uznanie. A to nie jest mało.
Jako ekspert pomożesz zaoszczędzić czas przygotowania programu lub większego systemu programistycznego. Firmie opłaci się zatrudnić Cię i dobrze zapłacić bo Twoja praca może zaoszczędzić tygodnie pracy zespołu ludzi.[2]

Napisałeś program i ...

Skąd się wziął Twój program? Niewiele jest takich programów, które wzięły się "*znikąd*". Na ogół program powstaje w odpowiedzi na czyjeś zamówienie lub na zapytanie czy da się obliczyć coś potrzebnego? Programy pisane są dla ludzi, a wykonywane przez komputer. Wiele firm i wielu ludzi myśli jednak inaczej. Przekonanie takie prowadzi wprost do myślenia magicznego, nieracjonalnego. Trąci bowiem magią wiara w to, że skoro kompilator nie wykrył błędu i skoro w kilku, kilkudziesięciu, a nawet kilku tysiącach prób program nie objawił zachowania błędnego to można go uznać za program poprawny, tj. bezpieczny w eksploatacji. A często słyszymy o błędzie oprogramowania i o kosztach ponoszonych przez stosowanie oprogramowania kryjącego w sobie błędy. Produkcja oprogramowania przynosi dziś ogromne zyski firmom, które je sprzedają (nie musimy tego dowodzić.) Z drugiej strony posługiwanie się oprogramowaniem w którym ukrywaja się błędy, może kosztować bardzo wiele. Jak często otrzymujemy program wraz z gwarancją jakości? Na czym taka gwarancja miałaby polegać? Może warto jednak zastanowić się nad kryteriami poprawności oprogramowania.

Program nie jest celem samym w sobie. Czy ma on do czegoś służyć? Tworzymy programy po coś. Czy potrafimy sformułowac po co napisano dany program? Czy użytkownika może spotkać niespodzianka? Dlaczego testujemy programy? Czy nie ma innej drogi?
Dowodzenie alternatywą dla testowania. Którą metodę weryfikacji wybrać?

[2] Nie chcemy bowiem byś przypominał nam pewnego znajomego, który (wiele lat temu) programował w taki sposób:
– *o! coś tu nie działa!*
– *wyrzucę tę linijkę stąd, a tu wstawię taką instrukcję, powinno działać*
– *znowu nie działa?* – cofa ten ruch i wstawia coś innego, gdzie indziej, a nuż pomoże...

Programowanie z klasą

Czy ten zwrot kryje jakąś zagadkę? Czytelnik może się domyślać, że naszym zamierzeniem jest nauka programowania obiektowego. Rzeczywiście, jest to prawda, ale nie cała prawda.
Mówimy programowanie z klasą mając na myśli programowanie profesjonalne, kompetentne. Chcemy pokazać, że możliwe jest nie tylko pisanie programów, ale i ich analizowanie. Programując z klasą potrafisz przewidzieć efekty działania Twojego programu i co więcej potrafisz przekonać innych, że masz rację.

Dlaczego warto przeczytać ten podręcznik?

Ten tekst, to pierwszy podręcznik, z którego możesz się nauczyć nie tylko pisania programów, książka ta pomoże Ci w rozumieniu co Twój program naprawdę robi, możesz też nauczyć się przekonywania innych, że Twoje rozwiązanie jest poprawne. Zachęcamy Cię byś porównał tę książkę z (nielicznymi jak dotąd) próbami osiągnięcia podobnego celu, (por.Alagić i Arbib[29], Apt, de Boer i Olderog[4], Hoare[15], Dijkstra[9]).

Pracując z tą książką możesz nauczyć się dowodzenia własności semantycznych programu. No tak, zapytasz: *co to jest semantyczna własność programu*? Cierpliwości, za chwilę się dowiesz. Na razie wystarczy przyjąć, że są to m. in. poprawność algorytmu względem warunków, początkowego i końcowego, kończenie obliczeń, zapętlanie się programu, zgłoszenie wyjatku, ...

Byliśmy zapytywani czy Loglan ma związek z logiką algorytmiczną i czy podamy aksjomatyczną definicję Loglanu. Zawsze zdawaliśmy sobie sprawę z tego, że stworzenie aksjomatycznej definicji całości Loglanu jest trudne ze względu na jego rozmiar – język zawiera niemal komplet znanych narzędzi programowania m.in. klasy i obiekty, współprogramy, procesy i obiekty aktywne procesów, moduły obsługi wyjątków, etc.

Ta książka nie przynosi aksjomatycznej definicji semantyki języka Loglan. Jest to podręcznik programowania w którym mówi się o składni, semantyce i o pragmatyce. Mówimy też o specyfikacjach i o dowodach. Czytelnik poznaje kolejne coraz bardziej skomplikowane narzędzia programowania. Proces ten wykorzystuje solidne fundamenty języka Loglan. Uważamy, że uczenie programowania nie może ograniczać się do przedstawienia składni i pewnego zbioru przykładów. A tak się dzieje do dzisiaj. Niektórzy adepci nie potrafią tego zaakceptować. Inni jakoś dają sobie radę i wyciągają prawidłowe wnioski.

Jak korzystać z tej książki?

To jest dość gruba książka i może być wykorzystana do różnych wykładów obecnie obowiązujacego curriculum. Wiele lat temu prowadziliśmy kurs całoroczny Li1 "Programmation objet, semantique et algorithmes" na pierwszym roku studiów licencjackich na Wydziale Informatyki Uniwersytetu w Pau. Kurs ok. 120 godzinny obejmował materiał zbliżony do przedstawionego w tej książce. Staraliśmy się wtedy przedstawić trzy przeplatające się wątki: programowanie obiektowe i rozproszone, analizę programów oraz wybrane algorytmy i struktury

danych. Uzupełnieniem tego kursu było 120 godzin zajęć laboratoryjnych i tablicowych.

W polskich warunkach niniejsza książka może być użyta jako podręcznik podczas następujących zajęć: wstęp do programowania, programowanie obiektowe, wstęp do informatyki I i II (dla matematyków), programowanię współbieżne i rozproszone, semantyka i weryfikacja oprogramowania, języki i narzędzia programowania, inżynieria oprogramowania, logika dla informatyków i in.

Do wykładowców

Książka może być pomocna w przeprowadzeniu całorocznych zajęć *Wstęp do informatyki* (120 godzin wykładu + 150 godzin pracy w laboratorium i przy tablicy). Ponadto, wybrane elementy tej książki mogą wnieść nowe treści do wykładów:

- *programowanie obiektowe* – reguła konkatenacji klas pozwala zrozumieć dziedziczenie klas, moduły współprogramu (ang. coroutine) i procesu (ang. process)
- *semantyka i weryfikacja oprogramowania* – ...
- *programowanie współbieżne i obiektowe* – moduły process i obiekty procesów, protokół obcego wołania metod obiektów procesów,
- *metody realizacji języków programowania* - bezpieczny i efektywny system zarządzania pamięcia obiektów (tzw. sterta), algorytm wyznaczania klasy dziedziczonej (rozszerzanej) w Javie i ...,
- *algorytmy i struktury danych* –
  - aksjomatyczne definicje struktur danych np. stosy zob. rozdział 15, kolejki FIFO, kontenery (dictionaries), kolejki priorytetowe, drzewa BST zob. rozdział 19, etc.,
  - dowody semantycznych własności algorytmów por.6.7.2,
- i in.

Do studentów i doktorantów

Studenci zechcą przekonać się, że oferowana przez nas książka może być przydatna w całym okresie studiów.
Niektóre z omówionych problemów i zadań mogą zainspirować Cię do samodzielnej pracy i zaowocować opublikowaniem wyników w czasopiśmie naukowym.

Do informatyków pracujących zawodowo

Podczas studiów nie zetknąłeś się z treściami przedstawionymi w tej książce. Nawet pobieżne przeczytanie tej książki może zachęcić Cię do zmiany Twego światopoglądu informatycznego. Dojrzały i światły informatyk powinien rozpoznawać pytania jakie się pojawiają w jego pracy zawodowej np. czy moja firma powinna stosować imperatywny czy raczej funkcyjny język programowania? jak określic fundamentalną różnicę pomiedzy jednym a drugim paradygmatem programowania? dlaczego rekomenduję pradygmat P?

Trochę poważniejsze problemy pojawiają sie gdy firma zamierza zakupić oprogramowanie reklamujące się, że przy pomocy metod sztucznej inteligencji oprogramowanie to będzie w stanie wykryć błędy zapętlania sie programu lub błędy wiszących referencji.
Nie zamierzamy udzielać odpowiedzi na te i podobne pytania. To do Ciebie należy ich rozpoznanie, sformułowanie i zajęcie stanowiska.
Mamy nadzieję, że książka ta nie czyniąc z Ciebie lepszego specjalisty pozwoli Ci jednak stać się specjalista bardziej świadomym.
Książka ta może stać na Twej półce byś mógł sięgnąć w razie potrzeby np. gdy trzeba udowodnić, że Twój program nie zawiera błędu, ...

inni potencjalni czytelnicy

Ćwiczenia

Możesz po prostu czytać lub kartkować, tę książkę i mamy nadzieję, że okaże się to pożyteczne. Jeśli jednak chcesz opanować umiejętność analizowania i dowodzenia to koniecznie rozwiązuj zadania jakie zamieściliśmy. ...

przenieś dalej

Uważamy, że warto zastanowić się i opracować program osobnych zajęć laboratoryjnych, integrujących materiał z wszystkich przedmiotów oferowanych na pierwszym (i odpowiednio na drugim) roku studiów informatycznych. Jedno z nas pamięta, że na pierwszym roku studiów matematycznych podczas ćwiczeń z analizy matematycznej należało obliczyć całkę oznaczoną stosując metodę prostokątów. Było to ponad 60 lat temu, przed epoką komputerów. W ten sposób studenci zdobywali pewne doświadczenie z obliczeniami. Oczywiście, żaden z nas nie wiedział o tym, jak doskonałym rachmistrzem był Carl Gauss. Jesteśmy przekonani, że napisanie procedury obliczającej całkę oznaczoną przynosi głębszą intuicję i wiedzę na temat całkowania niż tylko ćwiczenia tablicowe z rachunku całkowego. Adeptowi informatyki, napisanie odpowiedniej procedury może przynieść niejedno odkrycie – jeśli zechce on dobrze zrozumieć zadanie. Podobnie jest z algebrą i geometrią. Ile pięknych zadań można sformułować. Dziś Laboratorium informatyczne mogłoby być prowadzone przez odpowiednio liczny zespół asystentów pod kierunkiem profesora i obejmować zadania z analizy, algebry liniowej z geometrią, wstępu do matematyki, programowania ...

Podziękowania

Książka ta ma dwa źródła: logikę algorytmiczną czyli *rachunek programów* oraz Loglan - projekt badawczy, który zaowocował językiem programowania obiektowego i rozproszonego oraz jego kompilatorem. Oba te projekty badawcze nadal są otwarte: wciąż napotykamy nowe problemy i (od czasu do czasu) uzyskujemy nowe wyniki. Książka, którą Ci oddajemy, stanowi wkład w kolejny projekt badawczy *SpecVer*, zobacz repozytorium **??**.

Projekt Loglan

Celem projektu Loglan było znalezienie odpowiedzi na pytanie: czy można skonstruować język programowania obiektowego, który umożliwiałby także programowanie współbieżne i był wolny od rozmaitych problemów jakie kryły się

w języku Simula67. Udało się opisać taki język i w roku 1982 skonstruować kompilator na minikomputery Mera400, produkowane wtedy w Polsce. Loglan wyprzedził języki C++, Java, etc. i nadal je wyprzedza. Przekonasz się o tym czytając m.in. o bezpiecznej dealokacji obiektów klas, o współprogramach, o obiektach procesów i protokole *alien call* współpracy pomiędzy obiektami klas rozproszonymi w sieci komputerowej.

W pracach nad językiem Loglan brali udział: Antoni Kreczmar, Wiesława Bartol, Hanna Oktaba, Tomasz Müldner, Marek Lao, Andrzej Salwicki i in.
Kompilator Loglanu powstał dzięki usilnej pracy Antoniego Kreczmara, Danuty Szczepańskiej-Wasersztrum, Andrzeja Litwiniuka, Wojtka Nykowskiego, Marka Lao, Pawła Gburzyńskiego.
Dwukrotnie wymieniliśmy Antka Kreczmara (1945 – 1996). Bez niego nie osiągnęlibyśmy niczego. Był on postacią bardzo ważną dla polskiej informatyki. Wciąż nie możemy przeboleć jego przedwczesnej śmierci.

Loglan przyciągał zainteresowanie studentów. Wielu z nich wniosło wkład w jego dalszy rozwój. Na pierwszym miejscu chcemy wymienić Bolka Ciesielskiego (1988) – wymyślił od nowa koncepcję procesów i ich obiektów, a przede wszystkim wymyślił protokół współpracy pomiędzy obiektami procesów znany jako "obce wołanie" metod obiektu aktywnego (*ang.* alien call).
Studenci M. Benke i G. Grudziński przenieśli kompilator Loglanu na komputery PC. Paweł Susicki zainstalował kompilator Loglanu na maszynach Unixowych, Linuxowych.
Oskar Świda stworzył środowisko VLP - ...

Współpraca międzynarodowa

- Institut für Informatik, Universitaet zu Kiel,
  Bardzo wiele zawdzięczamy profesorowi Hansowi Langmaackowi: zauważył on i pomógł poprawić nieprawidłowość w semantyce wielkości nielokalnych, pomógł w instalacji Loglanu na m.c. Siemens (odpowiednik mainframe'a IBM),
- IASI CNR Roma, dr Gianna Cioni
- Universita "La Sapienza" Roma, prof. Giorgio Ausiello
- i in. (Tuebingen, Bordeaux, Caen, ...)

Doktoraty

1. Paweł Gburzyński
2. Hanna Oktaba
3. Wiesława M. Bartol
4. Andrzej Szałas
5. Marek Lao
6. Andrzej I. Litwiniuk

7. Danuta Szczepańska
8. Uwe Petermann
9. Oskar Świda

wkład doktorantów

- O. Świda (1996): VLP - wieloprocesorowy, rozproszony, wirtualny komputer loglanowski i środowisko,
- A. Szałas: system operacyjny dla fabryki nawozów w Puławach 1983,
- D. Szczepańska: system zgłaszania wyjątków i ich obsługi, 1982(wdrożenie), 1990 doktorat,
- P. Gburzyński: System automatycznego dowodzenia twierdzeń, realizacja koncepcji prof M. Bibela, 1982 (Bibel miał swoją realizacje 2 lata później)
- H. Oktaba: formalizacja systemu zarządzania pamięcią 1982,
- W.M. Bartol: opis systemu współprogramów, 1983
- U. Petermann: koncepcja komunikacji procesów poprzez przerwania, 1986 i inne ciekawe wyniki
- M. Warpechowski: algorytmy wyznaczania bezpośredniej superklasy w Javie, 2004-2009
- inne

Wkład studentów

- kompilator: parser – W. Nykowski 1981
- koncepcja i realizacja *obcego wołania* metod B. Ciesielski 1988
- debugger – T. Przytycka 1984
- przeniesienie kompilatora
  na IBM PC/AT - M. Benke i G. Grudziński 1985
  do systemu Unix – P. Susicki 1989
  na komputer Atari – Sebastien Bernard Universite de Pau 1992
- środowisko Lotek – grupa studentów UW 1983
- XIIUWGRaph klasy dla grafiki i obsługi myszki – studenci z Universite de Pau, J. Larrieu, P. Becourt, 1995
- Fredrick Pataud przeniesienie Loglanu do środowiska Windows95 na maszyny 32 bitowe, 1995
- wtyczka Loglanowska do Eclipse - A. Chwedoruk 2005
- ponowna kompilacja kompilatora Loglanu'82 - A. Adamski 2011

– na platformę Windows
– na platformę Linux

Ta lista nie jest pełna. Kamil Burzyński przeniósł środowisko VLP na platformę Windows.

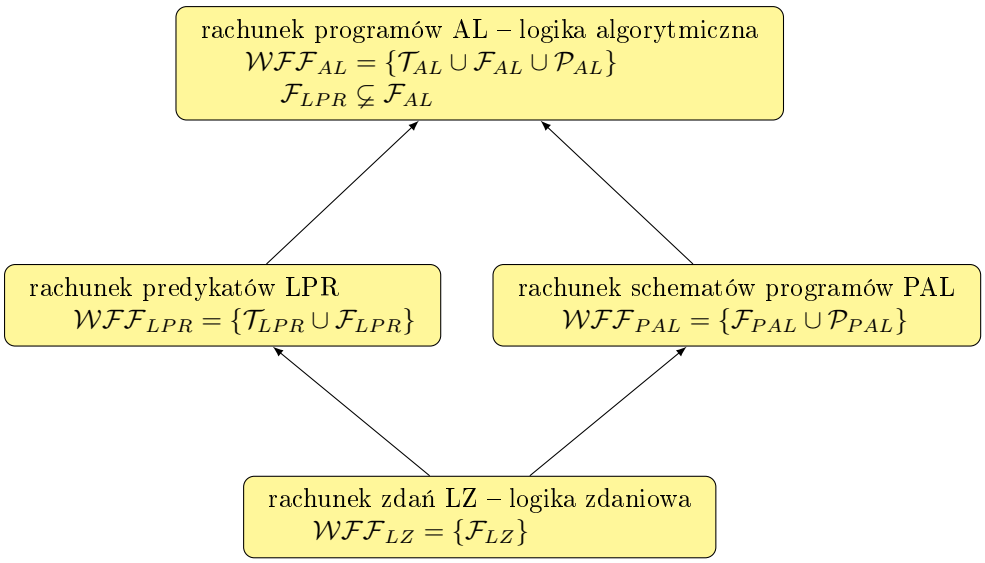
Rachunek programów

Celem projektu badawczego *rachunek programów* jest dostarczenie narzędzi przydatnych inżynierom oprogramowania w ich pracy nad projektowaniem oprogramowania (*specyfikacja* czyli sformułowanie wymagań) i później, podczas pracy polegającej na sprawdzeniu czy stworzone oprogramowanie jest zgodne z projektem (*weryfikacja*). To jest najkrótszy opis zadań rachunku programów, czyli logiki algorytmicznej. Przekonaliśmy się, że rachunek programów ma jeszcze inne zastosowania, również w matematyce. Rachunek programów służy nie tylko programistom, lecz także pracownikom dziś rzadkich zawodów: audytor oprogramowania, architekt – projektant oprogramowania. Pierwsze prace nt. logiki algorytmicznej zostały opublikowane w Warszawie na początku lat 70 ubiegłego wieku.

W kolejności chronologicznej należy wymienić następujących autorów: A. Salwicki (sformułowanie programu badawczego, kwantyfikatory iteracji), G. Mirkowska (twierdenie o pełności logiki algorytmicznej), A. Kreczmar (umieszczenie logiki algorytmicznej w hierarchii Kleene-Mostowskiego, badania programowalności w ciałach), L. Banachowski (zastosowanie logiki algorytmicznej do badania częściowej poprawności programów, algorytmiczna aksjomatyzacja drzew binarnych), H. Rasiowa (logika algorytmiczna wielowartościowa).
Projekt logika algorytmiczna czyli rachunek programów wiele zawdzięcza profesorom Helenie Rasiowej, Andrzejowi Grzegorczykowi i Andrzejowi Mostowskiemu. Obecni autorzy słuchali ich wykładów z logiki matematycznej i uczęszczali na ich seminaria.
Prekursorami rachunku programów byli między innymi:

- Jurij Janow (Kazań) – stworzył rachunek schematów programów, [33]
- Erwin Engeler(Zürich) – wskazał na fakt, że własność stopu programu jest wyrażalna w języku $\mathcal{L}_{\omega_1\omega}$ logiki z nieskończonymi alternatywami, [12]
- Helmut Thiele(Berlin) – twórca systemu łączącego rachunek λ z logiką pierwszego rzędu, [32]

Zwróć uwagę na Rysunek 1 pokazujący związki pomiędzy różnymi rachunkami logicznymi. Programista używa kilku konstrukcji programotwórczych złożenie (czyli średnik ;), if, while,... pozwalających tworzyć większe programy z mniejszych. Programami atomowymi są instrukcje przypisania i kilka innych wg uznania twórcy języka programowania. W odróżnieniu od formuł (wyrażeń logicznych), programy nie wyrażają prawdziwości lub nie. Natomiast możemy potraktować program jako modalność: *po zakończeniu obliczeń programu K zachodzi warunek ....* Wyrażenie postaci
<Program K> <formuła α>
jest formułą, która przyjmuje wartość prawda jesli: *po wykonaniu programu K, spełniona jest własność wyrażona przez formułę α*



Rysunek 1: Porównanie rachunków logicznych i ich języków tj. zbiorów występujących w nich wyrażeń poprawnie zbudowanych $\mathcal{WFF}$. Strzałki prowadzą od uboższego rachunku R do zawierającego go bogatszego i bardziej skomplikowanego rachunku R'.

Test

Czy jesteś ekspertem?

Przeczytaj poniższy program.
1° Czy program wydrukuje zachętę do dalszego czytania? Czy potrafisz opisać co jest wynikiem tego programu?
2° Czy potrafisz przeprowadzić dowód, że trafnie odgadłeś i zapisać ten dowód, tak byś mógł pokazać go znajomemu, nauczycielowi etc.
Masz na to trzy godziny. *Nie uruchamiaj programu!* Ale jeśli ma Ci to pomóc, to przepisz go w ulubionym języku programowania.
Życzymy powodzenia!

```
program PawelG:
    var A: arrayof integer;
    const l=1, u=8; (* te liczby mozesz zmienic *)
    var i, k: integer ;
    unit DrukujA: procedure;
        var j: integer
    begin
        for j:=l to u do write( A(j)) od;
        writeln
    end DrukujA;
    unit F: procedure;
        var i: integer;
    begin
        if k=u+1 then
            call DrukujA
        else
            for i:= l to u
            do
                if A(i)=0 then
                    A(i) := k; k := k+1;
                    call F;
                    k := k-1; A(i):=0
                fi;
            od;
        fi;
    end F;
begin
    array A dim(l:u);
    (* Tablica A ma u - l +1 elementów *)
    (* nastepna instrukcja jest zbedna, dodana dla nowicjuszy w Loglanie *)
    for i := l to u do A(i) := 0 od;
    k :=l;
    call F;
    writeln("Czytaj dalej")
end PawelG;
```

Wsparcie tej książki na stronie WWW

Na stronie

tu umieścić http://...

znajdziesz rozwiązania wybranych zadań, erratę dostrzeżonych błędów, materiały uzupełniające i pomocnicze etc.

Jak zainstalować środowisko do pracy z Loglanem?

Poradnik pozwalający Ci samodzielnie zainstalować kompilator i maszynę wirtualną Loglanu.

Linux

Wersja dla systemu operacyjnego Linux jest dostępna ze strony
https://sourceforge.net/projects/loglan82/
Wybierz Download i pingwina.

Windows

Nie istnieje zadowalająca nas wersja Loglanu dla systemu operacyjnego Windows. Masz dwie możliwości.
1) Przeczytaj instrukcję dla instalacjii na Windows.
http://lem12.uksw.edu.pl/images/9/9b/Instaluj_Loglan_na-_Windows.pdf
2) Spróbuj kompilatora i wykonawcy podobnego do instalacji Linuxowej.
Obie wersje są dostępne ze strony sourceforge.net.
https://sourceforge.net/projects/loglan82/
Wybierz Download i windows.

Źródła

Jeśli potrafisz, to możesz eksperymentować instalując Loglan w nowym środowisku: np. na Raspberry pi lub na maszyny Apple.
Źródła są dostępne na
https://sourceforge.net/projects/loglan82/
Wybierz Code.

Spis treści

Rozdział 1

Wprowadzenie

struktura tego rozdziału?

Przepisane ze wstępu do "logic for computer science"

1.1. Aims and Objectives Głównym celem jaki sobie stawiamy jest przekonanie Czytelnika do poglądu, że nie samym programem programista żyć powinien. Mówiąc nieco powazniej, napisane programu i wykonanie obliczeń przy jego pomocy to nie jest ani początek, ani koniec pracy.

Czym jest program? Zgodzimy się, że jest to opis pewnej funkcji przekształcającej dane w wyniki. Pewne cechy programu kojarzymy z matematyką, np. w programie występują wyrażenia arytmetyczne i boolowskie. Inne wymagania nakładane na program, to wykonywalność (inaczej efektywność) programu. tj. sens programu, jego znaczenie jest nam dane poprzez obliczenie. Obliczenie operuje na danych jakie i my i komputer może "wziąć do ręki". Oznacza to tyle, że nie możemy działać na nieskończonych ciągach rozwinięć liczb niewymiernych, nie możemy operowac na zbiorach nieskończonych. Co więcej, kolejne wymaganie powiada każda operacja musi być efektywna. Inaczej mówiąc nie możesz magicznie, wyciągnąć królika z kapelusza. Obliczenia mają mieć tę samą cechę co dowód matematyczny – mianowicie muszą być sprawdzalne, intersubiektywne jak powiadał prof. Grzegorczyk.

rozwiń, cyt. Kołmogorow

Ale po co napisano program? Lub inaczej mówiąc jakie własności ma mieć program?Czy wystarcza nam,że program jest napisem akceptowanym przez kompilator? Czy zadowalamy się tym, że program działa? A co to właściwie znaczy? Otóz program ma własności syntaktyczne i bardziej interesujące własności semantyczne.

Spójrzmy z innej strony. Programy są napisami, zbiór programów akceptowanych przez kompilator i wykonywanych przez komputer to język programowania. A własności semantyczne? Wymieńmy kilka podstawowych własności semantycznych: zatrzymywanie się programu, poprawność, równoważnośc programów, etc. Bardzo istotne dla historii informatyki było stwierdzenie, że potrzebujemy wyrażeń wyrażających własności semanyczne programów. Najwcześniej pojawiły się prace analizujące równoważnośc programów. Należy tu wymienić J. Yanova i Sh. Igarashi. Niestety niewiele udało się osiągnąć przy takim podejściu.

W 1967 pojawiła się praca E. Engelera, w której wykazał on, że własność stopu programu wyraża się formułą – nieskończoną alternatywą. Nieco inaczej podszedł do semantyki programów R. Floyd.

Wspomnij o Algolu 60 i pragnieniu opisania semantyki po opisie składni.

Zdania oznajmujące i rola Fregego oraz Boole'a.

Opowiedz o swoich odczuciach na początku pracy jako programista i równocześnie słuchacz wykładów A. Grzegorczyka o logice matematycznej.

Język termów (wyrażeń nazwowych) i formuł (wyrażeń zdaniowych) wymaga uzupełnienia o programy (algorytmy) .

Zdania o programach.

Rachunek programów czyli logika algorytmiczna

Działania nieskończone w rachunku programów – pętla while i kwantyfikatory iteracji.

Tarski – pojęcie spełniania.

Rózne definicje algorytmu, teza Curcha i brak formuł zdaniowych wyrażających własności algorytmów.

S. Kleene był blisko. Twierdzenie o postaci normalnej ... Związek operacji minimum z konstrukcją while jest oczywisty.

1.2. Background history

1.3. Background terminology

1.4. Propositions, Beliefs and Declarative Sentences

1.5. Contradictions

1.6. Formalization

Cele tej ksiązki:

- nauczyć tworzenia programów w Loglanie,
- nie możemy ograniczać się do nauki składni, konieczne jest rozumienie skutków umieszczenia tej a nie innej linijki tekstu w programie,
- a więc musimy opanować semantykę ,
- na tym nie kończy się nasze zadanie – program powstaje po coś, program ma rozwiązac jakieś zadanie,
- nauczmy się argumentowania, że nasz program *dobrze* rozwiązuje postawione zadanie, powinniśmy posiadać umiejetność oceny kosztów, a także umiejętnośći obniżania kosztów eksploatacji naszego oprogramowania, umiejętność optymalizacji oprogramowania.

O składni Język programowania jest jęykiem sformalizowanym, ponieważ tego wymaga kompilator. Formalizacja ta nie może przeszkadzac człowiekowi w rozumieniu i analizowaniu programu. Będziemy starali się trzymać powyższego – w celach dydaktycznych, ale nie tylko. Złozoność języka Loglan (pomimo zwięzłej składni) jest tak duża, że nie potrafimy dziś podać aksjomatycznej definicji semantyki Loglanu.

Wybieramy więc taką drogę: przedstawimy rosnący ciąg podjęzyków języka Loglan i dla każdego z nich postaramy się określić nie tylko skłdnię, lecz także semantykę opisaną aksjomatami.

Formalizm oparty będzie na logice algorytmicznej czyli rachunku programów.

Poznawać będziemy rosnący ciąg podjęzyków języka Loglan'82.

$$\mathcal{L}_0 \subset \mathcal{L}_1 \subset \mathcal{L}_2 \subset \mathcal{L}_3 \subset \ldots \mathcal{L}_n \subset \textsc{Loglan}$$

i ciąg coraz mocniejszych abstrakcyjnych komputerów

$$\mathcal{K}_0 \sqsubset \mathcal{K}_1 \sqsubset \mathcal{K}_2 \sqsubset \mathcal{K}_3 \sqsubset \ldots \mathcal{K}_n \sqsubset \text{VLP}$$

a także ciąg teorii algorytmicznych

$$\mathcal{T}_0 \subset \mathcal{T}_1 \subset \mathcal{T}_2 \subset \mathcal{T}_3 \subset \ldots \mathcal{T}_n.$$

Każdemu językowi $\mathcal{L}_i$ odpowiada abstrakcyjny komputer $\mathcal{K}_i$, który potrafi wykonywać programy zapisane w tym języku. Będziemy się starać, by równocześnie z definicją kolejnego podjęzyka $\mathcal{L}_i$ podać definicję algorytmicznej teorii $\mathcal{T}_i$. Teoria $\mathcal{T}_i$ ma dostarczać aksjomaty i reguły wnioskowania niezbędne do analizowania semantycznych własnośći programów z języka $\mathcal{L}_i$. Zamiar ten będziemy realizować dla języków opisanych w piewszej części naszej książki. Zapraszamy natomiast do współpracy nad aksjomatyzacją języków zawartych w częściach następnych.

Każdy program P wykorzystuje pewien zestaw narzędzi programowania i w ten sposób plasuje się w pewnym języku $\mathcal{L}_i$. Programy z języka $\mathcal{L}_i$ wykonuje komputer $\mathcal{K}_i$. Program może być wykonywany z różnymi danymi. Zbiór danych jest nieskończony. Czy można zawczasu (zanim oddamy program użytkownikowi) sprawdzić czy jest on poprawny? NIE! pisało o tym wielu autorów por. **??**.
Ale możemy podjąć się próby udowodnienia, że program ma pożądane właściwości. Właściwości semantyczne - to nas interesuje!

Pełność

Można sobie zadawać pytania: czy własność semantyczna W prawdziwa w realizacji na komputerze $\mathcal{K}$ posiada dowód? Okazuje się, że badanie właściwości programu $P \in \mathcal{L}_i$ można przeprowadzać w teorii $\mathcal{T}_i$ lub w jej pewnym rozszerzeniu $\mathcal{T}_i^P$. Teoria $\mathcal{T}_i^P$ jest w porównaniu z teorią $\mathcal{T}_i$ bogatsza o nowe aksjomaty zdefiniowane przez deklaracje funkcji zawarte w programie P. W drugiej części zobaczymy, że wzbogacenie może tez polegać na dodaniu całych nowych teorii – dzieje się tak , gdy w programie pojawiają się deklaracje klas. Por. część drugą Programuj z klasą.

popraw!

Tym, którzy lubią wyzwania proponujemy by wzięli udział w programie badania jakie teorie opisują programowanie z współprogramami i procesami. Należy spodziewać się czegos nowego.

A oto kilka pierwszych podjęzyków języka Loglan.

- $\mathcal{L}_0$ – język instrukcji drukowania,
- $\mathcal{L}_1$ – język wyrażeń (arytmetycznych, znakowych, boolowskich,)
- $\mathcal{L}_2$ – język programów liniowych, (same instrukcje przypisania)
- $\mathcal{L}_3$ – język programów elementarnych,
- $\mathcal{L}_4$ – język programów z tablicami,
- $\mathcal{L}_5$ – język programów iteracyjnych,
- ...
- Loglan

W każdej warstwie obowiązuje ta sama definicja pojęcia programu. Program jest specyficznym blokiem. Na blok składają się zbiór (a dokładniej, ciąg) deklaracji $\mathbb{D}$ i ciąg instrukcji $\mathbb{I}$. Kolejne elementy tych ciągów są oddzielone od siebie znakiem średnika; Pojęcia te będziemy definiować na nowo w każdym kolejnym języku $\mathcal{L}_i$.

Definicja 1.1. *Niech $\mathbb{D}$ oznacza ciąg deklaracji, niech $\mathbb{I}$ oznacza ciąg instrukcji.* ***Programem*** *jest wyrażenie o następującej strukturze*

program *Nazwa_programu*;
  $\mathbb{D}$
begin
  $\mathbb{I}$
end

Słowa **program, begin, end** są słowami kluczowymi języka.
Nie używaj ich do oznaczania czegokolwiek.
Uwaga dotycząca sposobu pisania słów kluczowych.
Dawno temu wymyślono następującą zasadę: *słowa kluczowe w druku pojawiają się jako półgrube, na tablicy pisane są zaś jako podkreślone.*
koniec uwagi.
Słów kluczowych używamy do organizowania struktury programu. Tutaj są to nawias otwierający **program**, śródnawias **begin**, nawias zamykający **end**. Słów tych nie formatujemy w pliku źródłowym programu, ale narzędzia edytujące mogą je pokazywać w kolorze.

Zbiór wyrażeń poprawnie zbudowanych $\mathcal{WFF}$ każdego rozważanego podjęzyka $\mathcal{L}_i$ języka Loglan jest sumą kilku zbiorów: zbioru wyrażeń $\mathcal{W}$, zbioru deklaracji $\mathcal{D}$, zbioru instrukcji $\mathcal{I}$ i zbioru programów $\mathcal{WP}$.

$$\mathcal{WFF} = \mathcal{W} \cup \mathcal{D} \cup \mathcal{I} \cup \mathcal{WP}$$

Omawiany w tym rozdziale zbiór wyrażeń $\mathcal{W}$ składa się z kilku rozłącznych podzbiorów: zbioru wyrażeń arytmetycznych typu całkowitoliczbowego $\mathcal{WI}$, zbioru wyrażeń arytmetycznych typu rzeczywistego – real $\mathcal{WR}$ (inaczej, termów), zbioru formuł (inaczej zbioru wyrażeń boolowskich) $\mathcal{WB}$, zbioru wyrażeń znakowych $\mathcal{WC}$, zbioru wyrażeń typu string $\mathcal{WS}$.

$$\mathcal{W} = \mathcal{WI} \cup \mathcal{WR} \cup \mathcal{WB} \cup \mathcal{WC} \cup \mathcal{WS}$$

Z każdym językiem $\mathcal{L}_i$ związemy odpowiedni abstrakcyjny komputer $\mathcal{K}_i$. Jest to abstrakcyjna maszyna wirtualna. Nie zastanawiamy się nad jej realizacją w komputerze. Z drugiej strony jest oczywiste, że operacje jakie wykonuje ta maszyna są efektywne (tj. obliczalne). Będziemy też mówić *wirtualny procesor loglanowski – VLP*. Komputery $\mathcal{K}_i$ tworzą ciąg maszyn coraz mocniejszych i przybliżają loglanowski procesor virtualny VLP.

rozwiń

Program logiki algorytmicznej

Lub inaczej, kolejne pytania i zadania jakie prowadzą do skonstruowania i zastosowań rachunku programów. Rachunek programów to inna nazwa logiki algorytmicznej.

1. Czy (algorytm) program jest tekstem?
2. Czy obliczenie algorytmu jest wyznaczone przez jego tekst? NIE. Są ciekawe przykłady algorytmów o zupełnie różnych obliczeniach gdy zastosowano je w różnych strukturach danych
3. Odróżniamy własności syntaktyczne programów (algorytmów) od własności semantycznych. Te pierwsze są nieciekawe, nietrudne w analizie. Np. ... Własności semantyczne są niebanalne. TAK!. Ponieważ ...
4. Czy mozna dowodzić własności semantyczne programów. TAK!
5. W tym celu należy wyrazić własnośc semantyczną przez odpowiednią formułę.
6. Wykazanie, że formuła ta jest prawdziwa to to samo co wykazanie, że prawdą jest iz własność semantyczna programu zachodzi.
7. Wymaga to wyjścia poza język pierwszego rzędu. Wprowadzamy formuły algorytmiczne.
8. Język zawiera trzy rodzaje napisów: termy, formuły i programy
9. Należy skonstruowac system aksjomatów i reguł wnioskowania pozwalający przeprowadzać dowody formuł algorytmicznych, a więc semantycznych własności programów.
10. Zbadac taki system, czy nie zawiera sprzeczności i czy jest kompletny.
11. Gromadzić doświadczenia w stosowaniu narzędzia jakim jest rachunek programów.

Część I

Programowanie algorytmów lub programowanie strukturalne

Ta część dzieli się na dwie mniejsze części. W rozdziałach 2 - 7 poznajemy składnię i semantykę programowania w pierwotnych typach danych (tzn. w strukturach algebraicznych integer, real, ..). Po opanowaniu tego mateiału będziesz umiał analizowac programy ... Chcemy podkreslic, że przedstawione tu własności operacji programotwórczych są istotne na dalszych szczeblach naszego rozumienia narzędzi programowania.

napisz to lepiej

W drugiej części, w rozdziałach **??** - 10 poznajemy narzędzia wzbogacania zastanych typów (tj. struktur danych) o nowe instrukcje atomowe i o nowe operacje i predykaty. Do tego celu służa deklaracje bloków, procedur i funkcji oraz instrukcje procedury i nazewniki funkcyjne.

Rozdział 2

Drukowanie $\mathcal{L}_0$

W tym, niedużym, rozdziale nauczymy się drukować liczby i teksty. Jest to niezbędne dla opanowania umiejętności tworzenia i uruchamiania własnych programów.

2.1 Język $\mathcal{L}_0$

W programach pierwszego języka $\mathcal{L}_0$, ciąg deklaracji – $\mathbb{D}$, jest zawsze napisem pustym, natomiast $\mathbb{I}$ – ciąg instrukcji, jest dowolnym skończonym, ciągiem instrukcji atomowych postaci: writeln i write( ). Argumentem instrukcji write może być tekst lub znak lub wyrażenie arytmetyczne. Niech s oznacza text, c oznacza znak, τ oznacza wyrażenie arytmetyczne.
W języku $\mathcal{L}_0$ *wyrażeniem arytmetycznym* jest liczba, całkowita lub liczba z przecinkiem dziesiętnym. Są to wyrażenia arytmetyczne najprostszej postaci. W kolejnych językach zbiór wyrażeń arytmetycznych będzie zawierać coraz więcej napisów.

Instrukcja	**Efekt**
writeln	wypisuj od nowego wiersza
write("$text$")	wypisz $text$
write('$znak$')	wypisz $znak$
write(τ)	wydrukuj wartość wyrażenia τ
write(τ:k)	wydrukuj wartośc τ, na k pozycjach
write(τ:k:m)	j.w., m znaków po przecinku

Ciąg kolejnych instrukcji write możesz skrócić wg tego wzoru

write(A); write(B); write(C); write(D);. . . ; write(M)

zastąp przez

write(A,B,C,D,. . . ,M).

Parę instrukcji

write(args);writeln

możesz zastąpić przez

writeln(args).

Przykłady

```
program druk1;
begin
  writeln("Witaj!")
  writeln;
  writeln("01234567890123456789012345678901234567890");
  write(3.1415926:12:7);write(' ');write('a'); writeln(3.1415926:12:3);
  writeln("01234567890123456789012345678901234567890");
  writeln("quick brown fox jumps over the leazy dog")
end
```

2.2 Abstrakcyjny komputer $\mathcal{K}_0$

Ta maszyna wykonuje po kolei instrukcje programu i wypisuje na ekranie odpowiednie znaki dopóki ciąg instrukcji pozostających do wykonania nie jest pusty.
RYSUNEK komputera K0
Wykonanie instrukcji writeln spowoduje przejście do nowej linii. Wykonanie instrukcji write('c') polega na wypisaniu znaku 'c' na ekranie.

2.3 Zaawansowane drukowanie

Możesz znacznie wzbogacić repertuar poleceń drukowania stosując tzw. "*sekwencje ESC*". Zobacz program ansitest.log. Ćwiczenia na umiejętność planowania wydruku ...
Wydrukuj tabelkę
Wydrukuj program
Drukowanie do pliku ... ?

2.4 Teoria konkatenacji

Drukowanie wydaje się być czynnością oczywistą i zrozumiałą. Właściwą dla niego teorią jest teoria konkatenacji, sformułowana przez Alfreda Tarskiego w XXw. Teoria ta jest nierozstrzygalna. Dowód tego faktu podał w 2006 roku Andrzej Grzegorczyk.

Aksjomaty teorii konkatenacji

Sygnatura tej teorii działanie konkatenacji (oznaczane *), stałe (tj. atomy) i równość.

$$x * (y * z) = (x * y) * z \tag{A1}$$

$$x*y = z*u \Rightarrow ((x = z \wedge y = u) \vee (\exists w)((x*w = z \wedge w*u = y) \vee (z*w = x \wedge w*y = u))) \tag{A2}$$

oraz pewna liczba aksjomatów opisujących atomy tj. znaki alfabetu np.

$$\neg(\alpha = x * y) \tag{A4}$$

$$\neg(\beta = x * y) \tag{A5}$$

$$\neg(\alpha = \beta) \tag{A6}$$

W powyższych trzech formułach należy napis α zastąpić przez symbol z alphabetu np. 'A', a napis β zastąpić przez inny element alfabetu np. '9', i powtórzyć to wielokrotnie.
Pierwszy aksjomat nie budzi wątpliwości: drukowanie jest operacją łączną – wszystko jedno czy najpierw wydrukujemy słowo $x * y$ a po nim słowo z, czy też najpierw wydrukujemy słowo x a po nim po kolei słowa y i z.
Drugi aksjomat też jest oczywisty: jeśli wydrukowaliśmy na dwa różne spsoby ten sam napis. tzn. gdy słowo $x * y$ jest równe słowu $z * u$ to albo $x = z$ i $y = u$ albo istnieje takie słowo w, że $x * w = z$ i $w * u = y$ lub $z * w = x$ i $w*y = u$. Tarski objasniał ten aksjomat w taki sposób: *Przypuśćmy, że na półce ustawiono dwa razy ciąg książek. Najpierw pierwsza osoba ustawiła podciąg x, do tego dostawiła (konkatenacja) podciąg y. Nieco później inna osoba na półce ustawiła najpierw podciąg z, a później u. Jeżeli w obu wypadkach uzyskano to samo ustawienie książek $x * y = z * u$, to albo $x = z$ i $y = u$ albo istnieje taki podciąg w książek, że $x * w = z$ i $w * u = y$ albo $z * w = x$ i $w * y = u$.*

$$\overbrace{books\_on\_the\_shelf\_were}^{z}\overbrace{\_put\_twice\_by\_different\_persons}^{u}$$

$$\underbrace{books\_on\_the\_shelf\_were\_put\_twice}_{x}\underbrace{\_by\_different\_persons}_{y}$$

rysunek półka z książkami

W tym przypadku słowo w to $\_put\_twice$.

$$\underbrace{books\_on\_the\_shelf\_were}_{z}\underbrace{\_put\_twice}_{w}\underbrace{\_by\_different\_persons}_{y}$$

Pozostałe aksjomaty stwierdzają, że napisy α i β są niepodzielne i różne.
Prof. Andrzej Grzegorczyk [14] wykazał, że w takiej teorii (nawet z dwoma tylko atomami) można reprezentować każdą funkcję programowalną. Stąd stosując rozumowanie przekątniowe można wyprowadzić następujące twierdzenie: *teoria konkatenacji jest nierozstrzygalna.*
Inaczej mówiąc, posługując się tylko operacją drukowania i kwantyfikatorami możemy zbudować formuły na tyle skomplikowane, że nie istnieje jeden ogólny algorytm rozstrzygania ich prawdziwości.

Rozdział 3

Wyrażenia $\mathcal{L}_1$

Język $\mathcal{L}_1$ wprowadza wyrażenia. Programy w języku $\mathcal{L}_1$ dopuszczają deklaracje zmiennych i stałych pięciu typów pierwotnych. Instrukcjami są nadal tylko instrukcje drukowania. Zauważ, że w instrukcjach tych możemy stosować bogatszy repertuar wyrażeń.
Ten rozdział poświęcony jest wyrażeniom, ale ponieważ wśród wyrażeń znajdujemy wyrażenia boolowskie, to język omawiany w tym rozdziale to język formuł otwartych (tj. bezkwantyfikatorowych). Po kolei omówimy różne rodzaje wyrażeń: całkowito-liczbowe $\mathcal{WZ}$, wyrażenia z liczbami rzeczywistymi $\mathcal{WA}$, wyrażenia boolowskie $\mathcal{WB}$, wyrażenia znakowe $\mathcal{WC}$, wyrażenia tekstowe (ang. string expression) $\mathcal{WS}$, wyrażenia tablicowe $\mathcal{WT}$, wyrażenia obiektowe $\mathcal{WO}$.
Zbiór wyrażeń $\mathcal{W}$ jest unią zbiorów

$$\mathcal{W} = \{\mathcal{WZ} \cup \mathcal{WA} \cup \mathcal{WB} \cup \mathcal{WC} \cup \mathcal{WS} \cup \mathcal{WT} \cup \mathcal{WO}\}$$

Omówienie wyrażeń tablicowych i obiektów tablic odkładamy na później. Wyrażeniom obiektowym i obiektom poświęcimy wiele miejsca w dalszym ciągu tej książki. Szczególne wartości wyrażeń obiektowych to obiekty współprogramów oraz obiekty procesów.
Najprostszymi wyrażeniami są stałe (spotkaliśmy je wcześniej) i zmienne. Zmienna x zadeklarowana jako typu integer jest wyrażeniem typu integer. Oznacza to tyle, że wartością tej zmiennej jest liczba całkowita, a takze że wartoś tej zmiennej może być argumentem działania dopuszczalnego w strukturze algebraicznej integer czyli typie pierwotnym integer. podbnie jest z deklaracjami innych zmiennych, być może innego typu. Ka

3.1 Przykłady programów

Obejrzyj te dwa przykłady. Zastanów się co może zostać wydrukowane. Uruchom te programy i sprawdź czy trafnie odgadłeś.

Przykład 3.1.

```
program delta;
  const a=2, b=5, c=1
begin
  writeln("delta=",b*b-4*a*c)
end
```

I jeszcze jeden przykład. Co tu robią deklaracje **var** zmiennych x i y?

Przykład 3.2.

```
program znaki;
  var x: real, y: integer;
  const a='a', b=5, c="Witaj"
begin
  writeln(c,' ', a, b*b+x+y)
end
```

3.2 Typy pierwotne

Typy pierwotne: Boolean $\mathbb{B}_0$, integer $\mathbb{Z}$, real $\mathbb{R}$, char (znak) $\mathbb{C}$, string(napis) $\mathbb{S}$ są strukturami algebraicznymi. Zakłada się, że wirtualny komputer loglanowski potrafi realizować działania opisane w tych strukturach.

3.2.1 Typ integer

Czym jest typ integer? Jest to struktura algebraiczna

$$\mathbb{Z} = \langle Z, \oplus, \ominus, \circledast, \div, \mathbf{modulo}, =, < \rangle$$

elementami struktury $\mathbb{Z}$ są liczby całkowite. Działaniami tej struktury są dodawanie $\oplus$, odejmowanie $\ominus$, mnożenie $\circledast$, dzielenie całkowito-liczbowe $\div$ i reszta z dzielenia tj. operacja **modulo**. W strukturze mamy dwie relacje $=$ równości i $<$ mniejszości.
Każda z pięciu operacji jest dwuargumentowa

$\oplus$	$: Z \times Z \to Z$	dodawanie
$\ominus$	$: Z \times Z \to Z$	odejmowanie
$\circledast$	$: Z \times Z \to Z$	mnożenie
divide	$: Z \times \{Z \setminus \{0\}\} \to Z$	dzielenie całkowitoliczbowe
modulo	$: Z \times \{Z \setminus \{0\}\} \to Z$	reszta

i dwie relacje (lub jeśli wolisz dwie funkcje charakterystyczne relacji)

$=: Z \times Z \to \{\mathbf{0}, \mathbf{1}\}$	równość
$<: Z \times Z \to \{\mathbf{0}, \mathbf{1}\}$	mniejszość

3.2.2 Typ Boolean

Ten typ to znana dwu-elementowa algebra Boole'a. Uniwersum składa się z dwu elementów **0** i **1**. W algebrze tej rozwazamy działania dwuargumentowe: kresu górnego $\cup$ i kresu dolnego $\cap$ oraz jednoargumentowe działanie uzupełnienia $-$. Inne operacje mogą być zdefiniowane przy pomocy tych trzech.

$$\mathbb{B}_0 = \langle \{\mathbf{0}, \mathbf{1}\}, \cup, \cap, - \rangle$$

$\cup$	**0**	**1**
0	**0**	**1**
1	**1**	**1**

$\cap$	**0**	**1**
0	**0**	**0**
1	**0**	**1**

	0	**1**
$-$	**1**	**0**

Przy pomocy powyższych tablic możemy określić tabelkę działania relatywnego pseudouzupełnienia $\rightarrow$, które definiujemy w ten sposób

$$a \rightarrow b \stackrel{df}{=} -a \cup b.$$

-	$\cup$	**0**	**1**
0	**1**	**1**	**1**
1	**0**	**0**	**1**

czyli

$\rightarrow$	**0**	**1**
0	**1**	**1**
1	**0**	**1**

3.2.3 Typ real

Ta struktura danych to zbiór liczb rzeczywistych (w Twoim komputerze będzie to pewien podzbiór zbioru liczb wymiernych). Dla naszych rozważań możemy jednak przyjąć, że mamy do czynienia ze zbiorem liczb rzeczywistych wraz z odpowiednimi działaniami i relacjami.

$$\mathbb{R} = \langle R, \oplus, \ominus, \circledast, \oslash, =, < \rangle$$

Mamy tu cztery działania dwuargumentowe:

$\oplus$: $R \times R \rightarrow R$ dodawanie
$\ominus$: $R \times R \rightarrow R$ odejmowanie
$\circledast$: $R \times R \rightarrow R$ mnożenie
$\oslash$: $R \times \{R \setminus \{0\}\} \rightarrow R$ dzielenie

i dwie relacje (lub jeśli wolisz dwie funkcje charakterystyczne relacji)

$=$: $R \times R \rightarrow \{\mathbf{0}, \mathbf{1}\}$ równość
$<$: $R \times R \rightarrow \{\mathbf{0}, \mathbf{1}\}$ mniejszość

3.2.4 Typ znakowy - char

Ten typ to skończony zbiór znaków jakie mogą byc wprowadzane z klawiatury.

$$\mathbb{C} = \langle A, ...z, 0, 1..., 9, +, -,), (, *, \&, \%, \$, \#, !, \rangle$$

bez żadnych operacji na znakach.
Uwaga. W kolejnych rozdziałach wprowadzimy funkcje *ord* i *char*. I przy ich pomocy określimy relację mniejszości w zbiorze znaków. **Koniec uwagi**

3.2.5 Typ tekstowy - string

Elementami tego typu są skończone ciągi znaków. Jedyne działanie to konkatenacja tekstów – operacja ta jest realizowana przez instrukcje drukowania.

3.3 Składnia

W języku zachowana zostaje struktura programu. Zbiór deklaracji zawiera deklaracje zmiennych i stałych typów pierwotnych. Zbiór wyrażeń jest bogatszy. Zbiór instrukcji to nadal tylko instrukcje drukowania, ale występujące w nich wyrażenia są teraz bardziej złożone.

3.3.1 deklaracje zmiennych i stałych

Definicja 3.3. *Niech ν oznacza identyfikator, ω niech będzie nazwą pewnego typu pierwotnego, $\omega \in$ {Boolean, integer, real, char, string } Deklaracja zmiennej ma następująca postać*

$$\textbf{var} \quad \nu : \omega$$

Mówimy: *zmienna ν jest (zadeklarowana) typu ω.*
Przykłady i skróty

```
var x: integer;
var b12: Boolean;
var c2: char;
var y: real;
var n: string;
```

Deklaracje zmiennych powinny być oddzielane znakiem średnika.
Można łączyć deklaracje np.

```
var c2:char, y: real, n: string;
```

Nazwa typu pierwotnego Boolean może rozpoczynać sie od małej litery.
Obie deklaracje są poprawne

```
var b12: boolean; var b13: Boolean;
```

Możemy też napisać

```
var b12, b13: boolean;
```

Podobnie wygląda deklaracja stałej. Niech ν będzie identyfikatorem. Niech τ będzie wyrażeniem typu pierwotnego.

Definicja 3.4. *Napis postaci*

$$\textbf{const } \nu = \tau$$

jest deklaracją stałej ν. Typem stałej ν jest typ wyrażenia τ.

Przykłady

```
const c1=17;
const c2=17.01;
const c3='v';
const c44="Witaj swiecie";
const c4= (c1+c2)/2;
```

3.3.2 Wyrażenia typów pierwotnych

Wyrażenia całkowito-liczbowe $\mathcal{WZ}$

Znaczeniem wyrażenia całkowito-liczbowego τ jest funkcja $\tau_{\mathfrak{A}}$ ze zbioru wartościowań zmiennych w zbiór liczb całkowitych

$$\tau_{\mathfrak{A}} : V^Z \to Z.$$

Definicja 3.5. *Zbiór wyrażeń całkowito-liczbowych jest to najmniejszy zbiór napisów $\mathcal{WZ}$ taki, że każda zadeklarowana zmienna całkowito-liczbowa z należy do zbioru $\mathcal{WZ}$, $z \in \mathcal{WZ}$, każda liczba całkowita c należy do zbioru $\mathcal{WZ}$, $c \in \mathcal{WZ}$, kazda zadeklarowana stała całkowito-liczbowa nalezy do zbioru $\mathcal{WZ}$, a ponadto jeśli do zbioru $\mathcal{WZ}$ należą napisy τ_1 oraz τ_2, to do zbioru $\mathcal{WZ}$ należą też napisy*

$$(\tau_1 + \tau_2), (\tau_1 - \tau_2), (\tau_1 * \tau_2), (\tau_1 \ \textbf{\textit{div}} \ \tau_2), (\tau_1 \ \textbf{\textit{mod}} \ \tau_2).$$

Przykłady wyrażeń całkowito-liczbowych

Wyrażenia arytmetyczne typu real

Wyrażenia boolowskie

Wyrażenia boolowskie odgrywają bardzo ważną rolę w programowaniu – dwie ważne konstrukcje programotwórcze: instrukcja warunkowa i instrukcja iteracji, wymagają napisania wyrażenia boolowskiego. Ponadto, przy pomocy wyrażeń boolowskich definiuje się budowę komputera i sposób realizacji działań na liczbach całkowitych.

Definicja 3.6. *Zbiór wyrażeń boolowskich to najmniejszy zbiór napisów $\mathcal{WB}$ taki, że*

(i) Zmienna q zadeklarowana jako boolean, należy do zbioru $\mathcal{WB}$, stała c zadeklarowana jako boolean, należy do zbioru $\mathcal{WB}$.

(ii) jeśli napisy α i β należą do zbioru $\mathcal{WB}$, to do zbioru $\mathcal{WB}$ należą też napisy postaci

$$(\alpha \text{ or } \beta), \quad (\alpha \text{ and } \beta), \quad \text{not } \alpha, \quad (\alpha \text{ orif } \beta), \quad (\alpha \text{ andif } \beta)$$

(iii) jeśli napisy τ oraz ν są wyrażeniami arytmetycznymi, to do zbioru $\mathcal{WB}$ należą też napisy postaci

$$(\tau = \nu), \quad (\tau \neq \nu), \quad (\tau < \nu), \quad (\tau > \nu), \quad (\tau \leq \nu), \quad (\tau \geq \nu).$$

Wyrażenia znakowe

Wyrażenia tekstowe

Wyrażeniem tekstowym jest zmienna zadeklarowana jako string lub stała tekstowa.
Przykłady.
"Witaj smutku"
s Gdzie s jest zmienną zadeklarowaną jako string. Każde wyrażenie może być przedstawione w postaci drzewa. Przykłady.

drzewa wyrażeń

3.4 Semantyka

Wartość wyrażenia

3.4.1 Wyrażenia całkowito-liczbowe

Obliczanie wartości wyrażenia
Jeśli dane jest wyrażenie τ to wartość wyrażenia $\tau(v)$ dla ustalonego wartościowania v wyznacza się zgodnie z następującą procedurą:

Gdy wyrażenie τ jest postaci	**to jego wartość wynosi**
τ jest liczbą l	$val(\tau) \stackrel{df}{=} l$
τ jest stałą zadeklarowana jako l	$val(\tau) \stackrel{df}{=} l$
τ jest zmienną z	$val(\tau) \stackrel{df}{=} v(z)$
$(\tau_1 + \tau_2)$	$val(\tau_1 + \tau_2) \stackrel{df}{=} val(\tau_1) \oplus val(\tau_2)$
$(\tau_1 - \tau_2)$	$val(\tau_1 - \tau_2) \stackrel{df}{=} val(\tau_1) \ominus val(\tau_2)$
$(\tau_1 * \tau_2)$	$val(\tau_1 * \tau_2) \stackrel{df}{=} val(\tau_1) \otimes val(\tau_2)$
$(\tau_1$ **div** $\tau_2)$	$val(\tau_1$ **div** $\tau_2) \stackrel{df}{=} val(\tau_1)$ **divide** $val(\tau_2)$
$(\tau_1$ **mod** $\tau_2)$	$val(\tau_1$ **mod** $\tau_2) \stackrel{df}{=} val(\tau_1)$ **modulo** $val(\tau_2)$

Przykłady
Wyrażenia $\tau : ((4 * (x * x)) - (9x + 5))$ i $\eta : ((4 * x - 9) * x) + 5$ mają tę samą wartość, wyznaczają tę samą funkcję $\tau_{\mathcal{N}} : N \to N$
...
Zauważ, że całkiem różne wyrażenia mogą mieć równe wartości dla wszystkich wartościowań, np. $x + y$ i $y + x$. Te spostrzeżenia mają realną wartość. Pomoga nam w skracaniu tekstu programu i w przyśpieszaniu jego obliczeń.
Ogólne zadanie: dla dowolnego wyrażenia całkowito liczbowego ω znajdź najbardziej optymalne równoważne wyrażenie, może okazać się bardzo trudne.
ZADANIA

Zadanie 3.1. *Do obliczenia wartości wyrażenia $b * b + a * a + a * 2 * b$ trzeba wykonać 4 mnożenia i 2 dodawania. Czy da się wyznaczyć tę wartość taniej, tzn. w mniejszej liczbie kroków?*

Własności (aksjomaty) struktury integer, struktury real.

Zadanie 3.2. *Do obliczenia wartości wyrażeń $a * b - c * d$ i $a * c + b * d$ potrzeba 4 mnożeń i dwu dodawań. Przyjmijmy, że operacja dodawania jest znacznie tańsza od operacji mnożenia. Czy można obliczyć te dwie wartości przy pomocy trzech mnożeń?*

3.4.2 Wyrażenia arytmetyczne

Obliczanie wartości wyrażeń arytmetycznych definiujemy w sposób analogiczny
...

3.4.3 Wyrażenia Boolowskie

Znaczeniem wyrażenia boolowskiego α jest funkcja $\alpha_{\mathcal{B}_0}$ ze zbioru wartościowań zmiennych boolowskich w dwuelementowy zbiór B_0

$$\alpha_{\mathcal{B}_0} \colon B_0^{V_B} \longrightarrow B_0$$

Funkcja ta jest okeślona przez indukcję ze względu na długość wyrażenia w następujący sposób: Niech V_B oznacza zbiór zmiennych boolowskich. Wartościami zmiennych boolowskich sa albo **true** albo **false**. Czasami zmaiast true będziemy pisać **1**, a zmiast **false** napiszemy **0**.

Wartościowaniem zbioru zmiennych boolowskich nazywamy odwzorowanie $v \colon V \to B_0$ przypisujące każdej zmiennej boolowskiej wartośc boolowską ze zbioru $B_0 = \{true, false\}$.

Litera W oznaczać będzie zbiór wartościowań zmiennych boolowskich $W = {B_0}^V$.

Każde wyrażenie boolowskie α zbudowane według reguł (i) - (ii) wyznacza funkcję α_{B_0} ze zbioru W w zbiór B_0. Niech v oznacza wartościowanie zmiennych boolowskich.

Wyrażenie	**Wartość**
true	**1**
false	**0**
$(\tau \leq \nu)$	$val((\tau \leq \nu), v) = val(\tau, v) \leq val(\nu, v)$
$q \in V_{Boolean}$	$val(q, v) = v(q)$
$\neg\alpha$	$val((\neg\alpha), v) = -val(\alpha, v)$
$(\alpha \vee \beta)$	$val((\alpha \vee \beta), v) = val(\alpha, v) \cup val(\beta, v)$
$(\alpha \wedge \beta)$	$val((\alpha \wedge \beta), v) = val(\alpha, v) \cap val(\beta, v)$

Zwróć uwagę na różnicę pomiędzy znakami $\vee$ i $\cup$. Pierwszy jest elementem alfabetu rozpatrywanego języka, drugi oznacza operację alternatywy w dwuelementowej algebrze Boole'a B_0. Podobnie ...
Funkcję *val* rozszerzymy na pozostałe wyrażenia boolowskie wykorzystując wcześniej określone znaczenie wyrażeń całkowito-liczbowych.

Zadanie 3.3. *Co można powiedzieć o wartościach wyrażeń?*

$$(\alpha \vee \beta) \ i \ (\beta \vee \alpha)$$

Napisać przykład długiego wyrażenia boolowskiego i jeszcze jednego wyrażenia i zapytać o ich równoważność.

Optymalizacja?

3.4.4 Wyrażenia znakowe

3.4.5 Wyrażenia tekstowe

Stałe tekstowe
Zmienne tekstowe
W Loglanie wyrażenia tekstowe są bardzo proste, nie ma operacji na tekstach.

3.5 Aksjomaty

Testowanie czy dowodzenie?
Analiza programu dokonywana jest w odpowiedniej teorii. W klasie języków omawianych w tym rozdziale, teoria jest wyznaczona przez zestaw deklaracji programu i zestaw aksjomatów. W zbiorze $\mathcal{L}_1$ programów, każdy program ustala zbiór zmiennych. Wyrażenia nie mogą zawierać zmiennych (lub odpowiednio stałych) niezadeklarowanych. Typy pierwotne użyte w tych deklaracjach decyduja o tym jaki zestaw aksjomatów jest niezbędny w trakcie analizy programu. Poniżej podajemy pięć zestawów dla każdego typu pierwotnego. Poczynając od języka $\mathcal{L}_7$ deklaracje programu będą mogły zawierać definicje funkcji, procedur, klas. W takim przypadku zestaw niezbędnych aksjomatów będzie musiał byc odpowiednio bogatszy.

3.5.1 aksjomaty rachunku zdań

Wyrażenia Boolowskie tj. formuły można analizować posługując się semantyka opisaną powyżej. Jest jeszcze inna droga, a mianowicie dowodzenie prwdziwości formuł wyprowadzane z aksjomatów. Poniżej podajemy zestaw aksjomatów i regułę wnioskowania.

Ax_1 $((\alpha \Rightarrow \beta) \Rightarrow ((\beta \Rightarrow \delta) \Rightarrow (\alpha \Rightarrow \delta)))$

Ax_2 $(\alpha \Rightarrow (\alpha \vee \beta))$

Ax_3 $(\beta \Rightarrow (\alpha \vee \beta))$

Ax_4 $((\alpha \Rightarrow \delta) \Rightarrow ((\beta \Rightarrow \delta) \Rightarrow ((\alpha \vee \beta) \Rightarrow \delta)))$

Ax_5 $((\alpha \wedge \beta) \Rightarrow \alpha)$

Ax_6 $((\alpha \wedge \beta) \Rightarrow \beta)$

Ax_7 $((\delta \Rightarrow \alpha) \Rightarrow ((\delta \Rightarrow \beta) \Rightarrow (\delta \Rightarrow (\alpha \wedge \beta))))$

Ax_8 $((\alpha \Rightarrow (\beta \Rightarrow \delta)) \Leftrightarrow ((\alpha \wedge \beta) \Rightarrow \delta))$

Ax_9 $((\alpha \wedge \neg\alpha) \Rightarrow \beta)$

Ax_{10} $((\alpha \Rightarrow (\alpha \wedge \neg\alpha)) \Rightarrow \neg\alpha)$

Ax_{11} $(\alpha \vee \neg\alpha)$

reguła odrywania

R_1 $\dfrac{\alpha, (\alpha \Rightarrow \beta)}{\beta}$

3.6 Komputer $\mathcal{K}_1$

Komputer Tarskiego Ten komputer oblicza wartości wyrażeń posługując się stosem.
Jak to działa?
Stos wartości
Elementami komputera są: pamięć czyli wartościowanie zmiennych (i stałych) oraz kalkulator zajmujący się obliczaniem wartości wyrażeń. Kalkulator z kolei ma stos wartości, aktualny symbol wejściowy , ... i tabelę akcji

wierzchołek stosu	**wejście**					
	stała	zmienna	operator	(	)	EoE
wartość	A	B	C	D	E	F
operator	A	B	C	D	BŁĄD	Błąd
(	A	B	C	D	BŁĄD	Błąd

Gdzie litery A, B, C, D, E, F oznaczają odpowiednio, następujące akcje:

A: wpisz wartość stałej na (wierzcholek) stos,

B: odnajdź wartość zmiennej w pamięci v i wpisz na wierzchołek stosu,

C: przepisz operator na wierzchołek stosu,

D: przepisz nawias otwierający na wierzchołek stosu,

E: pobierz ze stosu (pop) wartość w_2, operator $\circledcirc$ i wartość w_1, oblicz wartość $w = w_1 \circledcirc w_2$ i wpisz w na stos ,

popraw E

F pobierz wartość w z wierzchołka stosu; jeśli stos jest niepusty to BŁĄD w przeciwnym przypadku zwróć wartość w i zakończ pracę.

Twierdzenie 3.1. *Niech v będzie wartościowaniem zmiennych zadeklarowanych w programie. Niech τ będzie wyrażeniem całkowito-liczbowym.*
A) Dla każdego poprawnie zbudownego wyrażenia całkowito-liczbowego τ komputer $\mathcal{K}_1$ poprawnie obliczy wartość $\tau(v)$ tego wyrażenia dla danego wartościowania v.
B) Jeśli wyrażenie τ zawiera błąd składniowy, to zostanie on zasygnalizowany.

Dowód. Dowód przebiega przez indukcję ze względu na długość wyażenia τ. Jeśli wyrażenie τ jest zmienną x, to najpierw na stos zostanie wprowadzona wartość $v(x)$, a w następnym kroku po obejrzeniu symbolu EoE - koniec wyrażenia komputer zwróci tę wartość.

teza indukcyjna?

Podobnie będzie gdy wyrażenie τ jest stałą.
Załóżmy, że dla wyrażeń krótszych niż k, liczba naturalna teza twierdzenia jest prawdziwa. Rozpatrzmy wyrażenie τ postaci $(\tau_1 \circledcirc \tau_2)$. Z założenia indukcyjnego komputer poprawnie obliczy wartość $\tau_1(v)$, a potem wartość $\tau_2(v)$ i na wierzhołku stosu będą: nawias otwierający (, wartość $\tau_1(v)$, operator $\circledcirc$ i wartość $\tau_2(v)$. Z kolei na wejściu pojawi się nawias zamykający).
Dowód punktu B. Co to znaczy napis τ jest błędnym wyrażeniem całkowito-liczbowym? Rozpatrzmy po kolei przypadki:

- brakuje nawiasu otwierającego $\tau_1 \circledcirc \tau_2)$, lub
- brakuje pierwszego argumentu $(\circledcirc\tau_2)$, lub
- brakuje operatora $(\tau_1\tau_2)$, lub
- brakuje drugiego argumentu $(\tau_1\circledcirc)$, lub
- brakuje nawiasu zamykającego $(\tau_1 \circledcirc \tau_2$

Atomowe wyrażenie ? postać? W każdym z tych przypadków komputer zasygnalizuje błąd. □

Omów znaczenie punktu B. Jest to gwarancja odporności na ataki. Komputer jest *robust*.

Rysunek

Tabela : Wejście x Wierzchołek stosu –> Akcja

3.7 Analiza

Komputer $\mathcal{K}_1$ oblicza wartości wyrażeń. Wyrażenia opisują odwzorowania ze zbioru wartościowań w zbiór wartości odpowiedniego typu. Nietrudno zauważyć, że wiele wyrażeń opisuje tę samą funkcję. Możesz to potwierdzać eksperymentując i drukując wyniki. Jak rozpoznać równość wyrażeń? Jak dowodzić prawdziwości wyrażeń boolowskich postaci $\tau_1 = \tau_2$? Umiejętność rozpoznawania wyrażeń opisujących te same odwzoworowania ma znaczenie praktyczne. Porównaj wyrażenia $(a+b)^2$ i $b*b+ba+a^2+ab$. Pierwsze wyrażenie wymaga dwu działań, a drugie wymaga czterech mnożeń i trzech dodawań. Jeśli nasz program będzie musiał powtórzyć obliczenie milion razy to można zaoszczędzić bardzo wiele.

3.7.1 Aksjomaty liczb całkowitych

Aksjomaty pierścienia uporządkowanego oraz zdanie stwierdzające, że zbiór liczb całkowitych dodatnich jest dobrze uporządkowany: *każdy niepusty podzbiór zbioru N liczb całkowitych nieujemnych zawiera element najmniejszy.*

Liczby całkowite tworzą zbiór oznaczany **integer** (lub Z), razem z niepustym podzbiorem N (liczb całkowitych nieujemnych) i z dwoma operacjami dwuargumentowymi dodawania i mnożenia, oznaczanymi przez $+$ i $\cdot$, które spełniają następujące aksjomaty:

- (Przemienność) Dla każdej pary liczb całkowitych a, b zachodzą równości

$$a+b = b+a \quad \text{oraz} \quad a \cdot b = b \cdot a,$$

- (Łączność) Dla każdej trójki liczb całkowitych a, b, c zachodzą

$$(a+(b+c)) = ((a+b)+c) \quad \text{oraz} \quad (a \cdot (b \cdot c)) = ((a \cdot b) \cdot c),$$

- (Rozdzielność) Dla każdej trójki liczb całkowitych zachodzi

$$(a+b) \cdot c = a \cdot c + b \cdot c$$

- (Jedności) Istnieją liczby całkowite 0 i 1 takie, że dla każdego a zachodzi

$$a+0 = a \quad \text{oraz} \quad a \cdot 1 = a,$$

- (Domknięcie w N) Jeśli a i b są liczbami całkowitymi nieujemnymi to liczby $a+b$ oraz $a \cdot b$ też są liczbami całkowitymi, nieujemnymi,

- (Addytywna odwrotność) Dla każdej liczby całkowitej a, istnieje taka liczba całkowita $-a$, że zachodzi

a+-a =0

- (Trichotomia) Dla każdej liczby całkowitej a zachodzi dokładnie jedna z trzech relacji: albo a) liczba a jest nieujemna, albo b) liczba a jest zerem $a = 0$ albo c) liczba $-a$ jest nieujemna,
- (Dobre uporządkowanie zbioru N) Każdy niepusty podzbiór zbioru liczb całkowitych, dodatnich ma element najmniejszy.

Ostatni aksjomat ma inny charakter niż pozostałe. Użyto w nim kwantyfikatora wiążącego zbiory (a nie elementy). Nieco dalej, w rozdziale 7 podamy algorytmiczny aksjomat liczb całkowitych, bardziej przydatny w analizowaniu algorytmów.

3.7.2 Aksjomaty liczb rzeczywistych

Do wnioskowania o własnościach programów wykonywanych w strukturze liczb rzeczywistych będziemy wykorzystywać aksjomaty ciała uporządkowanego Archimedesowskiego. tzn. przyjmujemy, że działanie dodawania jest łączne, przemienne i rozdzielne z działaniem mnożenia. Dla każdej liczby l istnieje liczba o przeciwnym znaku $-l$, taka, że $l + -l = 0$. Działanie mnożenia jest łączne i przemienne. Dla każdej liczby $l, l \neq 0$ istnieje liczba l' taka, że $l * l' = 1$. Relacja $<$ jest spójna, przechodnia, asymetryczna, przeciwzwrotna. Zachodzi prawo $\forall x, y(x < y \vee x = y \vee y < y)$. Ponadto zachodzi prawo Archimedesa
dla każdej pary liczb dodatnich $x > 0, y > 0, y < x$ istnieje liczba całkowita, dodatnia k taka, że $x < k * y$.

wypisz aksjomaty

3.7.3 aksjomaty typu znakowego - char

O typie znakowym (char) wiadomo, że jest to skończony zbiór. Wystarczą więc dwa aksjomaty. Pierwszy to (długa) alternatywa mówiąca, że kazdy znak to $x =' A' \vee x =' B' \vee \cdots \vee x =' Z' \vee x =' 0' \cdots \vee x =' 9' \vee x =' +' \vee x =')' \ldots$
Drugi to jeszcze dłuższa formuła stwierdzająca, że kazdy element zbioru C jest różny od pozostałych.

3.7.4 aksjomaty typu string

Aksjomaty teorii konkatenacji przytoczyliśmy w rozdziale 2 Drukowanie. W języku nie występuje operacja konkatenacji na tekstach. Ale instrukcje drukowania właśnie tę operację realizują, dopisując kolejne znaki i teksty do poprzednio wydrukowanych.

3.8 Przykłady

Deklaracja zmiennej całkowito liczbowej z wygląda tak:

var z: integer;

Można też deklarować stałe całkowito-liczbowe, np.

const s7 = 7;

W instrukcjach write dopuszczamy teraz jako argumenty napisy będące wyrażeniami całkowito-liczbowymi.

Podobnie jak wcześniej program jest napisem o następującej budowie
program <nazwa>;
<deklaracje zmiennych i stałych>
begin
<instrukcje drukowania>
end

Wartość wyrażenia

Rozdział 4

Programy liniowe $\mathcal{L}_2$

W tym rozdziale zaczynamy programować tzn. tworzyć algorytmy. Programy liniowe, jakie będziemy rozpatrywać w tym rozdziale są bardzo ważną częścią każdego algorytmu. Są to ciągi instrukcji przypisania.

TODO

- SWAP(x,y) zamiana wartości,
- instrukcja x $\leftarrow$ *tau* moze być rozłożona na instrukcje prostsze,
- wykorzystanie własności struktur liczbowych
- wykorzystanie algebry Boole'a gdy instrukcja jest $q \leftarrow \gamma$
- mnożenie liczb zespolonych
- mnożenie macierzy 2 x 2 (przygotowanie do Strassena)
- najsłabszy warunek wstępny – programu liniowego
- najmocniejszy warunek końcowy – programu liniowego
-

4.1 Przykład programu i jego obliczenia

Przykładowy program

Niech będzie dany poniższy program $L1$

```
program L1;
  const a= 1,b= -3,c= 5,d= 11 ;
  var Aa,Bb: integer;
  var t1,t2, t3: integer
begin
  t1 ← a *c;
  t2 ← b*d;
  t3← (a+b)*(c+d);
  Aa← t1 -t2;
  Bb← t3 -t1-t2
end
```

Przykład obliczenia

Stan początkowy rekordu aktywacji

Pamięć

a	b	c	d	Aa	Bb	t1	t2	t3
1	-3	5	11	0	0	0	0	0

Ciąg instrukcji

```
t1:= a *c;
t2:= b*d;
t3:= (a+b)*(c+d);
Aa:= t1 -t2;
Bb:= t3 -t1-t2
```

Po pierwszej instrukcji

Pamięć

a	b	c	d	Aa	Bb	t1	t2	t3
1	-3	5	11	0	0	5	0	0

Ciąg instrukcji

```
t1:= a *c;
t2:= b*d;
t3:= (a+b)*(c+d);
Aa:= t1 -t2;
Bb:= t3 -t1-t2
```

Po wykonaniu dwóch instrukcji

Pamięć

a	b	c	d	Aa	Bb	t1	t2	t3
1	-3	5	11	0	0	5	-33	0

Ciąg instrukcji

```
t1:= a *c;
t2:= b*d;
t3:= (a+b)*(c+d);
Aa:= t1 -t2;
Bb:= t3 -t1-t2
```

Po wykonaniu trzech instrukcji

Pamięć

a	b	c	d	Aa	Bb	t1	t2	t3
1	-3	5	11	0	0	5	-33	-32

Ciąg instrukcji

```
t1:= a *c;
t2:= b*d;
t3:= (a+b)*(c+d);
Aa:= t1 -t2;
Bb:= t3 -t1-t2
```

Po czterech instrukcjach

Pamięć

a	b	c	d	Aa	Bb	t1	t2	t3
1	-3	11	5	38	0	5	-33	-32

Ciąg instrukcji

```
t1:= a *c;
t2:= b*d;
t3:= (a+b)*(c+d);
Aa:= t1 -t2;
Bb:= t3 -t1-t2
```

Po pięciu instrukcjach

Pamięć

a	b	c	d	Aa	Bb	t1	t2	t3
1	-3	11	5	38	-1	5	-36	-32

Ciąg instrukcji

```
t1:= a *c;
t2:= b*d;
t3:= (a+b)*(c+d);
Aa:= t1 -t2;
Bb:= t3 -t1-t2
```

I to jest koniec obliczenia tego programu. Na zielonym tle widać instrukcje jakie pozostaja do wykonania. Na tle blado-różowym są instrukcje już wykonane, można o nich zapomnieć. W tabeli Pamięci na czerwonym tle zaznaczono wynik ostatniej wykonanej instrukcji (przypisania). Po wykonaniu piątej instrukcji nie pozostała już żadna instrukcja do wykonania — obliczenie programu zostaje zakończone. Programiści mówia "*program zakończył działanie*".
Co się stanie jeśli zmienione zostaną stałe a,b,c,d? Np. w ten sposób a=11, b=5, c=12, d=-4.
Czy można podać jakąś ogólną prawidłowość?

4.2 Język

Rozpatrujemy język $\mathcal{L}_2$, który zawiera wszystkie wyrażenia z języka $\mathcal{L}_1$, a więc wszystkie poznane dotąd wyrażenia oraz instrukcje drukowania.

$$\mathcal{L}_1 \subsetneq \mathcal{L}_2$$

Składnia

Program w języku $\mathcal{L}_2$ ma budowę zgodną z definicją przyjętą wcześniej, zob. **??**. Instrukcją programu jest (oprócz instrukcji drukowania) instrukcja przypisania.

Definicja 4.1. *Niech z będzie zmienną i niech τ będzie wyrażeniem. Zakładamy, że typy zmiennej i wyrażenia sa równe.*
Instrukcją przypisania *jest napis postaci*

$$z \leftarrow \tau \ .$$

Symbol operacji przypisania $\leftarrow$ pojawił się tu nie przypadkiem. Po pierwsze, symbol ten pozwala odsunąć w czasie dyskusję, która ortografia jest lepsza ta z Algolu i Pascala tzn. $:=$, czy też dominująca dziś notacja w której równość $=$ jest wykorzystywana jako operator przypisania. Notacja z $=$, wywodzi się z lat 50tych XX wieku od języków Fortran i BCPL.
Znak strzałki w lewo miał występować w programach pisanych w Algolu60 w wersji publikacyjnej. A Ty przed oczami masz publikację. W przykładach może pojawićsię symbol $:=$, zamiennie z preferowanym przez nas, znakiem $\leftarrow$.
Przykład

Okaże się, że programy liniowe mogą nastręczyć sporo problemów.
Programy liniowe a dagi.
Por. książka Savage'a.

dagi programów liniowych

Problemy: optymalizacja,

Semantyka

W analizie programów liniowych pomocny będzie następujący schemat aksjomatu instrukcji przypisania

$$\{x \leftarrow \tau\}\alpha(x) \Leftrightarrow \alpha(x/\tau) \qquad \text{(ax18)}$$

Napis $\alpha(x/\tau)$ oznacza wyrażenie powstające z formuły α przez równoczesne zastąpienie wszystkich (wolnych) wystąpień zmiennej x w formule α. Nietrudno zauważyć, że napis $\alpha(x/\tau)$ jest formułą. Przykład.
$\{y := x + 7\}(x + y < 2) \Leftrightarrow (x + x + 7 < 2)$
Sens tego aksjomatu można odczytać z rysunku 4.1. Aksjomat stwierdza, że diagram na tym rysunku jest przemienny. A więc, wartość formuły algorytmicznej $\{x := \tau\}\alpha(x)$, którą wyznaczamy ... jest równa wartości formuły $\alpha(x\tau)$.

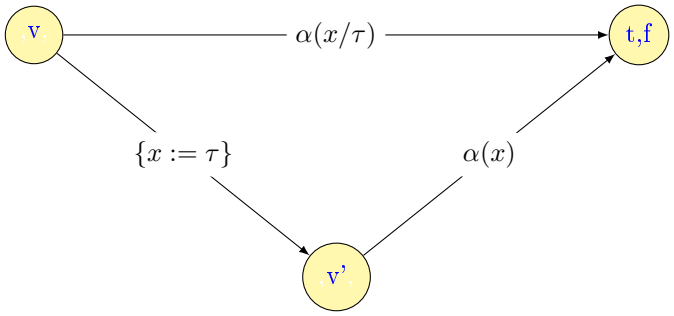
Co wynika z przemienności diagramu 4.1? Czy można ustalić jaka relacja zachodzi pomiędzy stanami pamięci v i v'?

Kolejnym aksjomatem rachunku programów jest

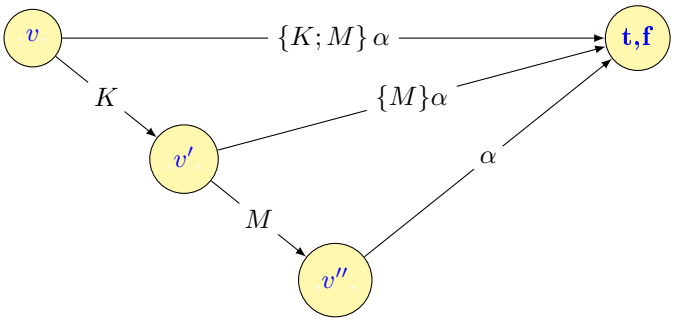
$$\{K; M\}\alpha \Leftrightarrow KM\alpha \ . \qquad \text{(ax19)}$$

Każda formuła zbudowana zgodnie z tym schematem jest tautologią.

Aksjomaty te zastosujemy w poniższym dowodzie.



Rysunek 4.1: Aksjomat instrukcji przypisania (schemat)



Rysunek 4.2: Aksjomat złożenia programów Ax19 (schemat)

Komputer $\mathcal{K}_2$

Komputer $\mathcal{K}_2$ umie nie tylko obliczać wartości wyrażeń, ale także potrafi obliczoną wartość prypisać do zmiennej jako jej nową wartość. Pamięć komputera $\mathcal{K}_2$ jest wyznaczona przez ciąg deklaracji programu. Instrukcje programu wykonywane są po kolei i zmieniają stan pamięci (tzn. wartościowanie zmiennych).

4.3 Kilka przykładów

SWAP – zamiana wartości

Czy jest prawdą, że po wykonaniu trzech instrukcji, zmienne x i y zamieniły się wartościami? Rozpatrzmy następujący program.

```
program SWAP;
  const k= , l= ;
  var x, y, t: integer;
begin
  t :=x;
  x:=y;
  y := t;
  writeln("x= ",x,"y= ",y)
end
```

Pytanie brzmi:
czy prawdą jest $(x = k \wedge y = l) \Rightarrow \{t := x; x := y; y := t\}(x = l \wedge y = k)$*?*
Zastosujemy aksjomat instrukcji przypisania ax18 i aksjomat instrukcji złożonej ax19.

Nr	Formuła	Podstawa
(1)↓	$(x = k \wedge y = l) \Rightarrow \{t := x; x := y; y := t\}(x = l \wedge y = k)$	
(2)↓	$(x = k \wedge y = l) \Rightarrow \{t := x; x := y\}\{y := t\}(x = l \wedge y = k)$	≡1, ax19 ↑
(3)↓	$(x = k \wedge y = l) \Rightarrow \{t := x; x := y\}(x = l \wedge t = k)$	≡2, ax18 ↑
(4)↓	$(x = k \wedge y = l) \Rightarrow \{t := x\}\{x := y\}(x = l \wedge t = k)$	≡3, ax19 ↑
(5)↓	$(x = k \wedge y = l) \Rightarrow \{t := x\}(y = l \wedge t = k)$	≡4, ax18 ↑
(6)	$(x = k \wedge y = l) \Rightarrow (y = l \wedge x = k)$	≡5, ax18 ↑

Nie ma wątpliwości, że formuła w wierszu 6 jest tautologią. Wobec tego, formuła w wierszu 5 też jest tautologią, ponieważ jest równoważna formule 6. Kontynuując to rozumowanie dochodzimy do wniosku, że wszystkie sześć formuł jest wzajemnie równoważnych. Wynika stąd, że pierwsza formuła jest tautologią. Zastosowanie. ... UOGÓLNIENIE. To samo rozumowanie mozna powtórzyć gdy zmiennne x,y i t są innego typu. W dowodzie nie wykorzystuje się, żadnych własności struktury integer (ani odpowiednio, innej struktury). Dlatego uogólnienie nie stanowi problemu.

Rozkładanie instrukcji przypisania

Wyrażenie po prawej stronie instrukcji przypisania może miec dowolnie złożona strukturę. W programach inżynierskich i naukowych zdarzaja się instrukcje przypisania zajmujące 3 i więcej linii.

Z drugiej strony na wiele instrukcji komputera możemy patrzec jak na instrukcje przypisania z jedna tylko operacją. Np. ciąg poleceń komputera

Operacja	Argumenty	czyli	stan pamięci
LOAD	x	Acc := x	Acc=x
ADD	y	Acc := Acc+y	Acc=x+y
STORE	t1	t1 := Acc	t1=x+y & Acc=x+y
LOAD	a	Acc := a	Acc=a & t1=x+y
MULT	t1	Acc := Acc * t1	Acc=a*(x+y)
STORE	z	z:= Acc	z=a*(x+y) & t1=x+y

realizuje instrukcję przypisania $z := a * (x + y)$. Można udowodnić, że każda instrukcja przypisania postaci $z := \tau$ może być rozłożona na ciąg instrukcji przypisania, z których każda zawiera conajwyżej jeden operator.

Twierdzenie 4.1. *Niech z będzie zmienną typu integer, a napis τ wyrażeniem całkowito-liczbowym. Istnieje ciąg instrukcji przypisania s taki, że instrukcja $z := \tau$ jest równoważna ciągowi poleceń przypisania s.*

Dowód. Dowód przebiega przez indukcję ze względu na długosc wyrażenia τ. Pozostawiamy go jako ćwiczenie. □

Wykorzystuj własności struktury danych

W powyższych przykładach dowodziliśmy tautologii. Oznacza to z jednej strony, że nasze rozumowania mają charakter uniwersalny. Wykorzystamy to w kolejnym rozdziale analizując procedurę *Swap*.
Z drugiej strony w rozumowaniach o programach bardzo przydatna jest znajomość własności struktur danych. Np. $x = 0 = x$ lub $x * 1 = x$ lub $x * 0 = 0$ itp.

Przykład 4.2. *Czy po wykonaniu programu P zachodzi warunek α?*
tu napisać dość długi program, który daje się skrócić

Wykorzystuj prawa algebry Boole'a

W podobny sposób mozemy wykorzystywac prawa algebry Boole'a.

Przykład 4.3. *tu napisać program działajacy na zmiennych boolean*

Mnożenie liczb zespolonych

Wcześniej zapytaliśmy czy potrafisz pomnożyc liczby zespolone wykonując tylko trzy mnożenia liczb rzeczywistych. Oto odpowiedź //

$$\mathfrak{R} \vdash \left\{ \begin{array}{l} \text{t1:=a+b;} \\ \text{t2:=c+d;} \\ \text{t3:=a*c;} \\ \text{t4:=b*d;} \\ \text{A:=t3-t4;} \\ \text{B:=t1*t2-t3-t4;} \end{array} \right\} (A = a * c - b * d \wedge B = a * d + b * c)$$

W powyższej formule $\mathfrak{R}$ oznacza zbiór formuł wyrażających łączność i przemienność dodawania, rozdzielnośc mnożenia względem dodawania, ...

Najsłabszy warunek wstępny – programu liniowego

W kilku przykładach jakie zbadaliśmy powyżej obliczyliśmy najsłabszy warunek wstępny programu liniowego. W rachunku takim wykorzystujemy dwa aksjomaty ax18 i ax19.

Najmocniejszy warunek końcowy – programu liniowego

Obliczenie symboliczne pozwala w wielu wypadkach wyznaczyć najmocniejszy warunek końcowy programu liniowego. Ale czy zawsze to się uda? Zbadajmy.

Zadanie 4.1. *W pewnych językach programowania dopuszcza się instrukcje przypisania równoczesnego np.* $x := \tau \ \& \ y := \mu \ \& \ z := \delta$. *Czy jest to istotne wzbogacenie języka? Inaczej mówiąc, czy instrukcje tego typu można rozłożyć na ciąg instrukcji przypisania pojedynczego?*

Zadanie 4.2. *Czy instrukcje przypisania równoczesnego tworza półgrupę?*

Zadanie 4.3. *Scharakteryzuj lewe zero przypisania s*

Zadanie 4.4. *Jak narysować program liniowy?*

Zadanie 4.5. ***Trudne****. Dla danego programu liniowego znajdź jego najszybszy odpowiednik. Optymalizacja*

4.4 Wprowadzenie – ćwiczenie w analizowaniu programów liniowych

Przeprowadzimy szczegółowy dowód tego, że program podany przez Antoniego Kreczmara oblicza iloczyn dwu macierzy kwadratowych o rozmiarze 2 x 2.

Znacie?
Znamy!
No to posłuchajcie ...

Niniejszy paragraf ma na celu pokazanie sposobu dowodzenia w rachunku programów jakim jest logika algorytmiczna. Zazwyczaj nie zastanawiamy się zbytnio nad tym, skąd się bierze nasza wiara w poprawność krótkiego algorytmu

$$
P: \left\{ \begin{array}{l}
\quad \textbf{var}\ a,b,c,d,e,f,g,h,A,B,C,D : ring; \\
\textbf{block} \\
\quad \textbf{var}\ p_1,p_2,p_3,p_4,p_5,p_6,p_7 : ring; \\
\textbf{begin} \\
\quad p_1 := a\cdot e; \\
\quad p_2 := b\cdot g; \\
\quad p_3 := (c+d-a)\cdot(h-f+e); \\
\quad p_4 := (c+d)\cdot(f-e); \\
\quad p_5 := (a-c)\cdot(h-f); \\
\quad p_6 := (b-c-d+a)\cdot h; \\
\quad p_7 := d\cdot(g-h+f-e); \\
\quad A := p_1+p_2; \\
\quad B := p_1+p_3+p_4+p_6; \\
\quad C := p_1+p_3+p_5+p_7; \\
\quad D := p_1+p_3+p_4+p_5 \\
\textbf{end}
\end{array} \right\}
$$

W pierwszej linijce zawarliśmy informację – przypomnienie, o tym, że wielkości $a, ..., h, A, B, C, D$ są elementami pewnego pierścienia.
Nasze zadanie polega na wykazaniu, że macierz

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix}$$

jest wynikiem mnożenia macierzy

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \cdot \begin{bmatrix} e & f \\ g & h \end{bmatrix}$$

Należy więc wykazać, że prawdziwa jest formuła

$$P \left(\begin{array}{l} A = a \cdot e + b \cdot g \wedge \\ B = a \cdot f + b \cdot h \wedge \\ C = c \cdot e + d \cdot g \wedge \\ D = c \cdot f + d \cdot h \end{array} \right)$$

Sprawdzenie

Rachujemy

$$P \left(\begin{array}{l} A = a \cdot e + b \cdot g \wedge \\ B = a \cdot f + b \cdot h \wedge \\ C = c \cdot e + d \cdot g \wedge \\ D = c \cdot f + d \cdot h \end{array} \right) \tag{4.1}$$

P oznacza program, wstawmy go na miejsce

Możemy pominąć deklaracje zmiennych. Nie są potrzebne w rachunkach jakie przeprowadzimy.

$$P : \left\{ \begin{array}{l} p_1 := a \cdot e; \\ p_2 := b \cdot g; \\ p_3 := (c + d - a) \cdot (h - f + e); \\ p_4 := (c + d) \cdot (f - e); \\ p_5 := (a - c) \cdot (h - f); \\ p_6 := (b - c - d + a) \cdot h; \\ p_7 := d \cdot (g - h + f - e); \\ A := p_1 + p_2; \\ B := p_1 + p_3 + p_4 + p_6; \\ C := p_1 + p_3 + p_5 + p_7; \\ D := p_1 + p_3 + p_4 + p_5 \end{array} \right\} \left(\begin{array}{l} A = a \cdot e + b \cdot g \wedge \\ B = a \cdot f + b \cdot h \wedge \\ C = c \cdot e + d \cdot g \wedge \\ D = c \cdot f + d \cdot h \end{array} \right) \tag{4.2}$$

stosujemy aksjomat instrukcji przypisania - eliminacja zmiennej D

$$\left\{\begin{array}{l} p_1 := a\cdot e; \\ p_2 := b\cdot g; \\ p_3 := (c+d-a)\cdot(h-f+e); \\ p_4 := (c+d)\cdot(f-e); \\ p_5 := (a-c)\cdot(h-f); \\ p_6 := (b-c-d+a)\cdot h; \\ p_7 := d\cdot(g-h+f-e); \\ A := p_1+p_2; \\ B := p_1+p_3+p_4+p_6; \\ C := p_1+p_3+p_5+p_7; \end{array}\right\} \left(\begin{array}{l} A = a\cdot e + b\cdot g \wedge \\ B = a\cdot f + b\cdot h \wedge \\ C = c\cdot e + d\cdot g \wedge \\ p_1+p_3+p_4+p_5 = c\cdot f + d\cdot h \end{array}\right) \tag{4.3}$$

eliminujemy zmienną C

$$\left\{\begin{array}{l} p_1 := a\cdot e; \\ p_2 := b\cdot g; \\ p_3 := (c+d-a)\cdot(h-f+e); \\ p_4 := (c+d)\cdot(f-e); \\ p_5 := (a-c)\cdot(h-f); \\ p_6 := (b-c-d+a)\cdot h; \\ p_7 := d\cdot(g-h+f-e); \\ A := p_1+p_2; \\ B := p_1+p_3+p_4+p_6; \end{array}\right\} \left(\begin{array}{l} A = a\cdot e + b\cdot g \wedge \\ B = a\cdot f + b\cdot h \wedge \\ p_1+p_3+p_5+p_7 = c\cdot e + d\cdot g \wedge \\ p_1+p_3+p_4+p_5 = c\cdot f + d\cdot h \end{array}\right) \tag{4.4}$$

eliminujemy zmienną B

$$\left\{\begin{array}{l} p_1 := a\cdot e; \\ p_2 := b\cdot g; \\ p_3 := (c+d-a)\cdot(h-f+e); \\ p_4 := (c+d)\cdot(f-e); \\ p_5 := (a-c)\cdot(h-f); \\ p_6 := (b-c-d+a)\cdot h; \\ p_7 := d\cdot(g-h+f-e); \\ A := p_1+p_2; \end{array}\right\} \left(\begin{array}{l} A = a\cdot e + b\cdot g \wedge \\ p_1+p_3+p_4+p_6 = a\cdot f + b\cdot h \wedge \\ p_1+p_3+p_5+p_7 = c\cdot e + d\cdot g \wedge \\ p_1+p_3+p_4+p_5 = c\cdot f + d\cdot h \end{array}\right) \tag{4.5}$$

eliminujemy zmienną A

$$\left\{\begin{array}{l} p_1 := a\cdot e; \\ p_2 := b\cdot g; \\ p_3 := (c+d-a)\cdot(h-f+e); \\ p_4 := (c+d)\cdot(f-e); \\ p_5 := (a-c)\cdot(h-f); \\ p_6 := (b-c-d+a)\cdot h; \\ p_7 := d\cdot(g-h+f-e); \end{array}\right\} \left(\begin{array}{l} p_1+p_2 = a\cdot e + b\cdot g \wedge \\ p_1+p_3+p_4+p_6 = a\cdot f + b\cdot h \wedge \\ p_1+p_3+p_5+p_7 = c\cdot e + d\cdot g \wedge \\ p_1+p_3+p_4+p_5 = c\cdot f + d\cdot h \end{array}\right) \tag{4.6}$$

eliminujemy zmienną p7

$$\left\{\begin{array}{l} p_1 := a\cdot e; \\ p_2 := b\cdot g; \\ p_3 := (c+d-a)\cdot(h-f+e); \\ p_4 := (c+d)\cdot(f-e); \\ p_5 := (a-c)\cdot(h-f); \\ p_6 := (b-c-d+a)\cdot h; \end{array}\right\} \left(\begin{array}{l} p_1+p_2 = a\cdot e + b\cdot g \wedge \\ p_1+p_3+p_4+p_6 = a\cdot f + b\cdot h \wedge \\ p_1+p_3+p_5+d\cdot(g-h+f-e) = \\ \qquad c\cdot e + d\cdot g \wedge \\ p_1+p_3+p_4+p_5 = \\ c\cdot f + d\cdot h \end{array}\right) \tag{4.7}$$

eliminujemy zmienną p6

$$\left\{\begin{array}{l} p_1 := a\cdot e; \\ p_2 := b\cdot g; \\ p_3 := (c+d-a)\cdot(h-f+e); \\ p_4 := (c+d)\cdot(f-e); \\ p_5 := (a-c)\cdot(h-f); \end{array}\right\} \left(\begin{array}{l} p_1+p_2 = a\cdot e + b\cdot g \wedge \\ p_1+p_3+p_4+ \\ (b-c-d+a)\cdot h = a\cdot f + b\cdot h \wedge \\ p_1+p_3+p_5+ \\ d\cdot(g-h+f-e) = c\cdot e + d\cdot g \wedge \\ p_1+p_3+p_4+p_5 = \\ c\cdot f + d\cdot h \end{array}\right) \tag{4.8}$$

eliminujemy zmienną p5

$$\left\{\begin{array}{l} p_1 := a\cdot e; \\ p_2 := b\cdot g; \\ p_3 := (c+d-a)\cdot(h-f+e); \\ p_4 := (c+d)\cdot(f-e); \end{array}\right\} \left(\begin{array}{l} p_1+p_2 = a\cdot e + b\cdot g \wedge \\ p_1+p_3+p_4+ \\ (b-c-d+a)\cdot h = a\cdot f + b\cdot h \wedge \\ p_1+p_3+ \\ (a-c)\cdot(h-f) + d\cdot(g-h+f-e) = \\ c\cdot e + d\cdot g \wedge \\ p_1+p_3+p_4+(a-c)\cdot(h-f) = \\ c\cdot f + d\cdot h \end{array}\right) \tag{4.9}$$

eliminujemy zmienną p4

$$\left\{\begin{array}{l} p_1 := a\cdot e; \\ p_2 := b\cdot g; \\ p_3 := (c+d-a)\cdot(h-f+e); \end{array}\right\} \left(\begin{array}{l} p_1+p_2 = a\cdot e + b\cdot g \wedge \\ p_1+p_3+(c+d)\cdot(f-e)+ \\ \quad (b-c-d+a)\cdot h = a\cdot f + b\cdot h \wedge \\ p_1+p_3+(a-c)\cdot(h-f)+ \\ \quad d\cdot(g-h+f-e) = c\cdot e + d\cdot g \wedge \\ p_1+p_3+(c+d)\cdot(f-e)+ \\ \quad (a-c)\cdot(h-f) = c\cdot f + d\cdot h \end{array}\right) \tag{4.10}$$

eliminujemy zmienną p3

$$\left\{ \begin{array}{l} p_1 := a\cdot e; \\ p_2 := b\cdot g; \end{array} \right\} \left(\begin{array}{l} p_1 + p_2 = a\cdot e + b\cdot g \wedge \\ p_1 + (c+d-a)\cdot(h-f+e) + (c+d)\cdot(f-e) + \\ \quad (b-c-d+a)\cdot h = a\cdot f + b\cdot h \wedge \\ p_1 + (c+d-a)\cdot(h-f+e) + (a-c)\cdot(h-f) + \\ \quad d\cdot(g-h+f-e) = c\cdot e + d\cdot g \wedge \\ p_1 + (c+d-a)\cdot(h-f+e) + (c+d)\cdot(f-e) + \\ \quad (a-c)\cdot(h-f) = c\cdot f + d\cdot h \end{array} \right) \tag{4.11}$$

eliminujemy zmienną p2

$$\left\{ \quad p_1 := a\cdot e; \right\} \left(\begin{array}{l} p_1 + b\cdot g = a\cdot e + b\cdot g \wedge \\ p_1 + (c+d-a)\cdot(h-f+e) + (c+d)\cdot(f-e) + \\ \quad (b-c-d+a)\cdot h = a\cdot f + b\cdot h \wedge \\ p_1 + (c+d-a)\cdot(h-f+e) + (a-c)\cdot(h-f) + \\ \quad d\cdot(g-h+f-e) = c\cdot e + d\cdot g \wedge \\ p_1 + (c+d-a)\cdot(h-f+e) + (c+d)\cdot(f-e) + \\ \quad (a-c)\cdot(h-f) = c\cdot f + d\cdot h \end{array} \right) \tag{4.12}$$

eliminujemy zmienną p1

$$\{\} \left(\begin{array}{l} a\cdot e + b\cdot g = a\cdot e + b\cdot g \wedge \\ a\cdot e + (c+d-a)\cdot(h-f+e) + (c+d)\cdot(f-e) + \\ \quad (b-c-d+a)\cdot h = a\cdot f + b\cdot h \wedge \\ a\cdot e + (c+d-a)\cdot(h-f+e) + (a-c)\cdot(h-f) + \\ \quad d\cdot(g-h+f-e) = c\cdot e + d\cdot g \wedge \\ a\cdot e + (c+d-a)\cdot(h-f+e) + (c+d)\cdot(f-e) + \\ \quad (a-c)\cdot(h-f) = c\cdot f + d\cdot h \end{array} \right) \tag{4.13}$$

teraz pora na upraszczanie, blok opustoszał

$$\{\} \left(\begin{array}{l} a\cdot e + b\cdot g = a\cdot e + b\cdot g \wedge \\ a\cdot e + (c+d-a)(h-f+e) + (c+d)(f-e) + (b-c-d+a)h = a\cdot f + b\cdot h \wedge \\ a\cdot e + (c+d-a)(h-f+e) + (a-c)(h-f) + d\cdot(g-h+f-e) = c\cdot e + d\cdot g \wedge \\ a\cdot e + (c+d-a)(h-f+e) + (c+d)(f-e) + (a-c)(h-f) = c\cdot f + d\cdot h \end{array} \right) \tag{4.14}$$

upraszczamy, pozbywamy się pustego bloku

$$\left(\begin{array}{l} a\cdot e + b\cdot g = a\cdot e + b\cdot g \wedge \\ a\cdot e + (c+d-a)\cdot(h-f+e) + (c+d)\cdot(f-e) + (b-c-d+a)\cdot h = a\cdot f + b\cdot h \wedge \\ a\cdot e + (c+d-a)\cdot(h-f+e) + (a-c)\cdot(h-f) + d\cdot(g-h+f-e) = c\cdot e + d\cdot g \wedge \\ a\cdot e + (c+d-a)\cdot(h-f+e) + (c+d)\cdot(f-e) + (a-c)\cdot(h-f) = c\cdot f + d\cdot h \end{array} \right) \tag{4.15}$$

upraszczam

$$\left(\begin{array}{l} a\cdot e+b\cdot g=a\cdot e+b\cdot g\wedge \\ a\cdot e+c\ h-c\ f+c\ e+d\ h-df+de-ah+af-ae+cf-ce+df-de+bh \\ -ch-dh+ah=a\cdot f+b\cdot h\wedge \\ a\cdot e+(c+d-a)\cdot(h-f+e)+(a-c)\cdot(h-f)+d\cdot(g-h+f-e)=c\cdot e+d\cdot g\wedge \\ a\cdot e+(c+d-a)\cdot(h-f+e)+(c+d)\cdot(f-e)+(a-c)\cdot(h-f)=c\cdot f+d\cdot h \end{array}\right) \tag{4.16}$$

upraszczam

$$\left(\begin{array}{l} a\cdot e+b\cdot g=a\cdot e+b\cdot g\wedge \\ \not{a}e+\not{c}h-\not{c}f+\not{c}e+\not{d}h-\not{d}f+\not{d}e-\not{a}h+af-\not{a}e+\not{c}f-\not{c}e+\not{d}f-\not{d}e+bh \\ -\not{c}h-\not{d}h+\not{a}h=a\cdot f+b\cdot h\wedge \\ a\cdot e+(c+d-a)\cdot(h-f+e)+(a-c)\cdot(h-f)+d\cdot(g-h+f-e)=c\cdot e+d\cdot g\wedge \\ a\cdot e+(c+d-a)\cdot(h-f+e)+(c+d)\cdot(f-e)+(a-c)\cdot(h-f)=c\cdot f+d\cdot h \end{array}\right) \tag{4.17}$$

upraszczam

$$\left(\begin{array}{l} a\cdot e+b\cdot g=a\cdot e+b\cdot g\ \wedge \\ a\cdot f+b\cdot h=a\cdot f+b\cdot h\ \wedge \\ a\cdot e+(c+d-a)\cdot(h-f+e)+(a-c)\cdot(h-f)+d\cdot(g-h+f-e)=c\cdot e+d\cdot g\wedge \\ a\cdot e+(c+d-a)\cdot(h-f+e)+(c+d)\cdot(f-e)+(a-c)\cdot(h-f)=c\cdot f+d\cdot h \end{array}\right) \tag{4.18}$$

upraszczam

$$\left(\begin{array}{l} a\cdot e+b\cdot g=a\cdot e+b\cdot g\ \wedge \\ a\cdot f+b\cdot h=a\cdot f+b\cdot h\ \wedge \\ \not{a}\cdot e+\not{c}h-\not{c}f+ce+\not{d}h-\not{d}f+\not{d}e-\not{a}h+\not{a}f-\not{a}e+\not{a}h-\not{a}f-\not{c}h+\not{c}f+dg \\ -\not{d}h+\not{d}f-\not{d}e=c\cdot e+d\cdot g\wedge \\ a\cdot e+(c+d-a)\cdot(h-f+e)+(c+d)\cdot(f-e)+(a-c)\cdot(h-f)=c\cdot f+d\cdot h \end{array}\right) \tag{4.19}$$

upraszczam

$$\left(\begin{array}{l} a\cdot e+b\cdot g=a\cdot e+b\cdot g\ \wedge \\ a\cdot f+b\cdot h=a\cdot f+b\cdot h\ \wedge \\ c\cdot e+d\cdot g=c\cdot e+d\cdot g\ \wedge \\ a\cdot e+(c+d-a)\cdot(h-f+e)+(c+d)\cdot(f-e)+(a-c)\cdot(h-f)=c\cdot f+d\cdot h \end{array}\right) \tag{4.20}$$

upraszczam

$$\left(\begin{array}{l} a\cdot e+b\cdot g=a\cdot e+b\cdot g\ \wedge \\ a\cdot f+b\cdot h=a\cdot f+b\cdot h\ \wedge \\ c\cdot e+d\cdot g=c\cdot e+d\cdot g\ \wedge \\ a\cdot e+ch-cf+ce+dh-df+de-ah \\ +af-ae+cf-ce+df-de+ah-af-ch+cf=c\cdot f+d\cdot h \end{array}\right) \tag{4.21}$$

upraszczam

$$\left(\begin{array}{l} a\cdot e+b\cdot g=a\cdot e+b\cdot g\ \wedge \\ a\cdot f+b\cdot h=a\cdot f+b\cdot h\ \wedge \\ c\cdot e+d\cdot g=c\cdot e+d\cdot g\ \wedge \\ \not{a}\cdot e+\not{c}h-\not{c}f+\not{c}e+dh-\not{d}f+\not{d}e-\not{a}h \\ +\ \not{a}f-\not{a}e+\not{c}f-\not{c}e+\not{d}f-\not{d}e+\not{a}h-\not{a}f-\not{c}h+cf=c\cdot f+d\cdot h \end{array}\right) \tag{4.22}$$

upraszczam

$$\left(\begin{array}{l} a\cdot e+b\cdot g=a\cdot e+b\cdot g\ \wedge \\ a\cdot f+b\cdot h=a\cdot f+b\cdot h\ \wedge \\ c\cdot e+d\cdot g=c\cdot e+d\cdot g\ \wedge \\ d\cdot h+c\cdot f=c\cdot f+d\cdot h \end{array}\right) \tag{4.23}$$

Wszystkie te formuły 4.1 – 4.23 są wzajemnie równoważne. Formuła ostatnia 4.23 jest prawdziwa w pierścieniu (pamiętasz?, założyliśmy, że obiekty typu ring są z pierścienia), wobec tego formuła 4.2 jest prawdziwa. Oznacza to, że nasz program P oblicza iloczyn macierzy. Udowodniliśmy więc następujący lemat.

Lemat 4.2. *Program P oblicza iloczyn macierzy* $\left[\begin{array}{cc} a & b \\ c & d \end{array}\right]\cdot\left[\begin{array}{cc} e & f \\ g & h \end{array}\right]$ *wykonując siedem mnożeń i 24 dodawania.*

Ta obserwacja poczyniona przez Antoniego Kreczmara w 1975 r.[?] ma znaczenie ponieważ pozwala poprawić wynik Strassena[]. Ta odmiana algorytmu Strassena ma zbliżony koszt czasowy, lecz jest znacznie bardziej oszczędna w użyciu pamięci. Zamiast $\frac{11}{3}n^2$ dodatkowych miejsc pamięci potrzebnych do obliczenia iloczynu macierzy $n\times n$ algorytm w wersji Kreczmara zużywa $\frac{2}{3}n^2$ komórek pamięci.
Zobacz też ...

Rozdział 5

Programowanie elementarne $\mathcal{L}_3$

5.1 Język $\mathcal{L}_3$

Ten język ma te same deklaracje co język $\mathcal{L}_2$. Zbiór instrukcji atomowych jest także ten sam co w języku $\mathcal{L}_2$. Natomiast pojawiają się dwie operacje programotwórcze: instrukcja warunkowa **if** i instrukcja powtarzania **for**.

$$\mathcal{L}_2 \subsetneq \mathcal{L}_3$$

Przykład 5.1. *Czy zgadniesz co oblicza ten algorytm?*

```
s:=0;  j:= 0; c:=1;
for i:= 1 to k
do
   c:=c*x/i;
   if i = 2 *j then j:=j+1; else  s := s+ c fi
od
```

5.1.1 Składnia

Zbiór deklaracji dopuszczalnych w języku $\mathcal{L}_3$ jest ten sam co w języku $\mathcal{L}_2$.

$$D_3 = D_2$$

Zbiór instrukcji atomowych jest ten sam.

$$i_3 = i_2$$

Przypomnijmy, są to instrukcje przypisania oraz instrukcje drukowania.
Zbiór instrukcji jest najmniejszym zbiorem zawierającym instrukcje atomowe i zamkniętym ze względu na dwie operacje: p1) jeśli napis γ jest wyrażeniem boolowskim, a napisy K oraz M są skończonymi ciągami instrukcji to instrukcją jest także napis postaci

$$\textbf{if } \gamma \textbf{ then } K \textbf{ else } M \textbf{fi},$$

Instrukcje tej postaci nazywamy instrukcjami warunkowymi. W przypadku gdy ciąg M jest pusty, to instrukcję warunkowa mozna zapisać krócej

$$\textbf{if } \gamma \textbf{ then } K \textbf{ fi},$$

p2) jeśli $i$jest zmienną typu integer, a napisy A oraz C sa wyrażeniami arytmetycznymi, to napis postaci

$$\textbf{for } i := A \textbf{ to } C \textbf{ do } K \textbf{ od}$$

jest instrukcją powtarzania.

Zbiór wyrażeń poprawnie zbudowanych $\mathcal{W}$ języka $\mathcal{L}_3$ jest sumą kilku zbiorów: zbioru wyrażeń arytmetycznych $\mathcal{WA}$ (inaczej, termów), zbioru formuł (inaczej zbioru wyrażeń boolowskich) $\mathcal{WB}$, zbioru instrukcji $\mathcal{WI}$. zbioru deklaracji $\mathcal{WD}$, zbioru programów $\mathcal{WP}$.

$$\mathcal{W} = \mathcal{WA} \cup \mathcal{WB} \cup \mathcal{WI} \cup \mathcal{WD} \cup \mathcal{WP}$$

Napis τ jest wyrażeniem arytmetycznym, napis x jest zmienną całkowitą zadeklarowaną w programie.

Składnia 5.2. *Instrukcje warunkowe mają postać*

$$\textbf{if } \gamma \textbf{ then } K \textbf{ else } M \textbf{ fi}$$

gdzie napisy K oraz M oznaczają ciągi instrukcji.
Dopuszcza się instrukcje warunkowe skrócone postaci

$$\textbf{if } \gamma \textbf{ then } K \textbf{ fi}.$$

Składnia 5.3. *Instrukcje powtarzania mają postać*

$$\boxed{\textbf{for } \; i := A \; \textbf{ to } \; B \textbf{ do } I \textbf{ od}}$$

gdzie i jest jakąś zmienną, A oraz B to wyrażenia arytmetyczne, zaś I jest ciągiem instrukcji. Dla dalszych rozważań warto przyjąć, że zmienna i kontrolująca instrukcję powtarzania może wystąpić w instrukcjach I tylko po prawej stronie w instrukcjach przypisania. Inaczej mówiąc, ciąg instrukcji I nie zmienia wartości zmiennej i.

Składnia 5.4. *Zbiór Ins instrukcji jest najmniejszym zbiorem napisów takim, że*

(i) instrukcje atomowe czyli instrukcje przypisania i instrukcje drukowania należą do zbioru Ins,

(ii) jeśli do zbioru Ins należą napisy K, M to do zbioru Ins należą też napisy postaci

$$\textbf{begin}\, K; M \,\textbf{end}$$

$$\textbf{if}\, \gamma \,\textbf{then}\, K \,\textbf{else}\, M \,\textbf{fi}$$

$$\textbf{while}\, \gamma \,\textbf{do}\, M \,\textbf{od}$$

5.2 Semantyka

Co robi program elementarny? Jakie sa efekty jego działania?
Widzieliśmy jakie są efekty instrukcji write. Jakie będą wyniki instrukcji przypisania i instrukcji iteracyjnej?
Formuły algorytmiczne.
Niech α oznacza wyrażenie boolowskie. Niech I będzie ciągiem instrukcji.
Wyrażenie $I\,\alpha$ jest formułą (algorytmiczną). Czytamy, Po wykonaniu programu I zachodzi warunek α.
Właśnie formuły algorytmiczne zostaną użyte w aksjomatach opisujących semantykę programów z języka $\mathcal{L}_2$. Znaczenie instrukcji języka $\mathcal{L}_2$ określają następujące schematy aksjomatów
Schemat aksjomatów instrukcji przypisania

$$\{x := \tau\}\alpha(x) \Leftrightarrow \alpha(x/\tau)$$

co czytamy, po wykonaniu instrukcji przypisania $\{x := \tau\}$ warunek $\alpha(x)$ zachodzi wtedy i tylko wtedy gdy przed wykonaniem tej instrukcji zachodził warunek $\alpha(x/\tau)$.
Wyrażenie $\alpha(x/\tau)$ powstaje z wyrażenia $\alpha(x)$ w ten sposób, że w warunku $\alpha(x)$ odnajdujemy wszystkie wolne wystąpienia zmiennej x, i równocześnie zastępujemy je wyrażeniem arytmetycznym τ.
Znaczenie instrukcji **for** opisują trzy schematy aksjomatów:

f1) $(B < A) \Rightarrow (\{\textbf{for } i := A \textbf{ to } B \textbf{ do } I \textbf{ od}\}\alpha \Leftrightarrow \alpha)$

f2) $(B = A) \Rightarrow (\{\textbf{for } i := A \textbf{ to } B \textbf{ do } I \textbf{ od}\}\alpha \Leftrightarrow (\{i := A;\, I\}\alpha))$

f3) $(\{\textbf{for } i := A \textbf{ to } B + 1 \textbf{ do } I \textbf{ od}\}\alpha \Leftrightarrow$
$\{\textbf{for } i := A \textbf{ to } B \textbf{ do } I \textbf{ od};\, i := i + 1;\, I\}\alpha)$

Przeczytajmy jeszcze raz co znaczą te trzy schematy:

f1) Jeśli wartośc poczatkowa A jest większa od wartości koncowej B to instrukcja **for** jest instrukcją pustą,

f2) Jeśli wartości, początkowa A i końcowa B są równe to instrukcja iteracyjna for wykonuje ciąg instrukcji I jeden raz,

f3) wykonanie instrukcji "*dla i od* A *do* $B + 1$ *powtarzaj instrukcje* I" jest równoważne wykonaniu programu {"*dla i od* A *do* B *powtarzaj instrukcje* I", *potem instrukcje* $i := i + 1;\ I$ }.

Znaczenie instrukcji warunkowej opisują formuły o następującym schemacie

$$\{\textbf{if } \gamma \textbf{ then K } \textbf{ else M } \textbf{ fi }\}\alpha \Leftrightarrow ((\gamma \wedge \mathrm{K}\,\alpha) \vee (\neg\gamma \wedge \mathrm{M}\,\alpha))$$

w powyższym schemacie γ oznacza wyrażenie boolowskie, a wyrażenia K oraz M oznaczają ciągi instrukcji.

5.3 Abstrakcyjna wirtualna maszyna $\mathcal{K}_3$

Wykonywanie programu P z języka L_3 polega na wykonywaniu kolejnych instrukcji zapisanych w rekordzie aktywacji programu P.
Rekord aktywacji programu
Na rekord aktywacji składają się dwa elementy: pamięć i ciąg instrukcji.

Pamięć	Ciąg instrukcji

Będziemy też używać nazwy *konfiguracja* lub *stan obliczenia.*
Obliczenie programu to ciąg stanów $\{s_i\}_i$ taki, że 1°s_0 jest stanem początkowym rekordu aktywacji programu, 2°jeśli w stanie s_i ciąg instrukcji do wykonania jest niepusty i I oznacza pierwszą instrukcję tego ciągu, to następny stan s_{i+1} ...

5.4 Analiza programów

Lemat 5.1. *Jeśli $A \leq B$, to po wykonaniu programu* ***for*** *i:= A* ***to*** *B* ***do*** *I* ***od*** *zachodzi $i = B$*

Dowód. Dowód przebiega przez indukcję.
Gdy $A = B$ to na mocy schematu f2) wykonano instrukcję $i := A$, a potem ciąg I instrukcji, które nie zmieniają wartości zmiennej i.
Załóżmy, że teza lematu jest prawdziwa dla $B = k$. Zauważmy, że prawdziwa jest implikacja $(i = k) \Rightarrow \{i := i + 1; I\}(i = k + 1)$. Teraz wykorzystamy regułę wnioskowania $\boxed{\text{R2:}\frac{\alpha \Rightarrow \beta}{M\alpha \Rightarrow M\beta}}$ i otrzymujemy prawdziwą implikację

$$\left\{\begin{array}{l}\textbf{for}\ \text{i:= A}\ \textbf{to}\ \text{B}\\ \textbf{do}\ \text{I}\ \textbf{od}\end{array}\right\}(i = k) \Rightarrow \left\{\begin{array}{l}\textbf{for}\ \text{i:= A}\ \textbf{to}\ \text{B}\\ \textbf{do}\ \text{I}\ \textbf{od}\end{array}\right\}\left\{\begin{array}{l}\text{i:=i+1;}\\ \text{I}\end{array}\right\}(i = k{+}1).$$

Teraz stosujemy schemat f3) i mamy

$$\{\textbf{for}\ i := A\ \textbf{to}\ B\ \textbf{do}\ I\ \textbf{od}\}(i = k) \Rightarrow \{\textbf{for}\ i := A\ \textbf{to}\ B{+}1\ \textbf{do}\ I\ \textbf{od}\}(i = k{+}1).$$

A więc dla każdych wartości A i B, jeśli $A \leq B$ to wartość zmiennej i po wykonaniu instrukcji **for** i:= A **to** B **do** I **od** wynosi B. □

Wniosek 5.2. *Obliczenie każdego programu P z języka $\mathcal{L}_2$ jest skończone.*

Dowód. Dowód przez indukcję ze wzgledu na długość programu P. Przeprowadź samodzielnie ten dowód. □

Twierdzenie 5.3. *Niech $\tau(i)$ będzie wyrażeniem arytmetycznym.*
Program

$$K : \left\{\begin{array}{l} s := 0;\\ \textbf{for}\ i := 0\ \textbf{to}\ n\ \textbf{do}\ s := s + \tau(i)\ \textbf{od}\end{array}\right\}$$

ma tę własność, że po wykonaniu programu K spełniony jest warunek końcowy $s = \sum_{i=0}^{n} \tau(i)$. Inaczej mówiąc, program K oblicza wartość wyrażenia $\sum_{i=0}^{n} \tau(i)$ i przypisuje ją zmiennej s.

Dowód. Dowód przebiega przez indukcję ze względu na wartość n.
(*Baza*). Dla $n = 0$ należy udowodnić, że

$$\{s := 0; \textbf{for } i := 0 \textbf{ to } 0 \textbf{ do } s := s + \tau(i) \textbf{ od}\}(s = \sum_{i=0}^{0} \tau(i)).$$

Prawdziwość tej formuły algorytmicznej wykazujemy stosując własność f1) instrukcji for oraz definicję znaku Σ.
(*Krok indukcyjny.*) Teraz mamy wykazać, że dla każdego n prawdziwa jest implikacja $(T(n) \Rightarrow T(n+1))$ tj. formuła algorytmiczna

$$\big[\ \{s := 0; \textbf{for } i := 0 \textbf{ to } n \textbf{ do } s := s + \tau(i) \textbf{ od}\}(s = \sum_{i=0}^{n} \tau(i)) \Rightarrow$$

$$\{s := 0; \textbf{for } i := 0 \textbf{ to } n+1 \textbf{ do } s := s + \tau(i) \textbf{ od}\}(s = \sum_{i=0}^{n+1} \tau(i))\ \big]$$

Zacznijmy od tautologii

$$s = \sum_{i=0}^{n} \tau(i) \Rightarrow s = \sum_{i=0}^{n} \tau(i) \tag{5.1}$$

Ta formuła jest równoważna następującej

$$s = \sum_{i=0}^{n} \tau(i) \Rightarrow s + \tau(n+1) = \sum_{i=0}^{n} \tau(i) + \tau(n+1) \tag{5.2}$$

Z definicji znaku Σ otzymujemy formułę

$$s = \sum_{i=0}^{n} \tau(i) \Rightarrow s + \tau(i+1) = \sum_{i=0}^{n+1} \tau(i) \tag{5.3}$$

Stosując aksjomat instrukcji przypisania otrzymujemy

$$s = \sum_{i=0}^{n} \tau(i) \Rightarrow \{i := n+1\}\big(s + \tau(i) = \sum_{i=0}^{n+1} \tau(i)\big) \tag{5.4}$$

Ponownie stosujemy ten sam aksjomat

$$s = \sum_{i=0}^{n} \tau(i) \Rightarrow \left\{ \begin{array}{l} i := n+1; \\ s := s + \tau(i) \end{array} \right\} \big(s = \sum_{i=0}^{n+1} \tau(i)\big) \tag{5.5}$$

Zastosujmy regułę wnioskowania $\boxed{\text{R2:}\frac{\alpha \Rightarrow \beta}{M\alpha \Rightarrow M\beta}}$

$$\left\{ \begin{array}{l} s := 0; \\ \textbf{for } i := 0 \textbf{ to } n \\ \textbf{ do} \\ \quad s := s + \tau(i) \\ \textbf{od} \end{array} \right\} \big(s = \sum_{i=0}^{n} \tau(i)\big) \Rightarrow \left\{ \begin{array}{l} s := 0; \\ \textbf{for } i := 0 \textbf{ to } n \\ \textbf{ do} \\ \quad s := s + \tau(i) \\ \textbf{od} \end{array} \right\} \left\{ \begin{array}{l} i := n+1; \\ s := s + \tau(i) \end{array} \right\} \big(s = \sum_{i=0}^{n+1} \tau(i)\big) \tag{5.6}$$

Stosując aksjomat f3) instrukcji **for**, otrzymujemy, że dlakażdej liczby naturalnej n prawdziwa jest implikacja

$$\left\{\begin{array}{l} s := 0; \\ \textbf{for } i := 0 \textbf{ to } n \\ \textbf{ do} \\ \quad s := s + \tau(i) \\ \textbf{od} \end{array}\right\} \left(s = \sum_{i=0}^{n} \tau(i)\right) \Rightarrow \left\{\begin{array}{l} s := 0; \\ \textbf{for } i := 0 \textbf{ to } n+1 \\ \textbf{ do} \\ \quad s := s + \tau(i) \\ \textbf{od} \end{array}\right\} \left(s = \sum_{i=0}^{n+1} \tau(i)\right) \tag{5.7}$$

co należało dowieść. Wobec tego, dla każdej liczby naturalnej n zachodzi

$$\left\{\begin{array}{l} s := 0; \\ \textbf{for } i := 0 \textbf{ to } n \\ \textbf{ do} \\ \quad s := s + \tau(i) \\ \textbf{od} \end{array}\right\} \left(s = \sum_{i=0}^{n} \tau(i)\right)$$

□

Własności programów z języka $\mathcal{L}_2$ c.d.

Nie tylko suma jest obliczalna w języku $\mathcal{L}_2$. Zobaczmy, że można udowodnić, że także iloczyny, alternatywy i koniunkcje oraz kilka innych operacji daje się zaprogramować w języku $\mathcal{L}_2$.

Twierdzenie 5.4. *Niech $\tau(i)$ będzie wyrażeniem arytmetycznym. Program*

$$\{p := 1; \textbf{\textit{for}}\ i := 1\ \textbf{\textit{to}}\ n\ \textbf{\textit{do}}\ p := \tau(i) * p\ \textbf{\textit{od}}\}(p = \prod_{i=1}^{n} \tau(i))$$

tzn. program ten oblicza wartość iloczynu tych wielkości $\tau(1) * \tau(2) * \ \ldots\ * \tau(n)$.

Podobne spostrzeżenie odnosi się do kwantyfikatora ogólnego

Twierdzenie 5.5. *Niech i będzie zmienną typu integer. Niech napis $\alpha(i)$ oznacza formułę. Niech $P(i)$ oznacza ciąg instrukcji. Poniższa reguła wnioskowania jest poprawna w systemie Loglan*

$$\mathfrak{Loglan} \vdash \frac{\forall_{1 \le i \ne j \le n}(\{P(j);\, P(i)\}\alpha(i) \Leftrightarrow \{P(i)\}\alpha(i))}{\forall_{i=1}^{n} \{P(i)\}\, \alpha(i) \Leftrightarrow \left\{\begin{array}{l} \textbf{for } i := 1 \textbf{ to } n \textbf{ do} \\ \quad P(i) \\ \textbf{od} \end{array}\right\} \forall_{i=1}^{n}\, \alpha(i)}.$$

Sens tego twierdzenia jest następujący: *jeżeli dla $i \ne j$ wykonanie programów P(i) oraz P(j) nie wpływają ...* Jak korzystamy z tych twierdzeń?

Przykład 5.5. *Mamy napisać program obliczający $\sum_{i=1}^{k} i^2$.*
Korzystając z twierdzenia 5.3 możemy napisać program
$P : \left\{\begin{array}{l} s := 0; \\ \textbf{for } i := 0 \textbf{ to } k \textbf{ do } s := s + i^2 \textbf{ od} \end{array}\right\}$
i nie musimy powtarzać dowodu, że zachodzi $P(s = \sum_{i=1}^{k} i^2)$.

Rozpatrzmy zadanie trochę trudniejsze

Przykład 5.6. *Należy obliczyć sumę* $\sum_{i=1}^{k} \frac{x^i}{i!}$.

Zaczynamy od ponownego wykorzystania twierdzenia 5.3.

$$P: \left\{ \begin{array}{l} s := 0; \\ \textbf{for } i := 0 \textbf{ to } n \textbf{ do } s := s + \frac{x^i}{i!} \textbf{ od} \end{array} \right\}$$

Możemy ten program przekształcić następująco.

$$P: \left\{ \begin{array}{l} s := 0; \\ \textbf{for } i := 0 \textbf{ to } n \textbf{ do} \\ \quad s1 := x^i; \\ \quad s2 := i!; \\ \quad s := s + \frac{s1}{s2} \\ \textbf{od} \end{array} \right\}$$

No tak, ale wyrażenia x^i oraz $i!$ nie należą do naszego języka, ponieważ nie ma w nim działania potęgowanie ani działania silnia. Wykorzystamy inne twierdzenie, odmianę twierdzenia 5.3. (Zastanów się jak je sformułować i udowodnić. Potraktuj to zadanie jako ćwiczenie.)

Korzystając z tego nowego twierdzenia piszemy:

$$P: \left\{ \begin{array}{l} s := 0; \\ \textbf{for } i := 0 \textbf{ to } n \textbf{ do} \\ \quad s1 := 1; \\ \quad \textbf{for } j := 1 \textbf{ to } i \textbf{ do} \\ \quad\quad s1 := s1 * x \\ \quad \textbf{od}; \\ \quad s2 := 1; \\ \quad \textbf{for } k := 1 \textbf{ to } i \textbf{ do} \\ \quad\quad s1 := s1 * k \\ \quad \textbf{od}; \\ \quad s := s + \frac{s1}{s2} \\ \textbf{od} \end{array} \right\}$$

Dowód prawdziwości formuły $P(s = \sum_{i=1}^{k} \frac{x^i}{i!})$ konstruujemy korzystając z lematów 5.6 i 5.7 przytoczonych poniżej.

Lemat 5.6. *(Obliczanie potęgi)*

$$\forall_x \forall_i \left\{ \begin{array}{l} \textit{s1:=1;} \\ \textbf{\textit{for}} \ \textit{j:=1} \ \textbf{\textit{to}} \ \textit{i} \ \textbf{\textit{do}} \\ \quad \textit{s1:=s1* x} \\ \textbf{\textit{od;}} \end{array} \right\} (s1 = x^i)$$

Lemat ten upoważnia nas do zastąpienia (nieformalnej) instrukcji {s1:=x^i} przez powyższy program obliczania potęgi. Idzie o to, że

$$\{s1 := x^i\}(s1 = x^i) \Leftrightarrow \left\{ \begin{array}{l} \text{s1:=1;} \\ \textbf{for } \text{j:=1 } \textbf{to } \text{i } \textbf{do} \\ \quad \text{s1:=s1* x} \\ \textbf{od;} \end{array} \right\} (s1 = x^i)$$

Lemat 5.7. *(Obliczanie silni)*

$$\forall_i \left\{ \begin{array}{l} \textit{s2:=1;} \\ \textbf{\textit{for}}\ \ \textit{k:=1}\ \ \textbf{\textit{to}}\ \ i\ \ \textbf{\textit{do}} \\ \quad \textit{s2:=s2* k} \\ \textbf{\textit{od;}} \end{array} \right\} (s2 = i!)$$

Warto zauważyć, że poprawny program P można zastąpić innym, równoważnym mu programem Q

$$Q: \left\{ \begin{array}{l} s := 0; s1 := 1; s2 := 1; \\ \textbf{for}\ i := 0\ \textbf{to}\ n\ \textbf{do} \\ \quad s := s + \frac{s1}{s2}; \\ \quad s1 := s1 * x; \\ \quad s2 := s2 * (i+1) \\ \textbf{od} \end{array} \right\}$$

Zechciej porównać koszty wykonania obu tych programów.

5.5 funkcje elementarnie rekurencyjne i pierwotnie rekurencyjne

Każda funkcja pierwotnie rekurencyjna (zob. [13]) może byc obliczona przez odpowiedni program w języku L2 w dziedzinie unsigned integer. Na czym polega definiowanie ?
Pierwotnie rekurencyjne sa funkcje: następnik, ...
Jeśli pierwotnie rekurencyjne są funkcje $g(x)$ oraz $h(n,x,y)$ to pierwotnie rekurencyjna jest funkcja $f(n,x)$ określona indukcyjnie następującymi wzorami
$f(0,x) \stackrel{df}{=} g(x)$
$f(n+1,x) \stackrel{df}{=} h(n,x,f(n,x))$
Sprawdź, że program

$$\left\{ \begin{array}{l} \text{i:=0; aux:=g(x);} \\ \text{for i :=1 to n do} \\ \quad \text{aux := h(i-1,x,aux)} \\ \text{od} \end{array} \right\}$$

oblicza poprawnie funkcję $f(n,x)$.
Należy udowodnić, że

$$\left\{ \begin{array}{l} \text{i:=0; aux:=g(x);} \\ \text{for i :=1 to n do} \\ \quad \text{aux := h(i-1,x,aux)} \\ \text{od} \end{array} \right\} (aux = f(n,x))$$

Rzeczywiście. Dla $n = 0$ mamy

$$\{\ \text{i:=0; aux:=g(x);}\ \}\,(aux = f(n,x))$$

Ponieważ $f(0,x) = g(x)$.
Załóżmy, że teza jest spełniona dla każdego x i dla $n < k$. Zbadajmy wyrażenie

$$\left\{\begin{array}{l} \text{i:=0; aux:=g(x);} \\ \text{for i :=1 to k+1 do} \\ \text{aux := h(i,x,aux)} \\ \text{od} \end{array}\right\} (aux = f(k+1,x))$$

Z własnośći f3) otrzymujemy formułę równoważną

$$\left\{\begin{array}{l} \text{i:=0; aux:=g(x);} \\ \text{for i :=1 to k do} \\ \text{aux := h(i-1,x,aux)} \\ \text{od;} \\ \text{i:=i+1;} \\ \text{aux:=h(i-1,x,aux);} \end{array}\right\} (aux = f(k+1,x))$$

Łatwo dostrzec, że końcowa wartość zmiennej aux jest równa $h(k,x,f(k.x))$ czyli jest równa $f(k+1,x)$.

5.6 Podsumowanie

Światły programista wie, że następujące funkcjonały tzn. operacje na funkcjach zwracające funkcje sa programowalne w języku $\mathcal{L}_2$:

Suma: $\sum_{i=1}^{n} \tau(i)$	s:=0;
	for i:= 1 **to** n **do**
	s:= s+τ(i)
	done
	$(s = \sum_{i=1}^{n} \tau(i))$

Iloczyn: $\prod_{i=1}^{n} \tau(i)$	s:=1;
	for i:= 1 **to** n **do**
	s:= s*τ(i)
	done
	$(s = \prod_{i=1}^{n} \tau(i))$

Maksimum: $\mathbf{Max}_{i=1}^{n}\tau(i))$	s:=-∞; **for** i:= 1 **to** n **do** s:= **if** s>τ(i) **then** s:= τ(i) **fi** **done** $(s = \mathbf{Max}_{i=1}^{n}\tau(i))$
Minimum: $\mathbf{Min}_{i=1}^{n}\tau(i))$	s:=∞; **for** i:= 1 **to** n **do** s:= min(s,τ(i)) **done** $(s = \mathbf{Min}_{i=1}^{n}\tau(i))$

alternatywa : $\bigvee_{i=1}^{n} \alpha(i)$	s:=false;
	for i:= 1 **to** n **do**
	if s:= s $\vee \alpha(i)$
	done
	$(s = \bigvee_{i=1}^{n} \alpha(i)$

koniunkcja:$\bigwedge_{i=1}^{n} \alpha(i)$	s:=true;
	for i:= 1 **to** n **do**
	s:= s $\wedge \alpha(i)$
	done
	$(s = \bigwedge_{i=1}^{n} \alpha(i)$

kwantyfikator ograniczony ogólny : $\forall_{i=1}^{n} \alpha(i)$	s:=true;
	for i:= 1 **to** n **do**
	if $\neg \alpha(i)$
	then s:= false ; exit fi
	done
	$(s = \forall_{i=1}^{n} \alpha(i)$

kwantyfikator ograniczony szczegółowy :$\exists_{i=1}^{n} \alpha(i)$	s:=false;
	for i:= 1 **to** n **do**
	if $\alpha(i)$
	then s:= true ; exit fi
	done
	$(s = \exists_{i=1}^{n} \alpha(i)$

indukcja ograniczona	i:=0; aux:=g(x);
	for i :=1 to n do
	aux := h(i-1,x,aux)
	if aux > j(n,x) then
	aux:=0
	fi
	od;
	$(aux = f(n, x))$

rekursja prosta
$f(0, x) \stackrel{df}{=} g(x)$
$f(n + 1, x) \stackrel{df}{=} h(n, x, f(n, x))$
	i:=0; aux:=g(x);
	for i :=1 to n do
	aux := h(i-1,x,aux)
	od
	$(aux = f(n, x))$

Minimum ograniczone

5.6.1 Wnioski

A) Jeden wniosek, to postulat by odwazniej używać instrukcji przypisania z notacją matematyczną. Możesz w programie umieścić napis

```
␣\assign{y␣\leftarrow␣\sum_{i=1}^n␣i*a}␣
```

i traktować go jak instrukcję przypisania. Z kolei program będzie kompilowany najpierw przez TEX i ... jeśli program (lub jego fragment) ma byc publikowany to zobaczymy notację matematyczną

$$y \leftarrow \sum_{i=1}^{n} i * a.$$

A jeśli program ma być skompilowany i wykonany to użyjemy innego stylu by uzyskać

```
var i: integer;
y:=0;
for i :=1 to n do
   y:= y+ i*a
od
```

Trzeba tylko te dwa pakiety .sty przygotować. Zauważ, nie ma potrzeby konstruowania nowego języka programowania i jego kompilatora!
Zauważ także: stosowanie takiej notacji zmniejsza ryzyko pojawienia się błędu.

B) Programiści odczuwają potrzebę instrukcji foreach, podobnej do intrukcji for. Instrukcja foreach miałaby odnosić się do zbiorów skończonych zapisanych w pamięci komputera i umożliwiać powtarzanie pewnej sekwencji poleceń dla każdego elementu z zadanego zbioru skończonego. Nasuwa się pytanie czy można wprowadzić taką instrukcję foreach do zestawu poleceń?

Na to pytanie postaramy się odpowiedzieć w części drugiej: *Programuj z klasą.*

Zadanie 5.1. *Napisz program wyznaczajacy największą z trzech liczb x, y, z.*

Zadanie 5.2. *Napisz program: weź następną czwórkę liczb naturalnych.*
Wskazówka 1. Napisz program weź następną parę liczb naturalnych.
Wskazówka 2. Czy ciąg czwórek utworzonych przez Twój program zaczyna się podobnie ... ?

Rozdział 6

Programowanie z tablicami $\mathcal{L}_4$

W tym rozdziale omówimy tworzenie obiektów tablicowych i operacje na takich obiektach. Zazwyczaj mówimy krótko tablice (*ang.* arrays). Wygodnie jest pojmować tablice jako zestawy zmiennych.

TODO
A. Składnia
B. Semantyka aksjomaty
C. Maszyna 1. opis typu tablicowego
2. iloczyn skalarny
3. macierz trójkatna
4. algorytm Gaussa
5. mnożenie macierzy
(zwykłe, alg. Winograda, mnożenie w strukturach tropikalnych, ...) 6. domknięcie tranzytywne relacji (algorytm kosztowny, binpower,)

6.1 Tablice

Tablice są obiektami, które zawierają jednorodny ciąg zmiennych.
Możemy też powiedzieć, że obiekt tablicowy jest przedłużeniem pamięci programu o pewien skończony zbiór zmiennych, wszystkie tego samego typu i wszystkie nazwy tych zmiennych są podobne.
Jeszcze inaczej, tablica jest funkcją. To są trzy różne spojrzenia, ale przedmiot oglądany, analizowany pozostaje ten sam.

Typy tablicowe i zmienne tablicowe.

Poniżej znajdziesz przykład deklaracji: zmiennej tablicowej A oraz typu tablicowego **arrayof** T, gdzie T jest typem zadeklarowanym lub pierwotnym.

var A : **arrayof** T;

Powtarzające się deklaracje typu możemy utożsamiać, dwie linie deklaracji:
`var A arrayof T;`

`var B arrayof T;`
znaczą tyle samo co następująca deklaracja
var A,B **arrayof** T

Przykład 6.1. Obliczanie iloczynu skalarnego wektorów

```
program ilSkal;
  var n,i, iloczyn: integer;
  var A, B: arrayof integer;
begin
  readln(n);
  array A dim (1:n);
  array B dim (1:n);
  for i := 1 to n do read(A(i)) od;
  readln;
  for i := 1 to n do B(i):=3*i od;
  for i := 1 to n do iloczyn :=iloczyn +A(i)*B(i) od;
  writeln("Iloczyn skalarny A * B=", iloczyn)
end
```

Jak będzie przebiegać obliczenie powyższego programu?

1) *stan początkowy*
integer: n =0 *wartościowanie zmiennych* integer: i =0 integer: iloczyn =0 arrayof integer A = none arrayof integer B = none
readln(n); **array** A **dim** (1:n); **array** B **dim** (1:n); **for** i := 1 **to** n **do** read(A(i)) **od**; readln; **for** i := 1 **to** n **do** B(i):=3*i **od**; **for** i := 1 **to** n **do** iloczyn :=iloczyn +A(i)*B(i) **od**; writeln("Iloczyn skalarny A * B=", iloczyn)

2) *wczytano n=3*
integer: n =3 integer: i =0 integer: iloczyn =0 arrayof integer A = none arrayof integer B = none
~~readln(n);~~ **array** A **dim** (1:n); **array** B **dim** (1:n); **for** i := 1 **to** n **do** read(A(i)) **od**; readln; **for** i := 1 **to** n **do** B(i):=3*i **od**; **for** i := 1 **to** n **do** iloczyn :=iloczyn +A(i)*B(i) **od**; writeln("Iloczyn skalarny A * B=", iloczyn)

```
3)  tworzymy obiekt tablicy A
integer: n =3
integer: i =0
integer: iloczyn =0
arrayof integer A =
arrayof integer B = none
readln(n);
array A dim (1:n);
array B dim (1:n);
for i := 1 to n do read(A(i)) od;
readln;
for i := 1 to n do B(i):=3*i od;
for i := 1 to n do iloczyn :=iloczyn +A(i)*B(i) od;
writeln("Iloczyn skalarny A * B=", iloczyn)
```

A=

(1)	0
(2)	0
(3)	0

```
4)  tworzymy obiekt tablicy B
integer: n =3
integer: i =0
integer: iloczyn =0
arrayof integer A =
arrayof integer B =
readln(n);
array A dim (1:n);
array B dim (1:n);
for i := 1 to n do read(A(i)) od;
readln;
for i := 1 to n do B(i):=3*i od;
for i := 1 to n do iloczyn :=iloczyn +A(i)*B(i) od;
writeln("Iloczyn skalarny=", iloczyn)
```

A=

(1)	0
(2)	0
(3)	0

B=

(1)	0
(2)	0
(3)	0

```
5)  kolejne wprowadzone liczby to 2, 4, 6
integer: n =3
integer: i =4
integer: iloczyn =0
arrayof integer A =
arrayof integer B =
readln(n);
array A dim (1:n);
array B dim (1:n);
for i := 1 to n do read(A(i)) od;
readln;
for i := 1 to n do B(i):=3*i od;
for i := 1 to n do iloczyn :=iloczyn +A(i)*B(i) od;
writeln("Iloczyn skalarny=", iloczyn)
```

A=

(1)	2
(2)	4
(3)	6

B=

(1)	3
(2)	6
(3)	9

Powiedz proszę, co wydrukuje program? Uzasadnij.
Co się stanie jeśli pierwsza instrukcja readln(n) wprowadzi wartość 0 zero?
Udowodnij, że dla n>0 program obliczy poprawną wartość iloczynu skalarnego.
Jak zmienić program by użytkownik nie napotkał niespodzianki podczas stosowania tego programu?

6.2 Składnia języka $\mathcal{L}_4$

Język $\mathcal{L}_4$ jest rozszerzeniem poprzedniego języka $\mathcal{L}_3$.

$$\mathcal{L}_3 \subsetneq \mathcal{L}_4$$

Rozszerzeniu ulegają: zbiór deklaracji, zbiór wyrażeń i zbiór instrukcji atomowych. Natomiast zachowane zostają operacje na instrukcjach: **if** i **for**.

Deklaracje

Definicja 6.2. *Deklaracją zmiennej tablicowej A jest napis postaci*

var A: {**arrayof** }$^+$ T: A ∈ Identyfikator, T ∈{boolean, integer, real, char, string }

Jest to równocześnie deklaracja zmiennej A i deklaracja typu tablicowego arrayof T.

Mamy więc

$$\mathcal{D}_4 \stackrel{df}{=} \mathcal{D}_3 \cup \{\textbf{var } A: \textbf{ arrayof } T: A \in Identyfikator, T \in \{boolean, integer, real, char, string\}\}$$

Czyli, deklaracją w języku $\mathcal{L}_4$ jest deklaracja zmiennej tablicowej lub deklaracja z języka $\mathcal{L}_3$.
Ciąg deklaracji $\mathbb{D}$ występujący w programie $\mathbb{D} \in D_4^*$
to skończony ciąg deklaracji zmiennych prostych i stałych (jak w języku $\mathcal{L}_3$) i deklaracji zmiennych tablicowych.

Wyrażenia

W zbiorze wyrażeń dopuszczalne są Wyrażenie tablicowe
zmienna tablicowa A jest wyrażeniem tablicowym,
generator array A dim (a:b)
Wyrażenie arytmetyczne
Zmienna indeksowana A(i) jest wyrażeniem typu T

wyrażenie boolowskie
A=B gdzie A i B są zmiennymi tego samego typu tablicowego
wyrażenie arytmetyczne
napisy lower(A) i upper(A) są wyrażeniami arytmetycznymi

Instrukcje

Niech A i B będą zmiennymi tablicowymi, a i b wyrażeniami arytmetycznymi.
Zbiór instrukcji atomowych języka $\mathcal{L}_4$

$$At_4 = At_3 \cup \{\textbf{array } A \textbf{ dim}(a:b), B := copy(A), kill(A)\}$$

Zbiór instrukcji $\mathcal{I}_4$ języka $\mathcal{L}_4$ jest najmniejszym zbiorem napisów zawierającym zbiór instrukcji atomowych At_4 i zamkniętym ze względu na operacje tworzenia instrukcji warunkowych (**if**) i instrukcji powtarzania (**for**)

6.3 Semantyka

Aksjomat (niezmiennik) języka Loglan

Niezmiennik systemu Loglan. *Dla każdej zmiennej tablicowej A jej wartość jest obiektem tablicowym typu* arrayof T *wymienionego w deklaracji zmiennej A lub jest równa* **none**.

$$Loglan \vdash A \textit{ in } \textbf{arrayof } T \vee A = \textbf{none}$$

Początkowa wartość zmiennej tablicowej $c_0 =$ **none**
Aksjomat utworzenia tablicy
Niech napis A będzie zadeklarowaną zmienną tablicową typu arrayof T, niech napisy δ i μ będą wyrażeniami arytmetycznymi. Symbol c_0 oznacza wartość początkową typu T.[1] Polecenie **array** A **dim**(δ:μ) ma efekt opisany formułami o schemacie podanym poniżej:

$$u \geq l \Rightarrow \{\textbf{array } A \textbf{ dim } (l:u)\}(A \neq \textbf{none} \wedge lower(A) = l \wedge upper(A) = u \wedge \overset{u}{\underset{i=l}{\forall}} A(i) = c_0) \qquad \text{(AxArr)}$$

popraw ten aksjomat uwzględnij typ T
Aksjomat przypisania
PRE $\Rightarrow$ {A(i):=wyr }(POST)
W dowodach poprawności przyda się nam następujące

Twierdzenie 6.1. *Zakładamy, że B jest tablicą n-elementową. Niech i będzie zmienną typu integer. Niech $P(i)$ oznacza ciąg instrukcji, taki że, napis $B(i)$ nie występuje w nim po prawej stronie instrukcji przypisania. Poniższa równoważność jest prawdziwa w systemie Loglan*

$$\mathfrak{Loglan} \vdash \overset{n}{\underset{i=1}{\forall}} \{P(i)\}\,(B(i) = \tau(i)) \Leftrightarrow \left\{ \begin{array}{l} \textbf{for } i \leftarrow 1 \textbf{ to } n \textbf{ do} \\ \quad P(i) \\ \textbf{od} \end{array} \right\} \overset{n}{\underset{i=1}{\forall}} (B(i) = \tau(i))$$

Dowód. Zastosujemy regułę wnioskowania 5.5 ... □

Przydatna też będzie odmiana powyższego twierdzenia

Twierdzenie 6.2. *Jeśli*

6.4 Komputer $\mathcal{K}_4$

Ten komputer jest rozszerzeniem komputera $\mathcal{K}_3$ i potrafi wykonywać wszelkie polecenia, jakie umie wykonywać ten ostatni, a ponadto komputer $\mathcal{K}_4$ potrafi: utworzyć obiekt tablicowy, odczytać wartość zmiennej indeksowanej , przypisać zmiennej indeksowanej nową wartość, usunąć obiekt tablicowy A wykonując polecenie kill(A).
Repertuar poleceń komputera $\mathcal{K}_4$ zawiera wszystkie rodzaje poleceń znane nam z opisów wcześniejszych i ponadto polecenia następujace:

- UTWÓRZ OBIEKT TABLICOWY
  $\langle Pam, \text{array A dim(l:u); Ins}\rangle \mapsto \langle Pam \cup \{o\}, \text{Ins}\rangle$
  gdzie $Pam' = Pam \cup \{o\}$ jest stanem pamięci Pam powiększonym o nowy obiekt tablicowy o ..., wartością zmiennej tablicowej A jest obiekt o

  (*i*) Oblicz wartości wyrażeń arytmetycznych l i u.

  (*ii*) Do istniejącego zbioru obiektów Pam dodaj nowy obiekt o rozmiarze $u - l + 1$ i przypisz go jako wartość zmiennej A.

[1] Przypomnijmy dla typu integer jest to 0, dla typu Boolean jest to false, dla typu tablicowego jest to none.

(*iii*) Zmiennym indeksowanym $A(l), A(l+1), A(u)$ przypisz wartości początkowe zgodne z typem T

- STWÓRZ KOPIĘ OBIEKTU TABLICOWEGO B := copy(A)

  (*i*) Niech obiekt o będzie wartością zmiennej tablicowej A.

  (*ii*) Stwórz nowy obiekt tablicowy o' będący kopią obiektu A.

  (*iii*) Przypisz obiekt o' jako wartość zmiennej tablicowej B.

  Właściwie, instrukcja B:=copy(A) jest skrótem oznaczającym następującą parę instrukcji

$$B := copy(A) \stackrel{df}{=} \left\{ \begin{array}{l} \textbf{array } B \textbf{ dim}(lower(A) : upper(A)); \\ \textbf{for } i := lower(A) \textbf{ to } upper(A) \textbf{ do} \\ \quad B(i) := A(i) \\ \textbf{od} \end{array} \right\}$$

- USUŃ OBIEKT TABLICOWY kill(A)
  Jeśli wartością zmiennej A jest **none** to nic nie rób. W przeciwnym przypadku (tzn. gdy wartością zmiennej A jest obiekt tablicowy o), przypisz wszystkim zmiennym, których wartością jest obiekt o (w tym zmiennej A) wartość **none** i usuń obiekt o tzn. zbiór jednostek dynamicznych $Pam' = Pam \setminus \{o\}$.
  WAŻNE. Czas wykonania operacji kill(A) nie zależy od liczby zmiennych wskazujących obiekt o jako swoją wartość!

- WYZNACZ WARTOŚĆ ZMIENNEJ A(i)

  Jeśli $A \neq$ **none**
  to
  { jeśli $lower(A) \leq i \leq upper(A)$
  to
  wartoscią wyrażenia $A(i)$ jest wartość i-tego elementu tablicy A
  inaczej
  podnieś sygnał błędu (`array index error`)
  }
  inaczej
  podnieś sygnał błędu (`reference to none`)
  koniec

  Sprawdź czy wartość zmiennej A jest różna od **none**. Jeśli jest to podnieś sygnał błędu – raise Reference to None. Gdy wartością zmiennej A jest obiekt tablicowy o, to sprawdź czy spełniony jest warunek $lower(A) \leq i \leq upper(A)$? Jeśli warunek ten nie jest spełniony to podnieś sygnał błedu – array index error. Jeśli warunek jest spełniony to wartością wyrażenia A(i) jest i-ty element obiektu o

- ZMIEŃ WARTOŚĆ ZMIENNEJ A(ι) A(i) := ω
  Oblicz wartość wyrażenia ι.Niech to będzie i.
  Wyznacz zmienną A(i). Niech to będzie zmienna z.

popraw oznaczenie z

Oblicz wartość wyrażenia ω. Niech to będzie w.
Jeśli typy zmiennej z i wartości w nie są zgodne to podnieś sygnał błędu.
Przypisz zmiennej z wyliczoną wartość w.

Operacje na tablicach

Jeśli mamy dwie zmienne tego samego typu tablicowego

var A, B: arrrayof integer;
array A **dim**(2:7);

to obie te zmienne mogą wskazywać na tę samą tablicę

B :=A ;

Po wykonaniu tej instrukcji spełniony jest warunek (formuła) A=B.
Natomiast wykonanie instrukcji

B := copy(A);

tworzy nową tablicę będącą kopią tablicy A i tę nową tablicę przypisuje jako wartość zmiennej B.
Podczas wykonywania instrukcji przypisania A := B wazna jest zgodność typów zmiennych A i B. Nie jest ważny rozmiar tablic A i B lecz czy typy ich elemntów sa zgodne.
Przed wykonaniem instrukcji B := A rozmiar tablicy B może być inny niz rozmiar tablicy A.
Obiekt tablicowy o może być wartościa wielu zmiennych, pod warunkiem, że zmienne te maja zadeklarowane zgodne typy.

Narysuj diagramy ilustrujące różnicę pomiędzy tymi instrukcjami!

Obiekt tablicowy o jest ciągiem zmiennych. Wyrażenie A(i) ma wartość wyznaczoną w ten sposób, że obliczamy wartość wyrażenia i w nawiasach i z kolei wyznaczamy wartość zmiennej A(val(i)).

lower i upper

W trakcie może wystąpic błąd “array index error”. Co to znaczy? Jak unikac takiego błędu?

Błąd “reference to none”. W jaki sposób może powstac taki błąd?

Przykład 6.3. przykład programu W tym programie jest błąd. Jaki? Napraw to.

```
program ilSkal;
  (* obliczanie iloczynu skalarnego wektorów *)
  var n,i, iloczyn: integer;
  var A, B: arrayof integer;
begin
  readln(n);
  array A dim (1:n);
  for i := 1 to n do read(A(i)) od;
  readln;
  for i := 1 to n do read(B(i)) od;
  for i := 1 to n do iloczyn :=iloczyn +A(i)*B(i) od;
  writeln(“Iloczyn skalarny A * B=”, iloczyn)
end
```

Przykład 6.4. przykład programu W tym programie jest błąd. Jaki? Napraw to.

```
program ilSkal;
   (* obliczanie iloczynu skalarnego wektorów *)
   var n,i, iloczyn: integer;
   var A, B: arrayof integer;
begin
   readln(n);
   array A dim (1:n);
   array B dim (1:n);
   for i := 1 to n do read(A(i)) od;
   readln;
   for i := 0 to n do read(B(i)) od;
   for i := 1 to n+3 do iloczyn :=iloczyn +A(i)*B(i) od;
   writeln("Iloczyn skalarny A * B=", iloczyn)
end
```

Przykład 6.5. przykład programu W tym programie jest błąd. Jaki? Napraw to.

```
program ilSkal;
   (* obliczanie iloczynu skalarnego wektorów *)
   var n,i, iloczyn: integer;
   var A, B: arrayof integer;
begin
   readln(n);
   array A dim (1:n);
   array B dim (n:n-2);
   for i := 1 to n do read(A(i)) od;
   readln;
   for i := 1 to n do read(B(i)) od;
   for i := 1 to n do iloczyn :=iloczyn +A(i)*B(i) od;
   writeln("Iloczyn skalarny A * B=", iloczyn)
end
```

6.5 Tablice dwuwskaźnikowe.

Tablica dwuwskaźnikowa może (i w Loglanie musi) być tablicą tablic.

```
var A, B, C: arrayof arrayof real;
...
array A dim(1:n);
for i:= 1 to n do
array A(i) dim(1:n)
od;
```

Może to jest uciążliwe, ale stwarza możliwość tworzenia tablic o różnych kształtach, np. tablice trójkątne, tablice wstęgowe, etc.

Przykład 6.6. Zapisać macierz trójkątną górną

$$\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ & a_{22} & \dots & a_{2n} \\ & & \ddots & \\ & & & a_{nn} \end{bmatrix}$$

Zadeklaruj tablicę dwuwskaznikową
var A **arrayof arrayof** real;
Utwórz tablicę
array A **dim**(1:n);
for i:=1 **to** n **do array** A(i) **dim(i:n)**;
Utworzymy w ten sposób tablicę o n wierszach A(1) , ... , A(n). Dla i=1, ..., n. i-ty wiersz ma elementy o numerach od i do n, A(i,i), ... A(i,n). Pozostaje wypełnić tę tablicę. Ile miejsca ona zajmie?

Ćwicz!

Zadanie 6.1. *Utwórz dwie tablice A i B o wymiarach 4×4. Wypełnij je liczbami przypadkowymi i oblicz sumę C=A+B tych tablic.*

Zadanie 6.2. *Utwórz dwie tablice A i B o wymiarach 4×4. Wypełnij tablicę B liczbami przypadkowymi, zapisz w tablicy A kopię tablicy B i oblicz sumę C=A+B tych tablic.*

Zadanie 6.3. *Utwórz dwie tablice A i B o wymiarach 4×4. Wypełnij je liczbami przypadkowymi, wykonaj przypisanie B:=A i następnie wypełnij tablicę A zerami. Wydrukuj obie tablice. Co zobaczysz?*

kill()

Co zrobić gdy tablica nie jest juz potrzebna? Pozbyć się jej wykonując polecenie kill(A); Semantyka polecenia kill jest opisana przez następujący aksjomat

Niezmiennik systemu Loglan. *Jeśli zmienne A, B, C wskazują na pewien obiekt tablicowy, to po wykonaniu polecenia **kill**(B) wszystkie trzy zmienne przyjmują wartość none, a obiekt zostaje usunięty.*

$$\mathrm{L}_O\mathrm{GL}^A\mathrm{N} \vdash (A = B = C \neq \textbf{none}) \Rightarrow \{\mathrm{kill}(A)\}(A = B = C = \textbf{none})$$

Oczywiście, to samo można powiedzieć gdy na pewien obiekt wskazują dwie, pięć, lub więcej zmiennych.

Zadanie 6.4. *Na pewien obiekt tablicowy o wskazują dwie zmienne A i B. Czy w tej sytuacji instrukcje są równoważne?*
kill(A) | *A := none* | *A,B :=none*

Aksjomat instrukcji przypisania

Ten aksjomat wymaga ponownego sformułowania. Formuła $\boxed{\{x := \tau\}\alpha(x) \Leftrightarrow \alpha(x/\tau)}$ nie opisuje całej prawdy o stanie pamięci po wykonaniu instrukcji przypisania np. $A(3) := 7$. Jeśli bowiem przed wykonaniem tej instrukcji na tablicę A wskazywała także zmienna B to po wykonaniu tej instrukcji zachodzi warunek

aksj.? zastosuj dwukrotnie aksj. przypisania

przemyśl tablice dwuwskaźnikowe

$A(3) = 7 \wedge B(3) = 7$. W związku z tym zalecane jest stosowanie następującej reguły: Jeśli zmienna x jest elementem tablicy A i formuła $\alpha(x/\tau)$ zawiera także klauzule wyliczające wszystkie aliasy obiektu o wskazywanego przez zmienną tablicową A, to formuła $\{x := \tau\}\alpha(x) \Leftrightarrow \alpha(x/\tau)$ jest prawdziwa. Niech A i B będą dwoma zmiennymi typu **arrayof arrayof** T. Jeśli spełniona jest formuła $A = B$ to znaczy, że $\forall_{i=lower(A)}^{upper(A)} A(i) = B(i)$. Z kolei to, że zachodzi równość $A(i) = B(i)$ oznacza, że dla każdego $lower(A(i)) \leq j \leq upper(A(i))$ spełniona jest równość $A(i,j) = B(i,j)$. Ale może się zdarzyć, że w takim stanie zostanie wykonane polecenie zmieniające pewien wiersz tablicy B, np. $B(4) \leftarrow C$, gdzie wartością zmiennej C jest obiekt typu **arrayof** $\top$. W wyniku nadal będzie prawdą, że $A = B$, ale zmieniona została też wartość $A(4)$ i jest równa C. Podobnie będzie z instrukcją $A(3,7) \leftarrow \tau$. Nadal zachodzi równość $A = B$.
Natomiast po wykonaniu instrukcji $A \leftarrow D$ lub $array\ A\ dim(2:9)$ równość $A = B$ przestaje zachodzić.

Przykład 6.7. $(A = B \wedge 7 < 9) \Leftrightarrow \{B(3) := 7\}(A = B \wedge A(3) < 9 \wedge B(3) < 9)$

6.6 Rozwiązywanie układu równań

Umiemy operowac na tablicach dwuwskaźnikowych. Zobaczmy do czego takie obiekty się przydają.

Należy rozwiązać układ równań liniowych dany przez trójkątną, nieosobliwą macierz górną współczynników A i wektor wyrazów wolnych B.

$$\begin{aligned} A_{11}x_1 \quad &+A_{12}x_2 \quad \cdots + A_{1n}x_n = B_1 \\ &+A_{22}x_2 \quad \cdots + A_{2n}x_n = B_2 \\ &\cdots \\ & \qquad\qquad\qquad A_{nn}x_n = B_n \end{aligned}$$

Szkic programu

```
program G;
    var A: arrayof arrayof real, B, X: arrayof real;
    var sum: real, n, k, l: integer;
begin
    (* utwórz i zainicjuj tablice A i B  *)
    readln(n);
    array A dim(1:n);
    for k := 1 to n do array A(k) dim(1:n);
    array B dim (1:n);
    (* wczytaj tablice A i B *)
    ...
    array X dim (1 :n);
    (* oblicz tablice X *)
    for k: =n downto 1 do
        suma :=0;
        for j: = k+ 1 to n do suma := suma+A(k,j)*X(j) od;
        X(k): =( B(k)-suma)/ A(k,k)
```

```
    od
    (* wydrukuj tablice X *)
end
```

Udowodnimy następujący

Lemat 6.3. *Jeżeli A jest trójkątną, nieosobliwą macierzą górną i tablica B jest wektorem wyrazów wolnych to prawdziwa jest następująca formuła*

$$G: \left\{ \begin{array}{l} \text{for k: =n downto 1 do} \\ \quad \text{suma :=0;} \\ \quad \text{for j: = k+ 1 to n do} \\ \quad\quad \text{suma := suma+A(k,j)*X(j)} \\ \quad\quad \text{od;} \\ \quad \text{X(k): =(B(k)-suma)/ A(k,k)} \\ \text{od} \end{array} \right\} \left(\bigwedge_{i=1}^{n} \sum_{j=i}^{n} A(i,j) * X(j) = B(i) \right)$$

To znaczy, że instrukcja G oblicza rozwiązanie układu równań podanego powyżej.

Dowód. Najpierw zauważmy, że na mocy twierdzenia 5.3, dla $k = n, n-1, \ldots, 1$ zachodzą formuły

$$\left\{ \begin{array}{l} \text{suma :=0;} \\ \text{for j: = k+ 1 to n do} \\ \quad \text{suma := suma+A(k,j)*X(j)} \\ \text{od;} \end{array} \right\} \left(\sum_{j=k+1}^{n} A(k,j) * X(j) = suma \right)$$

Stąd wnioskujemy, że dla $k = n, n-1, \ldots, 1$ zachodzą formuły

$$\left\{ \begin{array}{l} \text{suma :=0;} \\ \text{for j: = k+ 1 to n do} \\ \quad \text{suma := suma+A(k,j)*X(j)} \\ \text{od;} \\ \text{X(k): =(B(k)-suma)/ A(k,k)} \end{array} \right\} \left(X(k) = \frac{B(k) - \sum_{j=k+1}^{n} A(k,j) * X(j)}{A(k,k)} \right)$$

Każda z tych formuł jest równoważna odpowiednio następującej

$$\left\{ \begin{array}{l} \text{suma :=0;} \\ \text{for j: = k+ 1 to n do} \\ \quad \text{suma := suma+A(k,j)*X(j)} \\ \text{od;} \\ \text{X(k): =(B(k)-suma)/ A(k,k)} \end{array} \right\} \left(\sum_{j=k}^{n} A(k,j) * X(j)) = B(k) \right)$$

dla $k = n, n-1, \ldots, 1$.

unikaj opowiadania o obliczeniach

I to właściwie kończy dowód lematu. Warto jeszcze zauważyć kolejność w jakiej obliczamy wartości niewiadomych $x_n, x_{n-1}, \ldots, 1$. Raz obliczona wartość niewiadomej x_i nie ulega zmianie w kolejnych krokach algorytmu.

Wykorzystujemy odmianę twierdzenia 6.1 by wprowadzić polecenie **for**.

$$G: \left\{ \begin{array}{l} \text{for k: =n downto 1 do} \\ \quad \text{suma :=0;} \\ \quad \text{for j: = k+ 1 to n do} \\ \quad\quad \text{suma := suma+A(k,j)*X(j)} \\ \quad\quad \text{od;} \\ \quad \text{X(k): =(B(k)-suma)/ A(k,k)} \\ \text{od} \end{array} \right\} \left(\underset{i=1}{\overset{n}{\forall}} \sum_{j=i}^{n} A(i,j) * X(j) = B(i) \right)$$

□

6.7 Mnożenie macierzy

Dane są dwie tablice A i B. Należy sprawdzić czy można je pomnożyć przez siebie i jeśli to jest możliwe należy obliczyć macierz $C = A * B$. Przyjmując, że macierz A ma n wierszy i k kolumn, a macierz B ma k wierszy i m kolumn należy napisać program P i zapewnić prawdziwość następującej formuły

$$\{P\} \underset{i=1}{\overset{n}{\forall}} \underset{j=1}{\overset{m}{\forall}} \left(C_{ij} = \sum_{l=1}^{k} A_{il} * B_{lj} \right)$$

Operacja mnożenia macierzy ma tak wiele zastosowań, że warto pokusić się o algorytm o możliwie niskim koszcie. Zobacz prace [].

6.7.1 Algorytm podstawowy

Wykorzystamy spostrzeżenia poczynione w poprzednim rozdziale. Następująca formuła jest twierdzeniem Loglanu.

Twierdzenie 6.4.

$$\mathcal{AL} \vdash \left\{ \begin{array}{l} \textbf{for } i \leftarrow 1 \textbf{ to } n \textbf{ do} \\ \quad \textbf{for } j \leftarrow 1 \textbf{ to } m \textbf{ do} \\ \quad\quad s \leftarrow 0; \\ \quad\quad \textbf{for } l \leftarrow 1 \textbf{ to } k \textbf{ do} \\ \quad\quad\quad s \leftarrow s + A(i,l) * B(l,j) \\ \quad\quad \textbf{od}; \\ \quad\quad C(i,j) \leftarrow s; \\ \quad \textbf{od} \\ \textbf{od} \end{array} \right\} \left(\underset{i=1}{\overset{n}{\forall}} \underset{j=1}{\overset{m}{\forall}} C_{ij} = \sum_{l=1}^{k} A_{il} * B_{lj} \right)$$

Dowód. Z twierdzenia 5.3 mamy

$$\mathcal{LP} \vdash \underset{i=1}{\overset{n}{\forall}} \underset{j=1}{\overset{m}{\forall}} \left(\left\{ \begin{array}{l} s := 0; \\ \textbf{for } l := 1 \textbf{ to } k \textbf{ do} \\ \quad s := s + A(i,l) * B(l,j) \\ \textbf{od}; \\ C(i,j) := s \end{array} \right\} \left(C_{ij} = \sum_{l=1}^{k} A_{il} * B_{lj} \right) \right)$$

Teraz zastosujemy spostrzeżenie, że kwantyfikator ograniczony można wyrazić przez pętlę for 6.1.

$$\mathcal{L}\mathrm{P} \vdash \bigvee_{i=1}^{n} \left(\left\{ \begin{array}{l} \textbf{for}\ \mathrm{j}{:=}1\ \textbf{to}\ \mathrm{m}\ \textbf{do} \\ \quad \mathrm{s}{:=}0; \\ \quad \textbf{for}\ \mathrm{l}{:=}1\ \textbf{to}\ \mathrm{k}\ \textbf{do} \\ \quad\quad \mathrm{s}{:=}\mathrm{s}{+}\mathrm{A(i,l)}{*}\mathrm{B(l,j)} \\ \quad \textbf{od}; \\ \quad \mathrm{C(i,j)}{:=}\mathrm{s} \\ \textbf{od} \end{array} \right\} \left(\bigvee_{j=1}^{m} C_{ij} = \sum_{l=1}^{k} A_{il} * B_{lj} \right) \right)$$

Jeszcze raz stosujemy ten sam fakt

$$\mathrm{L}_{\mathrm{an}}^{\mathrm{og}} \vdash \left\{ \begin{array}{l} \textbf{for}\ \mathrm{i}{:=}1\ \textbf{to}\ \mathrm{n}\ \textbf{do} \\ \quad \textbf{for}\ \mathrm{j}{:=}1\ \textbf{to}\ \mathrm{m}\ \textbf{do} \\ \quad\quad \mathrm{s}{:=}0; \\ \quad\quad \textbf{for}\ \mathrm{l}{:=}1\ \textbf{to}\ \mathrm{k}\ \textbf{do} \\ \quad\quad\quad \mathrm{s}{:=}\mathrm{s}{+}\mathrm{A(i,l)}{*}\mathrm{B(l,j)} \\ \quad\quad \textbf{od}; \\ \quad\quad \mathrm{C(i,j)}{:=}\mathrm{s} \\ \quad \textbf{od} \\ \textbf{od} \end{array} \right\} \left(\bigvee_{i=1}^{n} \bigvee_{j=1}^{m} C_{ij} = \sum_{l=1}^{k} A_{il} * B_{lj} \right)$$

Co kończy dowód. □

Powinniśmy zawczasu sprawdzić czy w trakcie wykonywania tego programu nie wystąpi błąd array index error. Czyli należy sprawdzić czy każdy wiersz tablicy A ma k elementów i czy każda kolumna macierzy B ma k elementów. Czy potrafisz napisać odpowiedni program? Czy potrafisz go uzasadnić?

6.7.2 Algorytm Winograda

Algorytm Winograda może być stosowany do obliczania iloczynu macierzy kwadratowych. Jego przydatność jest szczególnie widoczna gdy elementami macierzy są obiekty reprezentujące jakiś pierścień inny niż pierścień liczb rzeczywistych. Np. w przypadku gdy dany pierścień $\mathcal{C}$ jest zaimplementowany w programie przez klasę C, zob. następna część II.

Drugim i ważniejszym powodem do skreślenia tej notatki jest potrzeba zwrócenia uwagi na znikomą przydatność asercji i tzw. programowania przez kontrakt (*ang.* design by contract). Asercje są przydatne jako notatki, ale nie stanowią rozwiązania problemu zapewnienia poprawności algorytmu.

Zapraszam do czytania, zwłaszcza rozdziału Dowód poprawności.

Algorytm

W tym rozdziale pojawiają się dwa warunki:

- warunek wstępny – Precondition,
- warunek końcowy – Postcondition, oraz
- algorytm – algorytm Winograda.

Jak się upewnić, że algorytm jest poprawny ze względu na warunek początkowy i warunek końcowy? W literaturze proponowane są dwa podejścia:

- udowodnij częściową porawność sprawdzając pewne niezmienniki – jest to tzw. metoda Floyda-Hoare'a,
- wykonaj pewną liczbę obliczeń testowych i zaufaj, że w trakcie eksploatacji algorytmu nie okaże się, że jest on niepoprawny.

W obliczeniach testowych można w trakcie obliczeń sprawdzać wcześniej wstawione warunki – asercje. Czy rzeczywiście są one pomocne?

W następnym punkcie pokażemy inną drogę - drogę dowodzenia lematów i w końcu twierdzenia o poprawności (całkowitej) algorytmu Winograda względem warunku początkowego **Precondition** i warunku końcowego **Postcondition**.

Przerób na program

1: **signal** *Niezgoda*;
2: **unit** *Winograd* : **procedure**(A, B : **array_of array_of** *real*; **output** C : **array_of array_of** *real*);
Precondition: wymagaj by A i B były macierzami kwadratowymi rozmiaru n x n
Postcondition: zapewnij, że obliczona macierz C jest produktem macierzy A i B, C = A * B
3: **var** i, j, k, n, m : **integer**, W, V : **array_of real**, p : **boolean**, s : **real**;
4: **begin**
{ *ustalić czy macierze mogą być mnożone tzn. czy ilość wierszy w A = ilosc kolumn w B?* }
{ *ustalić czy n jest parzyste?* }
{ *obliczyc preprocessing* }

{ dynamiczne sprawdzanie precondition }
5: **if** lower(A) ≠ lower(B) **or** lower(A)≠ 1 **or** upper(A)≠ upper(B) **then**
6: *raise Niezgoda*
7: **fi**;
8: i := upper(A);
9: j := lower(A);
10: n := i-j+1;
11: **for** $l := j$ **to** i **do**
12: **if** lower(A(l)) ≠ lower(B(l)) **or** lower(A(l))≠ 1 **or** upper(A(l))≠ upper(B(l) **or** upper(A(l)≠upper(A) **then**
13: raise Niezgoda
14: **fi**;
15: **od**;
Assertion sprawdzono: macierze są kwadratowe, rozmiaru n x n

{ można mnożyć }
16: $p := (n \ mod \ 2) = 0$;
17: $m := n \ div \ 2$;
18: **array** W **dim**$(1 : n)$;
19: **array** V **dim**$(1 : n)$;
20: **array** C **dim**$(1 : n)$;

21: **for** i := 1 to n **do**
22: **array** $C(i)$ **dim**$(1:n)$
23: **od**;

{ obliczanie "preprocessingu"}
24: **for** j:= 1 to n **do**
25: s:=0;
26: **for** i := 1 to m **do**
27: $s := A[j, 2*i-1] * A[j, 2*i] + s$;
28: **od**;
29: W[j] := s;
30: **od**;

Assertion 1: Dla każdego $j, 1 \leq j \leq n$, $\quad W_j = \sum_{i=1}^{n \div 2} A_{j,2i-1} * A_{j,2i}$

31: **for** j:= 1 to n **do**
32: s:=0;
33: **for** i := 1 to m **do**
34: s := B[2*i-1,j] * B[2*i,j] +s;
35: **od**;
36: V[j] := s;
37: **od**;

Assertion 2: Dla każdego $j, 1 \leq j \leq n$, $\quad V_j = \sum_{i=1}^{n \div 2} B_{2i-1,j} * B_{2*i,j}$

{obliczanie iloczynu macierzy }
38: **for** $i := 1$ to n **do**
39: **for** $j := 1$ to n **do**
40: s:= 0;
41: **for** $k := 1$ to m **do**
42: s:= (A[i,2*k-1]+B[2*k,j]) * (B[2*k-1,j]+A[i,2*k]) +s;
43: **od**;

Assertion 3: $\forall_{1\leq i\leq n}, \forall_{1\leq j\leq n}, \quad s = \sum_{k=1}^{n \div 2} (A_{i,2k-1} + B_{2k,j}) * (B_{2*k-1,j} + A_{i,2k})$

44: $C[i, j] := s - W[i] - V[j]$;

Assertion 4: $\forall_{1\leq i\leq n}, \forall_{1\leq j\leq n} \quad C_{i,j} = \sum_{k=1}^{2(n \div 2)} (A_{i,k} * B_{k,j})$

45: **if not** p **then**
46: $C[i, j] := C[i, j]+A[i, n]*B[n, j]$; { poprawiamy - gdy n jest nieparzyste
}
47: **fi**;
48: **od**; { j }
49: **od**; { i }

Assertion 5: Dla każdych wartości $i, j, 1 \leq i \leq n, 1 \leq j \leq n$, $\quad C_{i,j} = \sum_{k=1}^{n} (A_{i,k} * B_{k,j})$

50: **end** *Winograd*;

Dowód poprawności

Naszym zadaniem jest wykazać nastepującą implikację

$$\text{Precondition} \Rightarrow \{\text{Algorytm Winograda}\}\text{Postcondition}$$

co się czyta tak: *jeśli dane spełniają warunek wstępny to algorytm Winograda kończy obliczenia nie sygnalizując błędu i wyniki spełniają warunek końcowy.*

Sprawdzenie czy warunek wstępny jest spełniony przez dane może być w części wykonane przez kompilator. Kompilator może sprawdzić czy parametry aktualne sa tablicami dwuwymiarowymi. Druga część warunku, że rozmiary tablic są równe $n \times n$ nie może być sprawdzona przed wywołaniem procedury Winograd, nie może też być udowodniona. Wobec tego wykonywanie procedury rozpoczynamy od (dynamicznego) sprawdzania kształtu i rozmiaru tablic[2]. Można rozważać czy nie dałoby się udowodnić, o programie stosującym procedurę Winograd, że warunek wstępny jest spełniony za każdym razem gdy procedura Winograd jest wywoływana w naszym programie. Ale czy można zagwarantować, że każdy program stosujący algorytm Winograda będzie sprawdzał warunek wstępny? lub go dowodził? Lepiej więc zostawić sprawdzanie warunku wstępnego procedurze Winograd.

Do algorytmu Winograd wstawiliśmy asercje. Mają one za zadanie:

1. umożliwić sygnalizację naruszenia warunku asercji w trakcie wykonywania programu,
2. ułatwić argumentację na rzecz tezy o poprawności algorytmu.

W tym przypadku trudno mówić o dynamiczej weryfikacji: ażeby sprawdzić warunek wyliczony w asercji trzeba powtórzyć obliczenia – to niewiele nam daje. Ponadto, tu uwaga natury ogólnej, zamiana asercji na instrukcję warunkową

if *warunek_asercji* **then** wrzuć_ wyjątek **fi**

zapewnia tylko tyle, że podczas wykonywania algorytmu zostanie zasygnalizowany błąd. Nie mamy nawet gwarancji, że zdarzy się to zawsze gdy program zawiera błędy.

Natomiast asercje możemy zastąpić lematami i udowodnić je

Lemat 6.5. *Dla każdego* $j, 1 \leq j \leq n$, *i* $m = n \div 2$ *zachodzi*

$$K: \left\{ \begin{array}{l} s := 0; \\ \textbf{for } i := 1 \textbf{ to } m \\ \textbf{do} \\ \quad s := A[j, 2*i-1] * A[j, 2*i] + s; \\ \textbf{od}; \\ W[j] := s; \end{array} \right\} \left(W_j = \sum_{i=1}^{n \div 2} A_{j,2i-1} * A_{j,2i} \right)$$

lub to samo spostrzeżenie zapisane nieco inaczej

[2]W loglanie'82 dwuwymiarowej tablicy możemy nadać kształt trójkatny, wstęgowy i oczywiście kształt kwadratowy

Lemat 6.6. *Dla $m = n \div 2$ zachodzi*

$$M: \left\{ \begin{array}{l} \textbf{\textit{for}}\ j := 1\ \textbf{\textit{to}}\ n\ \textbf{\textit{do}} \\ \quad s := 0; \\ \quad \textbf{for}\ i := 1\ \textbf{to}\ m \\ \quad \textbf{do} \\ \quad\quad s := A[j, 2*i-1] * A[j, 2*i] + s; \\ \quad \textbf{od}; \\ \quad W[j] := s; \\ \textbf{\textit{od}} \end{array} \right\} \left(\forall_{1 \le j \le n}\, W_j = \sum_{i=1}^{n \div 2} A_{j,2i-1} * A_{j,2i} \right)$$

Zwróć uwagę na to, że zewnętrzna instrukcja **for** oblicza kwantyfikator ograniczony, a wewnętrzna instrukcja **for** oblicza sumę. Instrukcja **for** ma jeszcze wiele innych zastosowań.

Lemat 6.7. *Dla każdego $j, 1 \le j \le n$, i $m = n \div 2$ zachodzi*

$$\left\{ \begin{array}{l} s := 0; \\ \textbf{for}\ i := 1\ \textbf{to}\ m \\ \textbf{do} \\ \quad s := B[2*i-1, j] * B[2*i, j] + s; \\ \textbf{od}; \\ V[j] := s; \end{array} \right\} \left(V_j = \sum_{i=1}^{n \div 2} B_{2i-1,j} * B_{2i,j} \right)$$

Kolejny lemat

Lemat 6.8. $\forall_{1 \le i \le n}\ \forall_{1 \le j \le n}$

$$\left\{ \begin{array}{l} s := 0; \\ \textbf{for}\ k{:=}\ 1\ \textbf{to}\ m \\ \textbf{do} \\ \quad s := (A[i, 2*k-1] + B[2*k, j]) \\ \quad\quad *(B[2*k-1, j] + A[i, 2*k]) + s; \\ \textbf{od}; \end{array} \right\} \left(s = \sum_{k=1}^{n \div 2} (A_{i,2k-1} + B_{2k,j}) * (B_{2k-1,j} + A_{i,2k}) \right)$$

Lemat 6.9.
Przy założeniu, że tablice A i B są macierzami kwadratowymi rozmiaru n x n i że zachodzą lematy 1 oraz 2, prawdziwa jest nastepująca formuła algorytmiczna

$$\underset{1 \le i \le n}{\forall}, \underset{1 \le j \le n}{\forall} \left\{ \begin{array}{l} s := 0; \\ \textbf{for}\ k := 1\ \textbf{to}\ m \\ \textbf{do} \\ \quad s := (A[i, 2*k-1] + B[2*k, j]) \\ \quad\quad *(B[2*k-1, j] + A[i, 2*k]) + s; \\ \textbf{od}; \\ C[i, j] := s - W[i] - V[j]; \end{array} \right\} \left(s = \sum_{k=1}^{2*(n \div 2)} A_{i,k} * B_{k,j} \right)$$

Lemat 6.10.
Z prawdziwości lematów 1 i 2 wynika, że następująca formuła jest prawdziwa

$$\left\{\begin{array}{l}
\textbf{for } i := 1 \textit{ to } n \\
\textbf{do} \\
\quad \textbf{for } j := 1 \textit{ to } n \\
\quad \textbf{do} \\
\qquad s := 0; \\
\qquad \textbf{for } k{:=}\ 1 \textbf{ to } m \\
\qquad \textbf{do} \\
\qquad\quad s := (A[i, 2*k-1] + B[2*k, j]) \\
\qquad\qquad *(B[2*k-1, j] + A[i, 2*k]) + s; \\
\qquad \textbf{od}; \\
\qquad C[i, j] := s - W[i] - V[j]; \\
\qquad \textbf{if not } p \\
\qquad \textbf{then} \\
\qquad\quad C[i, j] := C[i, j] + A[i, n] * B[n, j] \\
\qquad \textbf{fi}; \\
\quad \textbf{od} \\
\textbf{od}
\end{array}\right\} \left(\bigvee_{1 \le i \le n} \bigvee_{1 \le j \le n} C_{i,j} = \sum_{k=1}^{n} A_{i,k} * B_{k,j} \right)$$

Dowody lematów

Dowód lematu 6.5

Dowód. W dowodzie wykorzystujemy następującą własność programów **for**: niech napis $\omega(i)$ oznacza wyrażenie arytmetyczne (zmienna i może, ale nie musi, w nim wystepować)

$$\left\{\begin{array}{l}
s := 0 \\
\textbf{for } i := 1 \textbf{ to } n \\
\textbf{do} \\
\quad s := \omega(i) + s \\
\textbf{od}
\end{array}\right\} \left(s = \sum_{i=0}^{n} \omega(i) \right)$$

Pozostaje skorzystać z aksjomatu instrukcji przypisania

$$\left(s = \sum_{i=0}^{n} \omega(i) \right) \Rightarrow \{W[j] := s\} \left(W(j) = \sum_{i=0}^{n} \omega(i) \right)$$

□

Dowody lematów 2 i 3 przebiegają podobnie.

Dowód lematu 6.9

Dowód. Należy udowodnić

$$(s = \sum_{k=1}^{n \div 2} (A_{i,2k-1} + B_{2k,j}) * (B_{2*k-1,j} + A_{i,2k}) \Rightarrow (s - W[i] - V[j] = \sum_{k=1}^{2(n \div 2)} A_{i,k} * B_{k,j})$$

Rozwińmy mnożenie i zastosujmy rozdzielność mnożenia wzgledem dodawania

$$\begin{aligned}
&\sum_{k=1}^{n\div 2}(A_{i,2k-1}+B_{2k,j})*(B_{2*k-1,j}+A_{i,2k})\\
&\quad=\sum_{k=1}^{n\div 2}[A_{i,2k-1}*B_{2*k-1,j}+A_{i,2k-1}*A_{i,2k}+B_{2k,j}*B_{2k-1,j}+B_{2k,j}*A_{i,2k}]\\
&\quad=\sum_{k=1}^{n\div 2}A_{i,2k-1}*B_{2*k-1,j}+\underbrace{\sum_{k=1}^{n\div 2}A_{i,2k-1}*A_{i,2k}}_{=W[i]}+\underbrace{\sum_{k=1}^{n\div 2}B_{2k,j}*B_{2k-1,j}}_{=V[j]}+\sum_{=1}^{n\div 2}B_{2k,j}*A_{i,2k}
\end{aligned}$$

Skorzystamy z lematów 6.6 oraz 6.7

$$\sum_{k=1}^{n\div 2}(A_{i,2k-1}+B_{2k,j})*(B_{2*k-1,j}+A_{i,2k})-W[i]-V[j]=\sum_{k=1}^{n\div 2}A_{i,2k-1}*B_{2*k-1,j}+\sum_{k=1}^{n\div 2}B_{2k,j}*A_{i,2k}$$

Wykorzystujemy przemienność mnożenia i łączność dodawania

$$\sum_{k=1}^{n\div 2}A_{i,2k-1}*B_{2*k-1,j}+\sum_{k=1}^{n\div 2}B_{2k,j}*A_{i,2k}=\sum_{k=1}^{2(n\div 2)}A_{i,k}*B_{k,j}$$

□

Dowód lematu 6.10

Dowód. Trzeba wykazać, że

$$(s=\sum_{k=1}^{2(n\div 2)}A_{i,k}*B_{k,j})\Rightarrow\left\{\begin{array}{l}\textbf{if not } p\\ \textbf{then}\\ \quad C[i,j]:=C[i,j]+A[i,n]*B[n,j]\\ \textbf{fi};\end{array}\right\}(s=\sum_{k=1}^{n}A_{i,k}*B_{k,j})$$

Pamiętamy, że p= (n mod 2 =0). Jeżeli n jest liczbą parzystą to $2(n\div 2)=n$ i

$$C_{i,j}=\sum_{k=1}^{n}A_{i,k}*B_{k,j}$$

W przeciwnym przypadku (tzn. gdy **not** p) sumowanie zakonczyło się dla $k=n-1$. Trzeba więc dodać wartość iloczynu $A[i,n]*B[n,j]$. □

6.8 Obiekty tablicowe – Podsumowanie

Tablice są tworzone, współdzielone, odczytywane, modyfikowane, usuwane wreszcie.

1. Deklaracja zmiennej

var A: array of T

jest równoczesnie deklaracją typu tablicowego. Możesz łączyć deklaracje:

```
var A: array of T; var B: array of T;
```

i zastąpić je przez jedną deklarację

```
var B,A: array of T;
```

Oznacza to, że typy bywają *zgodne*.

2. Opisz relację zgodności typów
3. Instrukcja

```
array A dim(low: up);
```

tworzy obiekt tablicowy. Oto co wiemy po wykonaniu takiej instrukcji:

   - A $\neq$ none,
   - lower(A) = lower
   - upper(A) = up
   - $\forall(lower(A) \leq i \leq upper(A)) \Rightarrow A(i) = initval$ init val dla typu integer i real jest 0, dla typu boolean jest false, dla typu char jest ..., dla typu tablicowego lub zadeklarowanego jako klasa jest none,

4. zmienne A i B mogą się dzielić obiektem tablicowym gdy wykonano instrukcję
   B:=A;
   w efekcie zachdzi relacja A=B i konskwentnie...
5. instrukcja kopii
   B:=copy(A);
   ma następujący efekt ...
6. instrukcja
   kill(A)
   usuwa obiekt tablicowy
   niezmiennik:
7. operacje na elementach tablicy ...

Rozdział 7

Programowanie z while $\mathcal{L}_5$

7.1 składnia i semantyka

Rozszerzenie języka wydaje się niepozorne. Ale pojawiają się nowe zjawiska obliczeniowe.
TODO
Rozwinąć to. Do tej pory nie mieliśmy problemu z zapewnieniem własności stopu programu. Każdy program, omawiany we wcześniejszych rozdziałach, ma obliczenie skończone. Wprowadzenie do języka programowania instrukcji iteracji while, z jednej strony w istotny sposób zwiększa zbiór funkcji obliczalnych, z drugiej strony tracimy gwarancję, że obliczenie programu będzie skończone. Pojawia się konieczność udowodnienia, że obliczenia programu będą skończone. A to nie zawsze jest łatwe i czasami stanowi wyzwanie dla pokoleń badaczy. W latach 30 dwudziestego wieku Ackermann zdefiniował, funkcję, która rósnie bardzo szybko i wykazał, że nie jest ona funkcją pierwotnie rekurencyjną. W przetłunaczeniu na dzisiejszy język oznacza to tyle, że funkcji Ackermana nie można zaprogramować ograniczając się do instrukcji for.

składnia

Zbiór programów iteracyjnych różni się tym od poprzednio opisanego zbioru instrukcji z tablicami, że w ciągu instrukcji takiego programu mogą pojawiać się instrukcje while.
Zbiór instrukcji programów iteracyjnych jest to najmniejszy zbiór P napisów taki, że

i) każda instrukcja przypisania należy do zbioru P,

ii) złożenie, for oraz rozgałęzienie

iii) jeśli napis γ jest formułą boolowską i napis K jest instrukcją, to napis while γ do K od jest instrukcją iteracji

TODO Poprawić tę definicję.
Przykłady

Semantyka

Znaczenie instrukcji while opisuje aksjomat i reguła wnioskowania.
Aksjomat while

$$\textbf{while } \gamma \textbf{ do } K \textbf{ od}\, \alpha \Leftrightarrow ((\neg\gamma \wedge \alpha) \vee (\gamma \wedge \{K; \textbf{while } \gamma \textbf{ do } K \textbf{ od}\}\alpha))$$

Reguła wnioskowania – wprowadzanie while w poprzedniku implikacji.

$$\frac{\left\{M(\textbf{if } \gamma \textbf{ then } K \textbf{ fi})^i\, \alpha \Rightarrow \beta\right\}_{i\in N}}{\{M; \textbf{while } \gamma \textbf{ do } K \textbf{ od}\}\, \alpha \Rightarrow \beta}$$

7.2 Programowanie z instrukcją while

TODO

- prawo Archimedesa, zasada wyczerpywania Eudoksosa, ...
- opisz obliczanie zera wielomianu wyższego stopnia – przygotowanie do bisekcji,
- obliczanie całki oznaczonej,
- przykład funkcji obliczalnej, ale nie pierwotnie rekurencyjnej czyli funkcja Ackermanna na ten temat można napisać sporo:
-

7.3 składnia i semantyka

Rozszerzenie języka wydaje się niepozorne. Ale pojawiają się nowe zjawiska obliczeniowe.
TODO
Rozwinąć to. Do tej pory nie mieliśmy problemu z zapewnieniem własności stopu programu. Każdy program, omawiany we wcześniejszych rozdziałach, ma obliczenie skończone. Wprowadzenie do języka programowania instrukcji iteracji while, z jednej strony w istotny sposób zwiększa zbiór funkcji obliczalnych, z drugiej strony tracimy gwarancję, że obliczenie programu będzie skończone. Pojawia się konieczność udowodnienia, że obliczenia programu będą skończone. A to nie zawsze jest łatwe i czasami stanowi wyzwanie dla pokoleń badaczy. W latach 30 dwudziestego wieku Ackermann zdefiniował, funkcję, która rósnie bardzo szybko i wykazał, że nie jest ona funkcją pierwotnie rekurencyjną. W przetłunaczeniu na dzisiejszy język oznacza to tyle, że funkcji Ackermana nie można zaprogramować ograniczając się do instrukcji for.

składnia

Zbiór programów iteracyjnych różni się tym od poprzednio opisanego zbioru instrukcji z tablicami, że w ciągu instrukcji takiego programu mogą pojawiać się instrukcje while.
Zbiór instrukcji programów iteracyjnych jest to najmniejszy zbiór P napisów taki, że

i) każda instrukcja przypisania należy do zbioru P,

ii) złożenie, for oraz rozgałęzienie

iii) jeśli napis γ jest formułą boolowską i napis K jest instrukcją, to napis while γ do K od jest instrukcją iteracji

TODO Poprawić tę definicję.
Przykłady

Semantyka

Znaczenie instrukcji while opisuje aksjomat i reguła wnioskowania.
Aksjomat while

$$\textbf{while } \gamma \textbf{ do } K \textbf{ od}\, \alpha \Leftrightarrow ((\neg\gamma \wedge \alpha) \vee (\gamma \wedge \{K; \textbf{while } \gamma \textbf{ do } K \textbf{ od}\}\alpha))$$

Reguła wnioskowania – wprowadzanie while w poprzedniku implikacji.

$$\frac{\left\{M(\textbf{if } \gamma \textbf{ then } K \textbf{ fi})^i\, \alpha \Rightarrow \beta\right\}_{i\in N}}{\{M; \textbf{while } \gamma \textbf{ do } K \textbf{ od}\}\, \alpha \Rightarrow \beta}$$

7.4 Programowanie z instrukcją while

TODO

- prawo Archimedesa, zasada wyczerpywania Eudoksosa, ...
- opisz obliczanie zera wielomianu wyższego stopnia – przygotowanie do bisekcji,
- obliczanie całki oznaczonej,
- przykład funkcji obliczalnej, ale nie pierwotnie rekurencyjnej czyli funkcja Ackermanna na ten temat można napisać sporo:
-

7.5 Własności instrukcji while

Instrukcja while jest idempotentna

$$\{\textbf{while } \gamma \textbf{ do } M \textbf{ od}\}\alpha \Leftrightarrow \{\textbf{while } \gamma \textbf{ do } M \textbf{ od}; \textbf{while } \gamma \textbf{ do } M \textbf{ od}\}\alpha$$

A więc nie warto powtarzać tej instrukcji dwukrotnie.
Instrukcja while zawiera w sobie instrukcję if

$$\{\textbf{if } \gamma \textbf{ then } M \textbf{ fi}; \textbf{while } \gamma \textbf{ do } M \textbf{ od}\}\alpha \Leftrightarrow \{\textbf{while } \gamma \textbf{ do } M \textbf{ od}\}\alpha$$

$$\{\textbf{while } \gamma \textbf{ do } M \textbf{ od}; \textbf{if } \gamma \textbf{ then } M \textbf{ fi}\}\alpha \Leftrightarrow \{\textbf{while } \gamma \textbf{ do } M \textbf{ od}\}\alpha$$

I tak jest dla każdej iteracji instrukcji warunkowej if. Dla każdej liczby $i \in N$ zachodzą następujące równoważności

$$\{(\textbf{if } \gamma \textbf{ then } M \textbf{ fi})^i; \textbf{while } \gamma \textbf{ do } M \textbf{ od}\}\alpha \Leftrightarrow \{\textbf{while } \gamma \textbf{ do } M \textbf{ od}\}\alpha$$

$$\{\textbf{while } \gamma \textbf{ do } M \textbf{ od}; (\textbf{if } \gamma \textbf{ then } M \textbf{ fi})^i\}\alpha \Leftrightarrow \{\textbf{while } \gamma \textbf{ do } M \textbf{ od}\}\alpha$$

Pomocnicza reguła wnioskowania

$$\frac{\alpha \Rightarrow \beta}{\textbf{while } \beta \textbf{ do } M \textbf{ od}\, true \Rightarrow \textbf{while } \alpha \textbf{ do } M \textbf{ od}\, true}$$

Jeszcze inna reguła

$$\frac{\textbf{while } \gamma \textbf{ do } M \textbf{ od}\, \alpha,\ K\gamma \Leftrightarrow \gamma}{\textbf{while } \gamma \textbf{ do } M; K \textbf{ od}\, \alpha}$$

pętle do – od

Pętla

$$\textbf{do } K;\ \textbf{if } \gamma \textbf{ then exit fi od}$$

jest równoważna programowi

$$\{K;\ \textbf{while } \gamma \textbf{ do } K \textbf{ od}\}$$

W niektórych językach programowania można znaleźć instrukcję **repeat**

$$\textbf{repeat } K \textbf{ until } \gamma\,;$$

Wprowadzając polecenie **do ... od** oraz instrukcje **exit** i **repeat** stwarzamy możliwość samodzielnego projektowania nowych instrukcji iteracyjnych. Np.

7.6 Algorytmiczne aksjomaty typów pierwotnych

Rozszerzeniu języka programowania odpowiada rozszerzenie języka formuł algorytmicznych. Okazuje się, że tak wzbogacony język pozwala sformułować własności struktur danych.

7.6.1 Aksjomaty typu integer

W rozdziale 3 o wyrażeniach podaliśmy aksjomaty opisujące typ integer jako aksjomaty pierścienia z dołączonym aksjomatem dobrego ufundowania. Ten ostatni aksjomat jest kłopotliwy, ponieważ występuje w nim kwantyfikator wiążący podzbiory zbioru liczb całkowitych. W miejsce tamtego aksjomatu należy(!) użyć następującej formuły algorytmicznej

$$\forall_{x \geq 0}\, \{y \leftarrow 0; \textbf{while } y \neq x \textbf{ do } y \leftarrow y + 1 \textbf{ od}\}(y = x)$$

wszystkie pozostałe aksjomaty zostają zachowane.
Ta aksjomatyzacja ma wiele zalet, najważniejszą z nich jest zapewnienie, że pewien program na pewno kończy obliczenia. Z tego faktu można wyprowadzać inne podobne twierdzenia dotyczące algorytmów. Np. prawo Archimedesa, własność stopu algorytmu Euklidesa itp.

7.6.2 Aksjomaty typu real

Do poprzednio wymienionych aksjomatów ciała możemy teraz dodać aksjomat (prawo?) Archimedesa

$$((0 < y \wedge y < x) \Rightarrow \{t \leftarrow y;\ \textbf{while}\ t \leq x\ \textbf{do}\ t \leftarrow t + y\ \textbf{od}\}(t > x)) \qquad \text{(Arch)}$$

Erwin Engeler [] udowodnił, że dla każdej formuły algorytmicznej postaci $K\alpha$ formuła taka jest prawdziwa w ciele liczb rzeczywistych z porządkiem wtedy i tylko wtedy gdy jest prawdziwa w każdym ciele uporządkowanym spełniającym aksjomat Archimedesa.

Wykorzystując twierdzenie o pełności logiki algorytmicznej (tj. rachunku programów) możemy twierdzenie Engelera sformułować tak.

Twierdzenie 7.1. *Niech Z oznacza zbiór formuł algorytmicznych na który składają się aksjomaty ciała uporządkowanego oraz aksjomat Archimedesa Arch. Niech ψ oznacza Boolowską kombinację formuł algorytmicznych (tj. formuł postaci $K\alpha$). Formuła ψ jest prawdziwa w uporządkowanym ciele $\mathfrak{RO}$ liczb rzeczywistych wtedy i tylko wtedy, gdy istnieje dowód tej formuły z aksjomatów zbioru Z.*

$$\mathfrak{RO} \models K\alpha \text{ wtedy i tylko wtedy, gdy } Z \vdash \psi$$

Sens tego twierdzenia sprowadza się do następującej obserwacji, dla semantycznej własności w_ψ programów z języka $\mathcal{L}_6$ realizowanych w ciele liczb rzeczywistych z porzadkiem, jeśli własnośc ta jest prawdziwa w ciele $\mathfrak{RO}$ to jest dowodliwa z aksjomatów zbioru Z. Jest tak, ponieważ widzieliśmy, że własności semantyczne takie jak stop programu, poprawność programu, własność fault, równoważność, etc. sa wyrażalne formułami będącymi boolowskimi kombinacjami formuł postaci $K\alpha$.

Rozdział 8

Rachunek programów

Do tej pory omówiliśmy wszystkie niezbędne konstrukcje programotwórcze. Programowanie w strukturze liczb całkowitych z wykorzystaniem instrukcji przypisania, złożenia (tj. średnika ;), oraz instrukcji while pozwala zaprogramować każdą funkcję obliczalną. Twierdzenie o takiej treści (innymi słowami) wypowiedział i udowodnił Stephen C. Kleene w roku 1936 [17].

Twierdzenie 8.1. *Prawdziwe są inkluzje:*

(i) Każda funkcja o obliczalna w dziedzinie liczb całkowitych jest programowalna w języku $\mathcal{L}_5$.

(ii) Każda funkcja f programowalna w języku $\mathcal{L}_5$, (zakładamy, dla uproszczenia, że w programie ją definiującym nie występują wyrażenia typu real) jest obliczalna w dziedzinie liczb całkowitych.

W związku z tym rodzi się wiele pytań. Widzieliśmy, że wiele własności semantycznych programów można wyrazić jako formuły algorytmiczne. W tym rozdziale odpowiadamy na kilka podstawowych pytań:

- czy podane tu aksjomaty i reguły wnioskowania o programach stanowią kompletny zestaw narzędzi?
- czy aksjomaty rachunku programów opisują semantykę programów w sposób jednoznaczny?
- jakie wnioski można wyciągnąć z twierdzenia o pełności rachunku programów tj. logiki algorytmicznej?

8.1 Rachunek programów

Rachunek programów jest rachunkiem logicznym (zob. Rys.1 str. xi) będącym rozszerzeniem rachunku predykatów. Używamy zamiennie nazw *logika algorytmiczna* i *rachunek programów*.
Rachunek programów to zbiór systemów formalnych. Na każdy taki system składają się język $\mathcal{L}$ i operacja konsekwencji syntaktycznej (logicznej) $\mathcal{C}$. System $\langle \mathcal{L}, \mathcal{C} \rangle$ nazywać będziemy *systemem dedukcyjnym*. Wszystkie języki tego

zbioru mają podobną strukturę. Każdy taki język ...

Oprócz systemów dedukcyjnych rozpatrywane są też *teorie*. Teoria $\mathcal{T} = \langle \mathcal{L}, \mathcal{C}, \mathcal{A} \rangle$. Ostatni element trójki $\mathcal{A}$ to zbiór formuł innych niz aksjomaty logiki. Mówimy *aksjomaty specyficzne teorii* $\mathcal{T}$.

axioms

Aksjomaty rachunku programów

Definicja 8.1. *Aksjomatem rachunku programów jest każda formuła, której struktura jest zgodna z jednym z poniższych schematów aksjomatów Ax1 – Ax23.*

Przykład 8.2. *Jest aksjomatem formuła*

$$(\underbrace{x < y}_{\alpha} \wedge \underbrace{z^2 < 0}_{\beta} \Rightarrow \underbrace{z^2 < 0}_{\beta}) \ .$$

Wskaż, który to aksjomat.

więcej

Ax_1 $((\alpha \Rightarrow \beta) \Rightarrow ((\beta \Rightarrow \delta) \Rightarrow (\alpha \Rightarrow \delta)))$

Ax_2 $(\alpha \Rightarrow (\alpha \vee \beta))$

Ax_3 $(\beta \Rightarrow (\alpha \vee \beta))$

Ax_4 $((\alpha \Rightarrow \delta) \Rightarrow ((\beta \Rightarrow \delta) \Rightarrow ((\alpha \vee \beta) \Rightarrow \delta)))$

Ax_5 $((\alpha \wedge \beta) \Rightarrow \alpha)$

Ax_6 $((\alpha \wedge \beta) \Rightarrow \beta)$

Ax_7 $((\delta \Rightarrow \alpha) \Rightarrow ((\delta \Rightarrow \beta) \Rightarrow (\delta \Rightarrow (\alpha \wedge \beta))))$

Ax_8 $((\alpha \Rightarrow (\beta \Rightarrow \delta)) \equiv ((\alpha \wedge \beta) \Rightarrow \delta))$

Ax_9 $((\alpha \wedge \neg\alpha) \Rightarrow \beta)$

Ax_{10} $((\alpha \Rightarrow (\alpha \wedge \neg\alpha)) \Rightarrow \neg\alpha)$

Ax_{11} $(\alpha \vee \neg\alpha)$

Ax_{12} $((x := \tau)true \Rightarrow ((\forall x)\alpha(x) \Rightarrow (x := \tau)\alpha(x))),$
gdzie term τ jest tego samego typu co zmienna x

Ax_{13} $(\forall x)\alpha(x) \equiv \neg(\exists x)\neg\alpha(x)$

Ax_{14} $K((\exists x)\alpha(x)) \equiv (\exists y)(K\alpha(x/y)),$ dla $y \notin V(K)$

Ax_{15} $K(\alpha \vee \beta) \equiv ((K\alpha) \vee (K\beta))$

Ax_{16} $K(\alpha \wedge \beta) \equiv ((K\alpha) \wedge (K\beta))$

Ax_{17} $K(\neg\alpha) \Rightarrow \neg(K\alpha)$

Ax_{18} $((x := \tau)\gamma \equiv (\gamma(x/\tau) \wedge (x := \tau)true)) \wedge ((q := \gamma\prime)\gamma \equiv \gamma(q/\gamma\prime))$

Ax_{19} **begin** $K; M$ **end** $\alpha \equiv K(M\alpha)$

Ax_{20} **if** γ **then** K **else** M **fi** $\alpha \equiv ((\neg\gamma \wedge M\alpha) \vee (\gamma \wedge K\alpha))$

Ax_{21} **while** γ **do** K **od** $\alpha \equiv ((\neg\gamma \wedge \alpha) \vee (\gamma \wedge K(\textbf{while } \gamma \textbf{ do } K \textbf{ od}(\alpha))))$

Ax_{22} $\bigcap K\alpha \equiv (\alpha \wedge (K \bigcap K\alpha))$

Ax_{23} $\bigcup K\alpha \equiv (\alpha \vee (K \bigcup K\alpha))$

Reguły wnioskowania

Definicja 8.3. *Reguła wnioskowania to relacja (operacja) ...*

$$R_1 \quad \frac{\alpha, (\alpha \Rightarrow \beta)}{\beta}$$

$$R_2 \quad \frac{(\alpha \Rightarrow \beta)}{(K\alpha \Rightarrow K\beta)}$$

$$R_3 \quad \frac{\{s(\textbf{if } \gamma \textbf{ then } K \textbf{ fi})^i(\neg\gamma \wedge \alpha) \Rightarrow \beta\}_{i\in N}}{(s(\textbf{while } \gamma \textbf{ do } K \textbf{ od } \alpha) \Rightarrow \beta)}$$

$$R_4 \quad \frac{\{(K^i\alpha \Rightarrow \beta)\}_{i\in N}}{(\bigcup K\alpha \Rightarrow \beta)}$$

$$R_5 \quad \frac{\{(\alpha \Rightarrow K^i\beta)\}_{i\in N}}{(\alpha \Rightarrow \bigcap K\beta)}$$

$$R_6 \quad \frac{(\alpha(x) \ \Rightarrow \ \beta)}{((\exists_x)\alpha(x) \ \Rightarrow \ \beta)}$$

$$R_7 \quad \frac{(\beta \ \Rightarrow \ \alpha(x))}{(\beta \Rightarrow (\forall_x)\alpha(x))}$$

W regułach R_6 i R_7, przyjmuje się, że x nie jest zmienną wolną w formule β, t.j. $x \notin FV(\beta)$. Reguła R_6 jest regułą wprowadzania kwantyfikatora egzystencjalnego w poprzedniku implikacji. Reguła R_7 jest regułą wprowadzania kwantyfikatora ogólnego w następniku implikacji.
Reguły R_4 i R_5 są algorytmicznymi odpowiednikami reguł R_6 i R_7, pozwalają wprowadzić kwantyfikator iteracji. Reguła R_4 wprowadza szczegółowy kwantyfikator iteracji w poprzedniku implikacji. Reguła R_5 wprowadza ogólny kwantyfikator iteracji w następniku implikacji. Reguły te mają jednak inny charakter, ponieważ zbiory ich przesłanek są zbiorami nieskończonymi. Reguła R_3 wprowadzania instrukcji **while** w poprzedniku implikacji też ma nieskończony zbiór przesłanek. Reguły R_3, R_4, R_5 są nazywane regułami ω.

Operacja syntaktycznej konsekwencji $\mathcal{C}$

Definicja 8.4. *Niech X będzie zbiorem formuł ustalonego języka $\mathcal{L}$. Zbiór $\mathcal{C}(X)$ syntaktycznych konsekwencji zbioru X jest to najmniejszy zbiór formuł taki, że*

- *Zbiór $\mathcal{C}(X)$ zawiera wszystkie aksjomaty rachunku programów $\mathcal{A}x$, i*
- *zbiór $\mathcal{C}(X)$ zawiera wszystkie formuły ze zbioru X, i*
- *jeśli do zbioru $\mathcal{C}(X)$ należą wszystkie przesłanki pewnej reguły wnioskowania R_i, $i = 1, \ldots, 7$ to do zbioru $\mathcal{C}(X)$ należy też konkluzja tej reguły.*

Własności operacji konsekwencji $\mathcal{C}$

c1) $X \subset \mathcal{C}(X)$,

c2) $X \subset Y$ pociąga $\mathcal{C}(X) \subset \mathcal{C}(Y)$,

c3) $\mathcal{C}(X) = \mathcal{C}(\mathcal{C}(X))$

Niech $\mathcal{T} = \langle \mathcal{L}, \mathcal{C}, \mathcal{A} \rangle$ będzie teorią algorytmiczną.

Definicja 8.5. *Twierdzeniem teorii $\mathcal{T}$ jest każda formuła α należąca do zbioru $\mathcal{C}(\mathcal{A})$.*

Na oznaczenie tego faktu można stosować różne zapisy:

$\mathcal{A} \vdash \alpha$ – co się czyta *formuła α posiada dowód ze zbioru $\mathcal{A}$*, lub

$\alpha \in \mathcal{C}(\mathcal{A})$ – formuła α jest konsekwencją zbioru $\mathcal{A}$.

Przykład 8.6. *tu przykłady*

8.2 Pełność AL

W tym miejscu zastanowimy się nad pytaniami:

- czy jeśli udowodnię formułę α, to czy jest ona prawdziwa?
- czy formuła prawdziwa posiada dowód?
- czy formuła wyprowadzona przy pomocy rachunku programów i aksjomatów struktury liczb całkowitych jest prawdziwa w tej strukturze?
- czy każdy model algorytmicznej teorii liczb całkowitych jest izomorficzny z modelem wzorcowym (lepiej powiedz, modelem standardowym)?
- czy teoria niesprzeczna posiada model?

Tw. o pełności i co z niego wynika

Twierdzenie 8.2. (o pełności rachunku programow)
Niech $\mathcal{T} = \langle \mathcal{L}, \mathcal{C}, \mathcal{A} \rangle$ będzie teorią niesprzeczną, niech α będzie formułą w języku tej teorii.
Następujące zdania są równoważne:

i) Formuła α jest twierdzeniem teorii $\mathcal{T}$, $\mathcal{A} \vdash \alpha$

ii) formuła α jest prawdziwa w każdym modelu teorii $\mathcal{T}$, $\mathcal{A} \models \alpha$.

Przykład. Poprawność programu swap - w każdej strukturze danych program swap jest poprawny. Nawet w tych o których dopiero będziemy mówić.
Zauważ, że dowód poprawności algorytmu swap korzysta wyłącznie z dwóch aksjomatów rachunku programów i żadnego aksjomatu jakiejś struktury danych. Zgodnie z twierdzeniem o pełności własność *algorytm swap zamienia miejscami wartości zmiennych x i y* jest prawdziwa w kazdej strukturze danych.

Algorytm Euklidesa - poprawnośc nie jest tautologią. (Szok Pitagorejcyków) Ale alg. Euklidesa zatrzymuje sie w strukturze liczb całkowitych i potrafimy to udowodnic.

8.3 AL definiuje semantykę programów while

W tym podrozdziale kontynuujemy dyskusję na temat związku semantyki języka(-ów) programowania i rachunku programów. Wcześniej ustaliliśmy taki związek pomiędzy operacjami syntaktycznej i semantycznej konsekwencji. Obecnie, podejmujemy próbę zrozumienia w jaki sposób przyjęcie systemu formalnego logiki algorytmicznej (tj. rachunku programów) może wpłynąć na realizację maszyny wirtualnej i kompilatora języka programowania. Mogliśmy wcześniej zaobserwować, że semantyczne reguły obliczania wartości formuł i/lub wykonywania programów znajdują swoje odpowiedniki w schematach aksjomatów.
Głównym pytaniem, jakie stawiamy w tym podrozdziale jest: *w jakim stopniu aksjomaty determinują sposób wykonywania programów, tj ich semantykę.* Czy można przyjąć system dedukcyjny logiki algorytmicznej jako definicję semantyki programów (napisanych w języku $\mathcal{L}_5$).

Definicja 8.7. *Strukturą semantyczną dla algorytmicznego języka $\mathcal{L}$ nazywać będziemy trójkę $\langle \mathbb{A}, I, \models \rangle$, gdzie,*

$\mathbb{A}$ jest strukturą algebraiczną,

I jest interpretacją programów (tj. funkcją, która każdemu programowi przyporządkowuje pewna relację binarną w zbiorze wartościowań zmiennych w strukturze $\mathbb{A}$) i

$\models$ jest relacją spełnialności,

przy czym trójka $\langle \mathbb{A}, I, \models \rangle$ ma następujące własności

(1) Struktura algebraiczna $\mathbb{A}$ jest ekspresywna, tj. dla każdego elementu $a \in A$ istnieje taki term τ_a w języku $\mathcal{L}$, że $\tau_{a\mathbb{A}}(v) = a$, miezależnie od wartosciowania v.

(2) Interpretacja termów i formuł pierwsego rzedu jest zdefiniowana tak jak w logice pierwszego rzędu (por.).

(3) Nastepująca własność relacji spełnialności zachodzi dla każdego wartościowania v w A

$$\mathbb{A}, v \models M\alpha \text{ iff } (\exists v')(v, v' \in I(M) \text{ and } \mathbb{A}, v' \models \alpha.$$

W przeciwieństwie do semantyki opisanej w poprzednich rozdziałach nie będziemy zakładać niczego o mechanizmach obliczeń ani o interpretacji operatorów programotwórczych. Krótko mówiąc nie znamy komputera $\mathcal{K}_5$.
Jedyne założenie jakie przyjmiemy to, że każdy program wyznacza relację binarna w zbiorze wartościowań zmiennych.

Niech trójka $< \mathbb{A}, I, \models >$ będzie strukturą semantyczną. Wtedy ma miejsce następująca własność separowalności.

Lemat 8.3. *Dla każdej pary wartościowań zmiennych v, v' in $\mathbb{A}$, wartościowania te są różne, $v \neq v'$ wtedy i tylko wtedy gdy istnieje pewna formuła α je odróżniająca tj. taka, że $\mathbb{A}, v \models \alpha$ i $\mathbb{A}, v' \models \neg\alpha$.*

Dowód. Let v, v' be two different valuations in the structure $\mathbb{A}$ such that $v \neq v'$. Therefore there exists a variable z such that $v(z) \neq v'(z)$. If z is a propositional

variable q, then setting $\alpha = q$ (in the case $v(q) = 1$) or $\alpha = \neg q$ (in the case $v(q) = 0$), we have $\mathbb{A}, v \models \alpha$ and $\mathbb{A}, v' \models \neg\alpha$.
If z is an individual variable x and $v(x) = a$, then setting $\alpha = (\tau_a = x)$ (recall the data structure (A) is expressive – c.f. assumption (1)), we have $\mathbb{A}, v \models (\tau_a = x)$ and *non* $\mathbb{A}, v' \models (\tau_a = x)$.
The inverse implication is easy. If $v = v'$, i.e. for every z, $v(z) = v'(z)$, then for every formula α, $\mathbb{A}, v \models \alpha$ iff $\mathbb{A}, v' \models \alpha$. □

Definicja 8.8. *A semantic structure* $< A, I, \models >$ *is called standard iff the following conditions are satisfied*

$$I(x := \omega) = \{(v, v') : v'(x) = \omega_{\mathbb{A}}(v),\ v'(z) = v(z) \text{ for } z \neq x\}$$
$$I(\textbf{begin } K; M \textbf{ end}) = I(K) \circ I(M)$$
$$I(\textbf{if } \gamma \textbf{ then } K \textbf{ else } M \textbf{ fi}) = id_{\mathbb{A}}(\gamma) \circ I(K) \cup id_{\mathbb{A}}(\neg\gamma) \circ I(M)$$
$$I(\textbf{while } \gamma \textbf{ do } K \textbf{od}) = \bigcup I(\textbf{if} \gamma \textbf{ then } K \textbf{ fi})^i \circ id_{\mathbb{A}}(\neg\gamma),$$

for every programs $K, M \in P$, *for every open formula* $\gamma \in F_o$,*for every variable* $x \in V$,*for every expression* ω, *where* $id_{\mathbb{A}}(\gamma) = \{(v, v) : \mathbb{A}, v \models \gamma\}$.

Lemat 8.4. *If* $\langle \mathbb{A}, I, \models \rangle$ *is a standard semantic structure, then for any program* M *and any valuations* v, v' *in* $\mathbb{A}$,

$$(v, v') \in I(M) \text{ iff } (v, v') \in M_{\mathbb{A}}.$$

i.e. the relation $I(M)$ *coincides with relation determined by the standard computation of the program* M *in data structure* $\mathbb{A}$.

The proof follows from the above definitions.

Definicja 8.9. *A semantic structure* $\langle \mathbb{A}, I, \models \rangle$ *is a model of the formal system* $AL(\Pi)$ *iff for each axiom* α *of* $AL(\Pi)$, $\mathbb{A} \models \alpha$, *and for each inference rule* r *of* $AL(\Pi)$, *validity of all premises of* r *implies validity of the conclusion of* r.

As an immediate consequence of Lemma 4.5.2 and Theorem 4.2.3 (on adequacy), we have the following fact

Twierdzenie 8.5. *Each standard semantic structure is a model for the formal system of algorithmic logic* $AL(\Pi)$.

In the rest of this section we examine the converse implication by considering these properties of interpretation I which are forced by particular axioms.

Lemat 8.6. *If a semantic structure* $\langle \mathbb{A}, I, \models \rangle$ *is a model of all formulas of the form of axiom Ax16*
Ax16 : $M(\alpha \wedge \beta) \equiv (M\alpha \wedge M\beta)$
where α, β *are any algorithmic formulas, then for any program* $M, I(M)$ *is a partial function on the set of all valuations in* $\mathbb{A}$.

Dowód. Suppose that, for some valuations v, v', v'' we have $v'' \neq v'$, $(v, v'') \in I(M)$ and $(v, v') \in I(M)$. Then by Lemma 8.3 there exists a formula α_0 such that $A, v' \models \alpha_0$ and $A, v'' \models \neg\alpha_0$. Thus $A, v \models M\alpha_0$ and $A, v \models M\neg\alpha_0$. By axiom Ax16 we have $A, v \models M(\alpha_0 \wedge \neg\alpha_0)$, this however is impossible. We have thus proved that, for every valuation v, there exists at most one valuation v' such that $(v, v') \in I(M)$, i.e. for every program M, $I(M)$ is a partial function. □

Lemat 8.7. *If a semantic structure* $\langle \mathbb{A}, I, \models \rangle$ *is a model for axioms Ax16 and Ax19*
Ax19 **begin** $K; M$ **end** $\alpha \iff K(M\alpha)$,
then, for any programs K *and* $M, I(begin\ K; M\ end) = I(K) \circ I(M)$.

Dowód. Let $(v, v') \in I($**begin** $K; M$ **end**$)$ and $A, v' \models \alpha$. Thus
$A, v \models$ **begin** $K; M$ **end** α, and, by axiom Ax19, $A, v \models K(M\alpha)$. By the definition of a semantic structure, there are valuations v_2, v_3 such that $(v, v_2) \in I(K), (v_2, v_3) \in I(M)$ and $A, v_3 \models \alpha$. According to Lemma 8.6, v_2 and v_3 are uniquely determined, independently of the formula α. We can repeat the same argument for any arbitrary formula β, and prove that $A, v' \models \beta$ implies $A, v_3 \models \beta$. Thus, by Lemma 8.3, $v_3 = v'$ and $(v, v_2) \in I(K), (v_2, v') \in I(M)$. As a consequence $(v, v') \in I(K) \circ I(M)$.The opposite inclusion can be proved analogously. □

Lemat 8.8. *If a semantic structure* $\langle \mathbb{A}, I, \models \rangle$ *is a model for axioms Ax19, Ax16 and Ax20*
Ax20 **if** γ **then** K **else** M **fi**$\alpha \iff (\gamma \wedge K\alpha) \vee (\neg\gamma \wedge M\,\alpha)$,
then for any programs K, M *and any open formula* γ,
$I($**if** γ **then** K **else** M **fi**$) = id_{\mathbb{A}}(\gamma) \circ I(K) \cup id_{\mathbb{A}}(\neg\gamma) \circ I(M)$.

The proof is left to the reader.

Lemat 8.9. *If a semantic structure* $\langle \mathbb{A}, I, \models \rangle$ *is a model for axioms Ax16, Ax19, Ax20 and*
Ax21 **while** γ **do** K **od**$\,\alpha \iff (ok(\gamma) \wedge ((\neg\gamma \wedge \alpha) \vee (\gamma \wedge K($**while** γ **do** K **od**$\,\alpha))))$,
then for any formula γ *and for any program* M
$\bigcup I((\textbf{if}\ \gamma\ \textbf{then}\ M\ \textbf{fi})^i) \circ id_{\mathbb{A}}(\neg\gamma) \subset I(\textbf{while}\ \gamma\ \textbf{do}\ M\ \textbf{od})$.

Dowód. Assume that $(v, v') \in \bigcup I((\textbf{if}\ \gamma\ \textbf{then}\ M\ \textbf{fi})^i) \circ id_{\mathbb{A}}(\neg\gamma)$. Thus there exists a natural number m such that
$A, v' \models \neg\gamma$ and $(v, v') \in I((\textbf{if}\ \gamma\ \textbf{then}\ M\ \textbf{fi})^m)$.
If, for some formula α, $A, v' \models \alpha$, then $A, v \models (\textbf{if}\gamma\textbf{then}M\textbf{fi})^m(\neg\gamma \wedge \alpha)$ and, by axiom Ax21,
$A, v \models$ **while** γ **do** M **od** α.
As a consequence there exists a valuation v'', which by lemma 8.6 is unique and such that
$(v, v'') \in I($**while** γ **do** M **od**$)$ and $A, v'' \models \alpha$.
Since the above argument can be repeated for any formula α, we have by lemma 8.3 that $v' = v''$. Thus finally
$(v, v') \in I($**while** γ **do** M **od**$)$. □

Lemat 8.10. *If a semantic structure* $\langle \mathbb{A}, I, \models \rangle$ *is a model for Ax16, Ax19, Ax20, Ax21 and if inference rule R3 is sound in* $\mathbb{A}$, *then*
$I($**while** γ **do** K **od**$) = \bigcup I((\textbf{if}\ \gamma\ \textbf{then}\ K\ \textbf{fi})^i) \circ id_{\mathbb{A}}(\neg\gamma)$.

Dowód. Let $(v, v') \in I($ **while** γ **do** M **od**$)$ and let $A, v' \models \alpha$. Assume that $x_1, x_2, ..., x_k$ are all individual variables and $q_1, ..., q_l$ are all the propositional variables that occur in the formula **while** γ **do** M **od**$\,\alpha$. Moreover, assume $v(x_j) = a_j$ for $j \leq k$. Let b be a formula describing the valuation v on the variables $x_1, x_2, ..., x_k, q_1, ..., q_l$, i.e.
$b = (x_1 = a_1) \wedge (x_2 = a_2) \wedge ... \wedge (x_k = a_k) \wedge q_1' \wedge ... \wedge q_l'$,

where the subformula
q_i' denotes simply q_i' if $v(q_i) = \mathbf{1}$ and
q_i' is the formula
$\neg q_i$ if $v(q_i) = \mathbf{0}$.

Hence $A, v \models b$ and $A, v \models$ *while g do M oda* and, therefore, $nonA, v \models$ (*while g do M od*)$a \Rightarrow \neg\ b$). By inference rule R3, there exists a natural number m, such that

$nonA \models (if\ g\ then\ M\ fi)m(a \wedge \neg g)\ \neg b$.

Thus there exists a valuation v'' such that
$A, v'' \models (if\ g\ then\ M\ fi)m(a \wedge \neg g)$ and $A, v\text{“} \models b$,
which implies, by the definition of formula b, that
$A, v \models (if\ g\ then\ M\ fi)m(a \wedge \neg g)$.

Assume that m is the smallest natural number with this property. There therefore exists a valuation v'', such that
$(v, v\text{“}) \in I(if\ g\ then\ M\ fi)m$ and $A, v\text{“} \models (a \wedge \neg g)$.

We will now prove that for any formula d, $A, v' \models d$ implies $A, v\text{“} \models (d \wedge \neg g)$. Following the same reasoning as above we have
$A, v \models (if\ g\ then\ M\ fi)j(d \wedge \neg g)$
for some natural number j. The assumption on m and the axiom Ax20 forces $m \leq j$. As a result $A, v \models (if\ g\ then\ M\ fi)m(d \wedge g)$. By lemma 4.5.4 and the definition of the valuation v'', we have
$A, v\text{“} \models (d \wedge \neg g)$.

We have thus proved that there exists a natural number m and a valuation v'' such that $(v, v'') \in I(if\ g\ then\ M\ fi)^m$ and for any formula d, if $A, v' \models d$ then $A, v'' \models (d \wedge \neg g)$. By Lemma 8.3 $v'' = v'$ and therefore
$(v, v') \in I((if\ \gamma\ then\ M\ fi)^m)\ \circ\ id_{\mathbb{A}}(\neg\gamma)$. □

Lemat 8.11. *If a semantic structure* $\langle \mathbb{A}, I, \models \rangle$ *is a model for axiom Ax18*
Ax18 $(x := \tau)\gamma(x)\ \equiv\ \gamma(x/\tau) \wedge (\tau = \tau)(q := \gamma)\gamma(q)\ \equiv\ \gamma(q/\gamma)$,
then for any individual variable x *and any term* τ
$I(x := \tau) = \{(v, v') : \tau_{\mathbb{A}}(v)$ *is defined and* $v'(x) = \tau_{\mathbb{A}}(v)$, *and* $v'(z) = v(z)$ *for* $z \neq x\}$.

pamietaj o termach czesciowych, czy je tu mamy??

and for every propositional variable q *and open formula* g'
$I(q := g') = \{(v, v') :\ v'(q) = g_A(v)$ *and* $v'(z) = v(z)$ *for* $z \neq q\}$.

Dowód. Let $(v, v') \in I(x := t), v'(x) = a$ and $v'(z) = b$ for some variable $z \neq x$. By part (1) of the definition of a semantic structure, there exist terms τ_a and τ_b without free variables such that
$A, v' \models\ (x = ta) \wedge (z = tb)$.
Thus
$A, v \models\ (x := t)((x = ta)\ \wedge\ (z = tb))$.
Applying axiom Ax18, we obtain
$A, v \models (ta = ta)\ \wedge\ (tb = tb)\ \wedge\ (t = t)$,
which means that $t_A(v)$ is defined and $tA(v) = taA(v) = a = v'(x)$ and $v(z) = tbA(v) = b = v'(z)$. Hence the inclusion $\subset$ has been demonstrated.

To prove the inverse inclusion let $tA(v)$ be defined and $v'(x) = a = tA(v), v'(z) = v(z) = b$. By the definition of a semantic structure, there exist ta and tb which are definitions of elements a and b of the universe of $\mathbb{A}$. Thus

$A, v' \models (x = ta) \wedge (z = tb) \wedge (t = t)$

Substituting g in axiom Ax18 by the formula $(x = ta) \wedge (z = tb)$ gives $A, v \models (x := t)((x = ta) \wedge (z = tb))$. By the definition of the satisfiability relation, there exists a valuation v" such that

$(v, v") \in I(x := t)$ and$A, v" \models (x = ta) \wedge (z = tb)$,

i.e.

$v"(x) = tav"(z) = tb$.

From our assumptions, we have $v"(x) = v'(x)$ and $v"(z) = v'(z)$. Since z was an arbitrary variable such that $z \neq x$, we can deduce that $v" = v'$, and therefore $(v, v') \in I(x := t)$.

The analogous proof for the assignment instruction of the form $(q := \gamma)$ is left to the reader. □

As an immediate consequence of the facts proved above we obtain the main result of this section

Twierdzenie 8.12. *Each semantic structure which is a model for the formal system of algorithmic logic is a standard structure.*

We have proved in this way that axioms of the system $AL(\Pi)$ determine properties of interpretation which guarantee that interpretation of programs is standard. By lemma 4.5.2 we have proved therefore the equivalence of the notion of model of $AL(\Pi)$ and of standard semantic structure. Thus by lemma 8.3 axioms of algorithmic logic AL(Π) uniquely determines the semantics of programs of the class Π. Moreover this is just the semantics defined by the notion of computation in the section **??**.

dopiszmy cos o innym podejsciu do semantyki aksjomatycznej!!!!!

8.4 Definicje

Zwracamy uwagę na podobieństwo definicji (jakie przyjmuje się w trakcie rozwijania teorii matematycznych) i deklaracji funkcji, jakie występują w programach.

Przypomnijmy parę istotnych faktów znanych w podstawach matematyki. Rozpatrujemy rachunek predykatów i teorie elementarne (pierwszego rzędu).
Dodanie do pewnej teorii matematycznej $\mathcal{T}$, która jest wyznaczona przez jej język $\mathcal{L}$, operację konsekwencji $\mathcal{C}$ i zbiór aksjomatów niebędących aksjomatami logiki $\mathcal{A}$

$$\mathcal{T} = \langle \mathcal{L}, \mathcal{C}, \mathcal{A} \rangle$$

pewnych definicji nowych funktorów i predykatów każe nam mówić o nowej teorii $\mathcal{T}' = \langle \mathcal{L}', \mathcal{C}', \mathcal{A}' \rangle$. Język $\mathcal{L}'$ nowej teorii jest bogatszy od języka $\mathcal{L}$ ponieważ wprowadzono nowe symbole funktorów i predykatów. Odpowiednio bogatsze są zbiory termów i formuł . $\mathcal{L} \nsubseteq \mathcal{L}'$
Operacja konsekwencji zachowuje wszystkie schematy aksjomatów i reguł wnioskowania, ale w ponieważ język $\mathcal{L}'$ jest bogatszy od języka $\mathcal{L}$, to mamy nowe aksjomaty logiki i nowe reguły wnioskowania.
Zbiór $\mathcal{A}'$ aksjomatów teorii $\mathcal{T}'$ powstaje przez dodanie do zbioru $\mathcal{A}$ zestawu dołączanych definicji. $\mathcal{A} \nsubseteq \mathcal{A}'$.

O logice pierwszego rzędu, a właściwie o teoriach pierwszego rzędu dowodzi się dwu faktów:

T1) Jeśli teoria $\mathcal{T}$ posiada model, to teoria $\mathcal{T}'$ też posiada model, a więc jest niesprzeczna.

T2) Każde twierdzenie α nowej teorii $\mathcal{T}' = \langle \mathcal{L}', \mathcal{C}', \mathcal{A}' \rangle$, które jest formułą języka $\mathcal{L}$ starej teorii jest twierdzeniem teorii $\mathcal{T}$.

$$\mathcal{C}(\mathcal{A}) = \mathcal{C}'(\mathcal{A}') \cap \mathcal{F}$$

Twierdzenie T2 nazywane jest twierdzeniem o nieistotnośći definicji. W języku angielskim używany jest zwrot *conservative extension*. Czyli teoria $\mathcal{T}'$ jest ...

Kolejny fakt o podobnej wymowie został wypowiedziany i udowodniony przez S. Kleenego [17].

T3) Każda funkcja rekurencyjna jest μ-rekurencyjna.

Oznacza to tyle, że każda funkcja obliczalna określona na zbiorze liczb całkowitych dodatnich, jest programowalna w języku $\mathcal{L}_5$. Podkreślmy, funkcja ma być obliczalna tzn. dla każdego argumentu ma istnieć określony wynik. Ten warunek jest fundamentalny. Niespełnienie tego warunku może spowodować sprzeczność w systemie wykorzystującynm formalizm Hoare'a. Pisali o tym M. Wand i M. O'Donnell.
Jesteśmy przekonani, że te twierdzenia T1) i T2) mówią nam, że jeśli program (lub jakiś jego moduł) zawiera deklaracje funkcji, to

- oznacza to tyle, że wzbogacamy język,
- struktura algebraiczna (model) zostaje wzbogacona,
- rozszerzenie jest nieistotne, w tym sensie, że dodanie deklaracji nowych funkcji nie wzbogaca zbioru twierdzeń,
- wzbogacenie struktury danych jest obliczalne w terminach dotychczasowej struktury danych. Tzn. jesli dotychczasowa struktura danych była obliczalna to nowa struktura jest też strukturą obliczalna

Pamiętajmy, że rachunek programów, czyli logika algorytmiczna, ma następująca własność:
Twierdzenie (Skolema-Loevenheima) *Jeśli teoria algorytmiczna ma model to posiada też model przeliczalny w dziedzinie liczb naturalnych.*

Warto też wspomnieć wynik Laszlo Kalmára [16], ...

W tym miejscu wykażemy nieistotnośc jawnych definicji w rachunku programów.

2 twierdzenia i dowody

Natomiast wprowadzanie definicji niejawnych (programiści nazywają takie definicje rekurencyjnymi) wymaga wielkiej ostrożności.

Zbiór definicji niejawnych może zawierać sprzeczności. W takim przypadku nie można udowodnic własności T1).

Zbiór definicji niejawnych, a nawet definicji algorytmicznych może byc *czczy* tzn. definicje mogą być spełniane przez wiele różnych funkcji.

Wreszcie definicje powinny gwarantowac istnienie wartości definiowanej funkcji.

Każda z tych własności semantycznych jest bardzo ważna i naruszenie którejkolwiek z nich może zakończyć się niepowodzeniem obliczeń.

Własność niesprzeczności (niejawnych tj. rekurencyjnych) deklaracji funkcji **nie jest tożsama** z własnościa stopu programu.
Dlaczego?

Podobnie z własnoscia semantyczną: niejawne(rekurencyjne) deklaracje funkcji z pewnego zbioru deklaracji Z, nie gwarantują jednoznaczności takich funkcji. Tego typu własność semantyczna powinna być rozpoznana i nsprawiona w miarę mozliwości. Nie jest to własnośc tożsama z brakiem własności stopu. Dlaczego?

Inaczej mówiąc, programista wprowadzający deklarację funkcji f, lub zestawu Z deklaracji funkcji powinien przeprowadzić dowód, że taki zestaw deklaracji jest niesprzeczny (a więc ma rozwiązanie) i że rozwiązanie jest tylko jedno.

tu przenieś z sekcji funkcje I

Zwrócimy też uwagę na problem niesprzeczności wprowadzanych definicji. Ma to istotne znaczenie dla programowania z funkcjami.

Rozdział 9

Bloki $\mathcal{L}_6$

Instrukcje bloku mają taką postać jak program (w wersji zaczynającej się od słowa block). Znane są co najmniej trzy powody dla których warto stosować instrukcje bloku:

- gdy chcemy podzielić pracę nad programem na dwa zespoły pracujące niezależnie wtedy każdy z nich może opracowywać swój blok. Powstałe bloki B_1 i B_2 mogą korzystać z wspólnych deklaracji programu, a każdy z nich może dla swoich potrzeb wprowadzić deklaracje D_1 (odp. D_2). rys.
- pewne instrukcje $\mathbb{I}$ wykorzystują dodatkowe zasoby (wprowadzane przez lokalne deklaracje) $\mathbb{D}$,
- podobieństwa i różnice w odniesieniu do instrukcji procedury
- gdy trzeba wykonać pewien algorytm w środowisku zadeklarownaym przez klasę K stosujemy instrukcję bloku prefiksowanego
  pref K **block** $\mathbb{D}$ **begin** $\mathbb{I}$ **end**. O tej instrukcji napiszemy obszerniej w następnej części Programuj z klasą.

9.1 Składnia

Do zbioru instrukcji dodajemy napisy zbudowane w następujący sposób.

block
$\mathbb{D}$
begin
$\mathbb{I}$
end

gdzie symbol $\mathbb{D}$ oznacza ciąg deklaracji, a symbol $\mathbb{I}$ oznacza ciąg instrukcji.
Poniższy przykład ukazuje pewną motywację dla wprowadzenia instrukcji bloku. Jeżeli chcemy napisać pewien ciąg instrukcji I, i w tym ciagu potrzebujemy zmiennych roboczych, to instrukcja bloku pozwoli nam wprowadzić takie zmienne bez naruszenia (tj. bez konfliktu) dotychczas używanych zmiennych. To jest ważne, bowiem pozwala rozdzielić pracę w zespole. Gdy jeden zespół ma zaprogramować blok B_1, a drugi ma za zadanie napisać blok B_2, to w

obu blokach można używać otoczenia współdzielonego i w każdym bloku, bezpiecznie, bezkonfliktowo używać własnego środowiska. W bloku B_1 wprowadzamy jego środowisko lokalne poprzez deklaracje D_1. W bloku B_2 deklaracje D_2 rozszerzaja wspólne środowisko w inny spsób.

rysunek?

Przykład 9.1.

```
program SWAPzBlokiem;
  const k= -12 , l=35 ;
  var x, y, t: integer;
begin
  x := k; y:= l; t:=x+y;
  block
    var t: integer
  begin
    t :=x;
    x:=y;
    y := t;
  end;
  writeln("x= ",x,"y= ",y, "t=",t)
end
```

Gdy wskazane jest by zmienna i używana w instrukcji for nie była użyta poza tą instrukcją, to możemy ograniczyc jej zasięg w taki sposób.

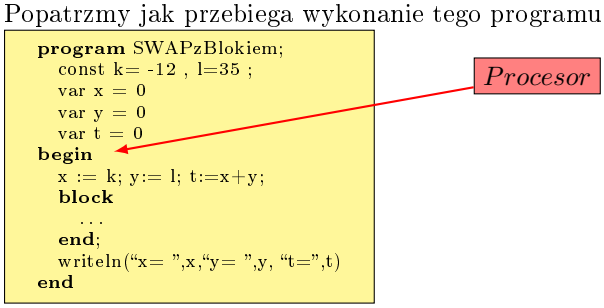
Przykład 9.2.

```
block
  var i: integer
begin
  s:= 0;
  for i:= 1 to n do s := s+ A(i)*B(i) od
end
```

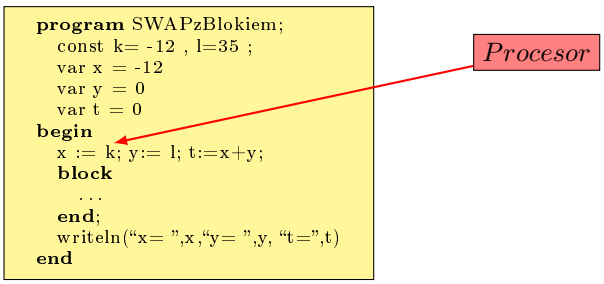
9.2 Komputer $\mathcal{K}_6$

Do zbioru jednostek dynamicznych dodawana jest nowa jednostka. Jej struktura jest podobna do struktury rekordu aktywacji programu głównego Main. Ale taka jednostka jest wyposażona w strzałkę SL , która przydaje się gdy wykonanie instrukcji napotyka na zmienną nielokalną. Rekord aktywacji bloku wyposażony jest w jeszcze jedną strzałkę DL.

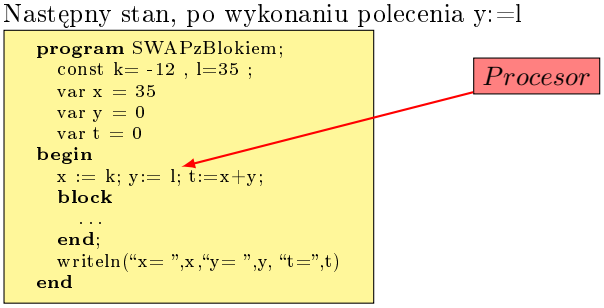
Przykłady
Popatrzmy jak przebiega wykonanie tego programu, stan początkowy.



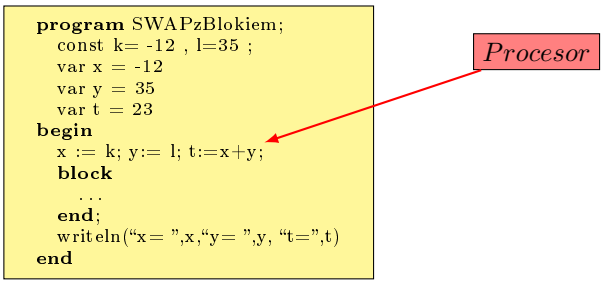
Następny stan, po wykonaniu polecenia x:=k



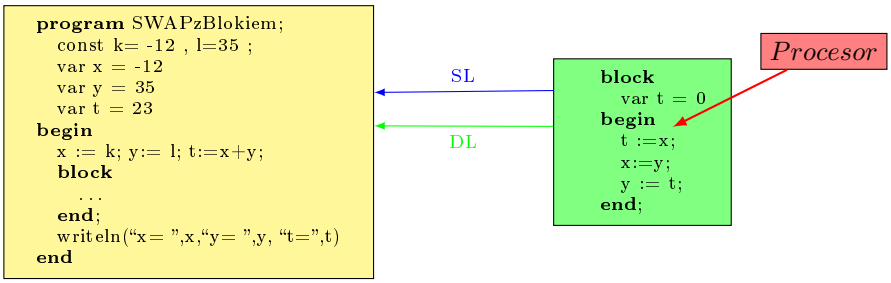
Następny stan, po wykonaniu polecenia y:=l



Następny stan, po wykonaniu polecenia t:=x+y.



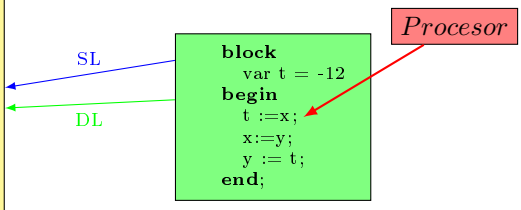
Następny stan, rozpoczynamy wykonywanie instrukcji bloku



Na rysunku widać, że komputer utworzył nową jednostkę dynamiczną (jej struktura jest podobna do rekordu aktywacji programu MAIN). Rekord aktywacji bloku jest wyposażony w dwie strzałki SL i DL. Ich znaczenie okaże się za chwilę. Procesor zaczyna wykonywanie instrukcji bloku.

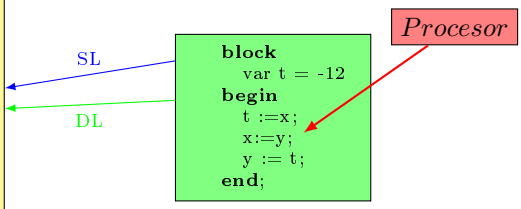
Przystępujemy do wykonania pierwszej instrukcji. Gdzie jest zmienna t? Procesor szuka zmiennej t w bloku i znajduje. Zmienna x? Nie ma jej wśród zmiennych zadeklarowanych w bloku – szukamy dalej przechodząc po strzałce SL. Znaleziono. Oto co widzimy po wykonaniu instrukcji t:=x.

```
program SWAPzBlokiem;
  const k= -12 , l=35 ;
  var x = -12
  var y = 35
  var t = 23
begin
  x := k; y:= l; t:=x+y;
  block
    ...
  end;
  writeln("x= ",x,"y= ",y, "t=",t)
end
```



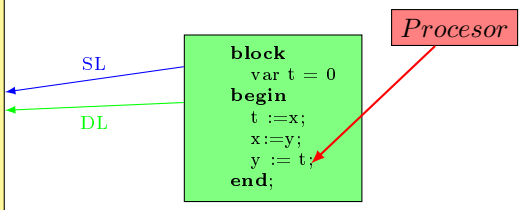
Teraz druga instrukcja. Łatwo się domyślić co się dzieje. Zmiana dokonuje się w rekordzie aktywacji MAIN. Oto co widzimy po wykonaniu instrukcji x:=y.

```
program SWAPzBlokiem;
  const k= -12 , l=35 ;
  var x = 35
  var y = 35
  var t = 23
begin
  x := k; y:= l; t:=x+y;
  block
    ...
  end;
  writeln("x= ",x,"y= ",y, "t=",t)
end
```



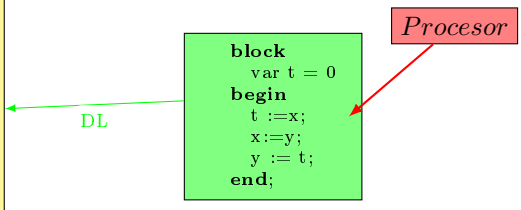
Ostania instrukcja bloku. Po wykonaniu polecenia y:=t widzimy taki obraz.

```
program SWAPzBlokiem;
  const k= -12 , l=35 ;
  var x = 35
  var y = -12
  var t = 23
begin
  x := k; y:= l; t:=x+y;
  block
    ...
  end;
  writeln("x= ",x,"y= ",y, "t=",t)
end
```

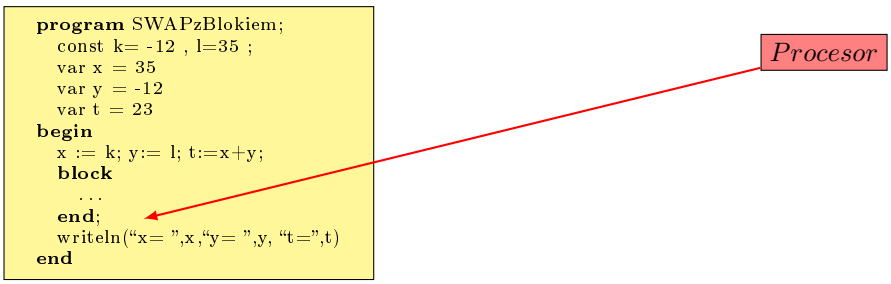


Zakończono wykonywanie bloku. Można usunąć rekord jego aktywacji. Gdzie należy kontynuować? Procesor przesuwamy wzdłuż strzałki DL. Za chwilę, w następnym rozdziale zobaczymy, że osobna strzałka DL jest niezbędna. Tylko w przypadku instrukcji bloku strzałki SL i DL mają ten sam początek i koniec.

```
program SWAPzBlokiem;
  const k= -12 , l=35 ;
  var x = 35
  var y = -12
  var t = 23
begin
  x := k; y:= l; t:=x+y;
  block
    ...
  end;
  writeln("x= ",x,"y= ",y, "t=",t)
end
```



Teraz można wydrukować wyniki.



Zapamiętajmy:

- procesor wskazuje na jeden rekord aktywacji: programu głównego MAIN lub rekord aktywacji jakiegoś bloku.
- SL – ta strzałka wskazuje gdzie jest rozszerzenie pamięci.
- DL – ta strzałka wskazuje procesorowi gdzie nalezy kontynuawać wykonywanie poleceń po wypełnieniu wszystkich zadań z bieżącego rekordu aktywacji.
- *bind* – funkcja przyporządkowująca wystąpieniu zmiennej x w instrukcji i odpowiedni rekord aktywacji, w którym instrukcja i może zapisać nową wartość zmiennej x, lub z którego instrukcja i odczyta wartość zmiennej x.
  Dokładniejszy opis tej funkcji podamy w następnych rozdziałach.

9.3 Semantyka

obmyśl przykład, w którym nowowprowadzone funkcje są sprzeczne ze zbiorem definicji w obejmującym module, ale gdy uwzględnimy zasłanianie nazw to nie ma sprzeczności!

Jak napisać aksjomat instrukcji bloku?
Wygląda na to, że każdy moduł (odp. każda jednostka dynamiczna) powinien być analizowany w nowej teorii. Dlaczego? Instrukcja bloku zawiera wyrażenia i instrukcje zawierające nowe zmienne (i zasłaniające niektóre zmienne zadeklarowane w module lub modułach obejmujących instrukcję bloku. Dalej, w bloku możemu zadklarować nowe procedury i funkcje, a nawet klasy. Oznacza to przyjęcie nowego zestawu definicji. Definicje te powinny być niesprzeczne (są to bowiem aksjomaty definiujące nowe nowe operacje) z tymi definicjami, które nie są zasłonięte przez nowo deklarowane funkcje i procedury.

Zacznijmy od następującego schematu: każda formuła następującej postaci jest tautologią.

$$\{\textbf{block}\ \text{var x : T}\ \textbf{begin}\ \text{I(x)}\ \textbf{end}\}\,\alpha \Leftrightarrow \{\textbf{block}\ \text{var y : T}\ \textbf{begin}\ \text{I(x/y)}\ \textbf{end}\}\,\alpha$$

gdzie zmienna y jest dobrana tak, że nie występuje w instrukcjach $I(x)$. Formuła α jest dowolna. Ale to nie opisuje całej prawdy Możesz przyjąć następujący schemat aksjomatów bloku

$$\{\text{block var x : T begin I(x) end}\}\,\alpha \Leftrightarrow \{\text{I(x/y)}\}\,\alpha$$

gdzie zmienna y nie występuje w instrukcjach I.
Grażyna zauważyła, że nie napisano nic o pozostałych zmiennych i innych deklaracjach.

Wydaje się, że brakuje mi pomysłu jak opisać środowisko.
W opisie komputera to się pokazuje, ale w formułach?

9.4 Teoria $\mathcal{T}_6$

A jak napisać aksjomat programu??

Jeśli w programie występują bloki, deklaracje funkcji, procedur, klas to mamy do czynienia z przechodzeniem od jednej teorii do teorii większej w tym sensie, że jej język jest bogatszy o: nowo zadeklarowane zmienne, nowe zadeklarowane funkcje etc.

Jak to się ma odzwierciedlić w rachunku programów?

Możemy formalizm podzielić na warstwy: wprowadzając definicje (przez indukcję) kolejnych warstw [trochę to może przypominac pracę Helmuta Thiele.]

Wniosek. Każdy program definiuje swoją własną teorię ponieważ deklaracje to forma zdefiniowania: zbioru zmiennych, zbioru definiowanych funkcji, zbioru definiowanych instrukcji (instrukcje procedury!), zbioru definiowanych typów obiektowych.

Co więcej, Struktura zagnieżdżeń modułów ujawnia, że mamy do czynienia z sekwencją teorii. W każdym module obowiązuje inny alfabet etc.

Informacja o grafie modułów i relacjach *decl* oraz *int* jest bardzo ważna.

Rozdział 10

Funkcje I $\mathcal{L}_8$

W matematyce wprowadzamy definicje symboli funkcyjnych i relacyjnych, po to, by operować krótszymi formułami. Formalnie rzecz biorąc, dodanie nowej definicji oznacza zmianę (rozszerzenie) języka teorii w której prowadzimy rozumowania. W naturalny sposób pojawiają się pytania: 1°czy przyjęcie nowej definicji nie prowadzi do sprzeczności? 2°czy przyjęcie nowej definicji pozwala udowodnić coś nowego (jakiś nowy fakt) o pojęciach starej teorii, czego nie można udowodnić w starej teorii? W związku z tym, wprowadzenie nowej definicji jest obłożone pewnymi warunkami.

W środowisku programistów, stwierdzamy to ze smutkiem, nikt nie zadaje sobie takich pytań, a powinien. Rozwiniemy ten temat poniżej.

funkcja nieokreslona

jednoznaczność f.

Przykład 10.1. *W strukturze danych liczby całkowite, czyli dla programistów w typie integer, można określić relację parzyste i operację* ***div2****.*
Matematyk zapisze to tak:

$$even(n) \stackrel{df}{\Leftrightarrow} \exists z(n = z + z)$$
$$n\,\mathbf{div}2 = y \stackrel{df}{\Leftrightarrow} (n = y + y) \vee (n = y + y + 1)$$

Programista napisze algorytmy, na przykład tak:

```
unit div2: function(n: integer): integer;
  var i,j: integer
begin
  i:=0; j:=0;
  while i<n do
    i:=i+2; j:=j+1;
  od
  result := j;
end div2
```

```
unit even: function(n: integer): Boolean;
  var i,j: integer
begin
  i:=0; j:=0;
  while i<n do
    i:=i+2; j:=j+1;
  od
  result := j=n;
end div2
```

Obaj (mniej lub bardziej swiadomie) wykorzystują dwa twierdzenia teorii liczb

$$\forall n \exists z((n = z + z) \vee (n = z + z + 1))$$
$$\forall n, x, y(n = x + x \wedge n = y + y) \Rightarrow x = y$$

□

Zajmiemy się najpierw definicjami funkcji i relacji według wzoru znanego z podstaw matematyki.

Niech napis $\alpha(x_1, \ldots, x_m)$ będzie formułą otwartą (por. **??**), taką, że ciąg $x_1, \ldots, x_m$ zawiera wszystkie zmienne występujące w formule $\alpha(x_1, \ldots, x_m)$. Niech napis ρ_α będzie identyfikatorem, który nie występuje w formule α.

Definicja 10.2. *Napis zbudowany według następującego schematu jest deklaracją funkcji boolowskiej*

unit ρ_α: **function**$(x_1 : T_1, \ldots, x_m : T_m)$:Boolean
begin
result:= $\alpha(x_1, \ldots, x_m)$
end ρ_α

Przykład 10.3. *Funkcję charakterystyczną relacji* liczba x jest dzielnikiem liczby y *można zadeklarować w następujący sposób*

unit JestDzielnikiem: **function**(x,y: integer): Boolean;
begin
result := (y mod x = 0)
end JestDzielnikiem

i można wykorzystywać do budowania wyrażeń boolowskich, np.
(JestDzielnikiem(2,x) or JestDzielnikiem(2,3x+1)).

Niech napis $\tau(x_1, \ldots, x_m)$ będzie wyrażeniem całkowito-liczbowym (por. **??**), takim, że ciąg $x_1, \ldots, x_m$ zawiera wszystkie zmienne występujące w wyrażeniu $\tau(x_1, \ldots, x_m)$. Niech napis ϕ_τ będzie identyfikatorem, który nie występuje w formule τ.

Definicja 10.4. *Napis zbudowany według następującego schematu jest deklaracją funkcji typu integer*

unit ϕ_τ: **function**$(x_1 : T_1, \ldots, x_m : T_m)$:integer
begin
result:= $\tau(x_1, \ldots, x_m)$
end ϕ_τ

Gdzie można umieścić deklarację funkcji?

Deklaracja funkcji może wystąpić wśród deklaracji dowolnego modułu, a więc: programu, bloku, procedury, funkcji, klasy, współprogramu, procesu i modułu obsługi sygnału.

Nieistotność definicji

Twierdzenie o rozszerzaniu modelu poprzez przyjęcie zestawu definicji.
Niech $\mathcal{T} = \langle L, C, A\rangle$ będzie pewną teoria algorytmiczną określoną przez język L, operację syntaktycznej konsekwencji C i zbiór specyficznych dla tej teori aksjomatów A. Niech D_F będzie zestawem definicji. Dodatkowe założenie o definicjach ze zbioru D_F brzmi ... Przyjęcie takiego zestawu pozwala określić nową teorię $\mathcal{T} = \langle L', C', A'\rangle$, gdzie język L' ...

uzupełnić

Twierdzenie 10.1. *Każdy model $\mathcal{A}$ teorii $\mathcal{T}$ można rozszerzyć do modelu dla teorii $\mathcal{T}'$.*

Dowód. Dowód przebiega podobnie do dowodu w książce Rasiowej i Sikorskiego str. 322 □

Dyskusja.

Twierdzenie 10.2. *Teoria $\mathcal{T}'$ otrzymana z niesprzecznej teorii $\mathcal{T}$ przez przyjęcie zbioru definicji ... jest jej nieistotnym rozszerzeniem.*

$$\mathcal{C}(\mathcal{A}) = \mathcal{C}'(\mathcal{A}') \cap \mathcal{F}$$

Dowód. Należy udowodnić, że zbiór twierdzeń *starej* teorii $\mathcal{T}$ jest równy podzbiorowi zbioru twierdzeń *nowej* teorii $\mathcal{T}$, który zawiera tylko formuły $\mathcal{F}$ z języka starej teorii. Por. Ras-Sik str. 203 i str. 322 □

Jakie warunki mają byc spełnione by twierdzenie było prawdziwe. Lub inaczej, co się stanie gdy np. zrezygnujemy z wymagania by symbol ρ_α definiowanej funkcji nie występował w formule α.

Rozpatrzmy następującą deklarację

Przykład 10.5.

```
unit Fb: function(x: integer): Boolean;
begin
  result := ¬ Fb(x)
end Fb
```

Zauważmy, że nie istnieje funkcja Fb spełniająca równoważność $Fb(x) \Leftrightarrow \neg Fb(x)$. Przyjęcie takiej deklaracji powoduje przejście do teorii $\mathcal{T}'$ sprzecznej.

Kolejny przykład.

Rozpatrzmy następującą deklarację

Przykład 10.6.

```
unit Fc: function(x: integer): Boolean;
begin
  result := Fc(x)
end Fc
```

Dowolna funkcja $Fc(x)$ spełnia równowazność $Fc(x) \Leftrightarrow Fc(x)$.

Przykładów podobnych do tych dwóch, można podać bardzo wiele. Wnioski z nich wypływające są takie: 1° Przyjęcie deklaracji, które prowadzą do nowej sprzecznej teorii jest bezsensowne, jest stratą czasu. Zadbajmy o to, by zestaw deklaracji funkcji był niesprzeczny! 2° Zadbajmy też o to, by układ funkcji wyznaczony przez zestaw deklaracji był wyznaczony jednoznacznie. W przeciwnym wypadku stracimy czas na naprawianie takiego błędu w projektowaniu wymagań na oprogramowanie.

dlaczego? wyjaśnij!

przeczytaj str. 203 i 324 książki RS

Obliczalność

W odróżnieniu od rozważań przeprowadzanych w podręcznikach logiki matematycznej, musimy się zastanowić nad efektywnością deklarowanych funkcji. Nie wystarczy bowiem stwierdzić, że funkcja istnieje, należy wykazać, że przyjęcie deklarcji funkcji prowadzi od struktury obliczalnej $\mathfrak{A}$ do struktury obliczalnej $\mathfrak{A}'$. W związku z tym w definicjach funkcji boolowskich (tj. relacji) rodzaju pierwszego, będziemy wymagać by definiens był formułą postaci $M\,\gamma$. I rzeczywiście, w nieuwikłanych deklaracjach funkcji boolowskich możemy przyjąć, że sa one postaci

unit ρ: function(*params*): boolean;
$\mathbb{D}$
begin
$\mathbb{I}$
result := γ
end

objaśń

Co to znaczy nieuwikłane? Należy jeszcze wraz z taką deklaracją dołączyc dowód, że algorytm opisany instrukcjami $\mathbb{B}$ nie zapętli się i nie zerwie obliczenia. Tzn. wymagamy by jego obliczenia były skończone i udane.

Podobnie jeśli definicja funkcji jest uwikłana (znaczy to mniej więcej to samo co funkcja rekurencyjna (w slangu programistów) to wraz przyjęciem takiej definicji należy dołączyć dowód, że obliczenia będa udane i skończone. Jeśli tego nie zrobimy, to może okazać się, że nie istnieje wynik. A co zatem idzie nie istnieje model dla nowej teorii. Trzeba też udowodnić, że dla każdego zestawu argumentów istnieje conjwyżej jeden wynik. To jest znacznie łatwiejsze.

Deklaracje uwikłane

Niech napis $K\,\tau(x_1,\ldots,x_m)$ będzie wyrażeniem całkowito-liczbowym (por. **??**), takim, że ciąg $x_1,\ldots,x_m$ zawiera wszystkie zmienne występujące w wyrażeniu $K\,\tau(x_1,\ldots,x_m)$. Niech napis ϕ_τ będzie identyfikatorem, który nie występuje w formule τ. Do wprowadzenia definicji według poniższego schematu wymagane jest by formuła stopu programu K, np. $K\,true$ była twierdzeniem teorii $\mathcal{T}$ w której pracujemy.

wytłumacz!

Definicja 10.7. *Napis zbudowany według następującego schematu jest deklaracją funkcji typu integer*

unit ψ_τ: **function**$(x_1 : T_1,\ldots,x_m : T_m)$:integer
begin
K;
result:= τ
end ψ_τ

Zobaczmy parę przykładów

Przykład 10.8.

Kolejne pytanie jakie się pojawia, to czy tego rodzaju definicje pozwalają na istotne wzbogacenie mocy obliczeniowej?

Jawne i niejawne definicje

Definicje niejawne a rekursja

Jak się ma jedno do drugiego?

Definicje w których definiens jest termem algorytmicznym postaci $M\tau$ lub formułą algorytmiczną postaci $K\alpha$ gdzie α jest formułą otwartą.

Jeśli definicje są takie..., to rozszerzenie modelu $\mathcal{A}$ teorii $\mathcal{T}$ do nowego modelu jest obliczalne (o ile model $\mathcal{A}$ był obliczalny).

Rozdział 11

Procedury $\mathcal{L}_7$

Naturalnym wzbogaceniem(?) pojęcia bloku jest procedura. Koncepcja bloku jest przydatna, ale dość ograniczona. Instrukcja bloku jest instrukcją atomową, równoczesnie blok może mieć złożoną budowę. Wielu programistów uważa, że jest to ujęcie pewnego ciągu poleceń w nawiasy. I byłoby to poprawne, gdyby nie fakt, że blok może zawierac też deklaracje: zmiennych, procedur i funkcji, a nawet typów. Analiza semantyki instrukcji bloku wymaga przejścia od początkowej teorii $\mathcal{T}$ do nowej teorii $\mathcal{T}'$.

Deklaracja procedury jest definicją nowej instrukcji atomowej zdefiniowanej w programie. Instrukcja procedury ... Z wprowadzeniem procedur wiąże się wiele pytań i niebanalnych problemów. Problemy te powstają, gdy chcemy mieć coraz bardziej abstrakcyjne narzędzia.
Najpierw omówimy deklaracje procedur bezparametrowych i wykonywanie odpowiednich instrukcji procedury.
Potem omówimy protokół przekazywania parametrów aktualnych i odbierania wyników. Nasz pomysł: protokół wykonywania instrukcji procedury:

1. utwórz paczkę parametrów aktualnych (chciałbym powiedzieć obiekt - sygnał),
2. prześlij ją w poszukiwaniu procedury o nazwie P wzdłuż ścieżki SL,
3. po znalezieniu deklaracji procedury P utwórz blok (zmodyfikowaną treść procedury P) i wykonaj ten blok. Rekord aktywacji tego bloku ma strzałkę SL prowadzącą do rekordu aktywacji w którym znaleziono deklarację procedury P oraz strzałkę DL prowadzącą do rekordu aktywacji w którym wykonano instrukcję procedury.

Nie zapomnij!
Call P(args) jest protokołem współpracy rekordu aktywacji zawierającego instrukcję procedury i rekordu aktywacji procedury wywoływanej. Opis przesyłania parametrów i odbierania wyników: PODOBNIE JAK W CSP, jeden wysła drugi odbiera. Wykonanie zmodyfikowanego bloku treści procedury P poprzedzane jest przez ciąg par ...

Omówimy część z nich (tzn. tych problemów) na przykładzie procedury swap. STRUKTURA

- Przykład

- Składnia
- Semantyka
- Komputer
- Analiza przykładów

11.1 Przykłady procedur

Krótki przykład to procedura Swap. Przypomnij sobie program Swap i zobacz jak można wielokrotnie wykorzystać zdefiniowaną tam operację zamiany wartościami zmiennych x i y.

Przykład 11.1.

```
program sortuj3elementy;
  var a, b, c: integer;
  unit: Swap procedure(inout x,y: integer);
    var t: integer
  begin
    t :=x; x:=y; y := t;
  end Swap;
begin
  readln(a, b, c);
  if a > b then call Swap(a,b) fi;
  if b > c then call Swap(b,c) fi;
  if a > b then call Swap(a,b) fi;
  writeln("a= ",a,"b= ",b, "c=",c)
end
```

obmyśl

Drugi przykład ... mergesort? quicksort? search w BST?

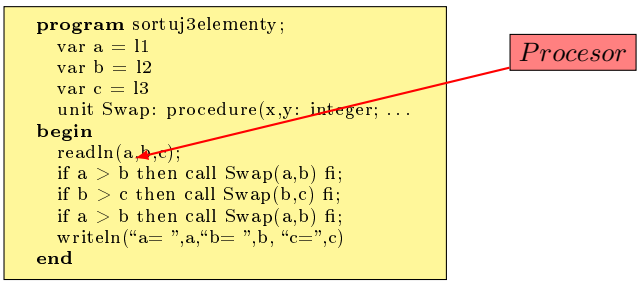
Trzeci przykład to procedura WHILE – tak! procedura równoważna instrukcji while. Pokazuje że można się obyć bez instrukcji while. Jeśli ktoś tak lubi ... Weź jakiś program z **while** i przerób go na procedurę. Np. bisection

Popatrzmy jak przebiega wykonanie tego programu, stan początkowy.

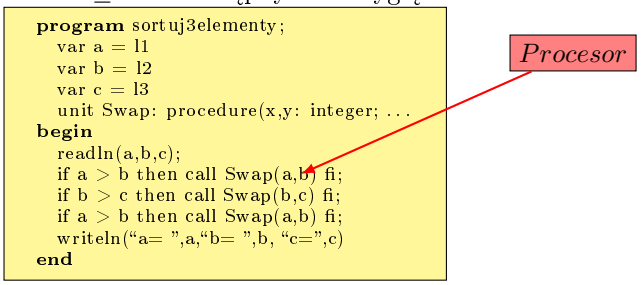
```
program sortuj3elementy;
  var a = 0
  var b = 0
  var c = 0
  unit Swap: procedure(x,y: integer;
  end swap;
begin
  readln(a,b,c);
  if a > b then call Swap(a,b) fi;
  if b > c then call Swap(b,c) fi;
  if a > b then call Swap(a,b) fi;
  writeln("a= ",a,"b= ",b, "c=",c)
end
```

Procesor

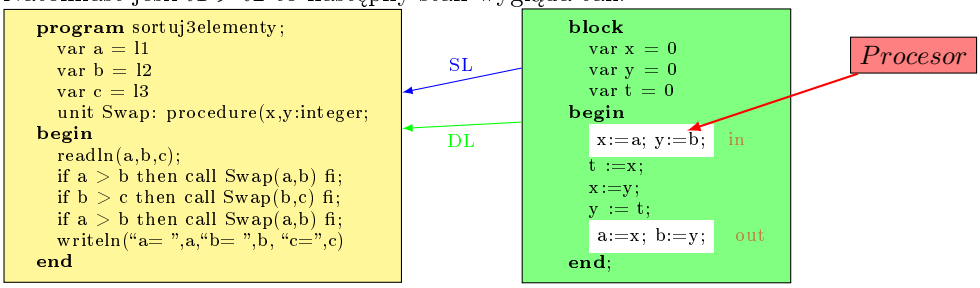
Następny stan, po wykonaniu polecenia readln(a,b,c). Oznaczmy wczytane wielkości przez, odpowiednio, l1, l2 i l3.



Jeśli $l1 \leq l2$ to następny stan wygląda tak.



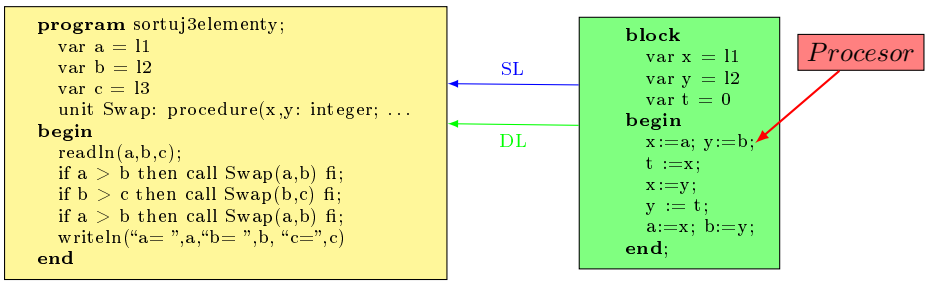
Natomiast jeśli $l1 > l2$ to następny stan wygląda tak.



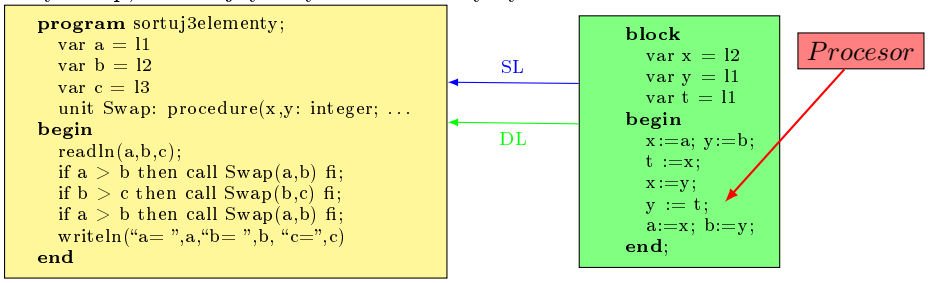
Pojawiła się nowa jednostka dynamiczna: *rekord aktywacji procedury* Swap. Utworzono ją według następujacej *reguły kopii*: Deklarację procedury przekształca się w instrukcję bloku. Parametry formalne stają się wielkościami (lokalnymi) zadeklarowanymi wewnątrz bloku. W treści bloku pojawiają się najpierw instrukcje przypisania, tak by parametrowi formalnemu f_i przypisac wartość parametry aktualnego a_i, o ile i-ty parametr formalny został zadeklarowany z modyfikatorem **in**. Potem w treści bloku pojawia się treśc procedury (tu trzy instrukcje). Potem pojawiaja sie instrukcje przypisania przekazujące wartości parametrów formalnych zadeklarowanych z modyfikatorem **out** odpowiednim parametrom aktualnym.

Sprawdź, że wykonanie instrukcji procedury **call** Swap(a,b) doprowadziło do utworzenia pokazanego na powyższym rysunku rekordu aktywacji procedury Swap.

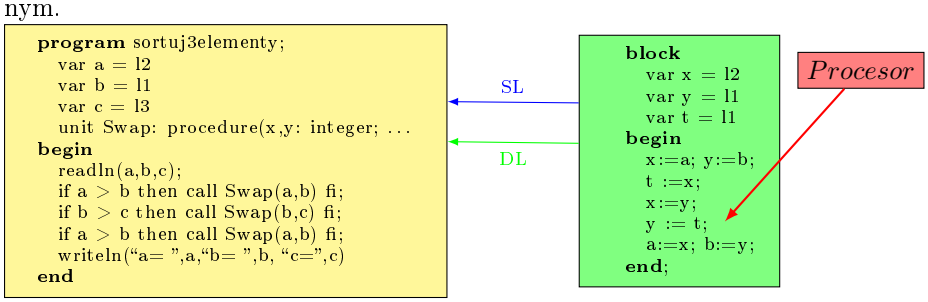
Efektem wykonania dwu poleceń x:=a; y:=b; przekazujących parametry aktualne parametrom formalnym jest stan pokazany na następujacym rysunku.



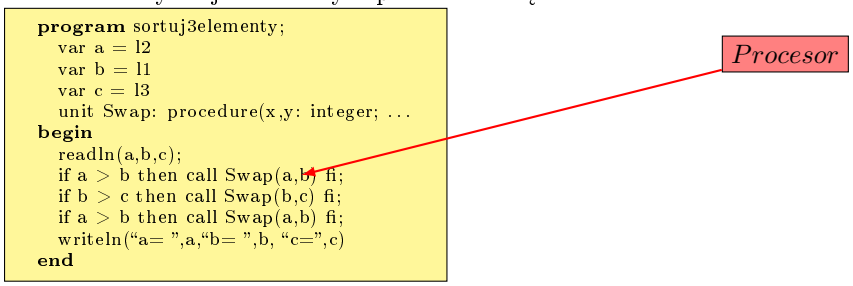
Po wykonaniu trzech kolejnych instrukcji przypisania zawartych w treści procedury Swap, na kolejnym rysunku zobaczymy taki obraz.



Kończymy wykonywanie procedury Swap. Przekazujemy wartości parametrów formalnych oznaczonych modyfikatorem **out** odpowiednim parametrom aktualnym.

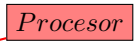


Usuwamy rekord aktywacji procedury Swap. Procesor wznawia działanie w rekordzie aktywacji wskazanym przez strzałkę DL.



Do tego miejsca można dojść dwoma różnymi drogami. Oto, jak zapisać wspólne cechy stanu jaki widzimy powyżej.

```
program sortuj3elementy;
  var a = min(l1,l2)
  var b = max(l1,l2)
  var c = l3
  unit Swap: procedure(x,y: integer; ...
begin
  readln(a,b,c);
  if a > b then call Swap(a,b) fi;
  if b > c then call Swap(b,c) fi;
  if a > b then call Swap(a,b) fi;
  writeln("a= ",a,"b= ",b, "c=",c)
end
```

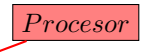


W podobny sposób przekonujemy się, że po wykonaniu drugiej instrukcji warunkowej if b >c ... fi zachodzi równość $c = max(max(a,b),c)$. Natomiast nie wiadomo czy $a < b$.

```
program sortuj3elementy;
  var a = min(l1,l2)
  var b = max(l1,l2)
  var c = max(max(l1,l2),c)
  unit Swap: procedure(x,y: integer; ...
begin
  readln(a,b,c);
  if a > b then call Swap(a,b) fi;
  if b > c then call Swap(b,c) fi;
  if a > b then call Swap(a,b) fi;
  writeln("a= ",a,"b= ",b, "c=",c)
end
```

Po wykonaniu trzeciej instrukcji warunkowej spełnione są dwie relacje $a \leq b$ i $c = max(max(l1, l2), l3)$. Wobec tego instrukcja writeln("a= ",a,"b= ",b, "c=",c) wydrukuje wartości a, b, c w porządku niemalejącym. Chcesz to sprawdź. Ale co dla Ciebie znaczy słowo sprawdź?

```
program sortuj3elementy;
  var a = min(l1,min(l2,l3))
  var b = max(min(l1,l2),min(l2,l3))
  var c = max(max(l1,l2),l3)
  unit Swap: procedure(x,y: integer; ...
begin
  readln(a,b,c);
  if a > b then call Swap(a,b) fi;
  if b > c then call Swap(b,c) fi;
  if a > b then call Swap(a,b) fi;
  writeln("a= ",a,"b= ",b, "c=",c)
end
```



Rozpatrzmy kolejny przykład

11.2 Składnia

Zaczynamy od procedur z parametrami in, out, inout. Bez procedur i typów jako parametrów.
Kłopoty z procedurami jako parametrami – problemy powierzchowne (Łatwiejsze) zapobiegać wywołaniu z parametrem P o nieprawidłowej liczbie argumentów, typach argumentów. Problemy trudniejsze – pojawiają się wtedy gdy autor procedury zakłada, ze parametr aktualny spełnia jakiś warunek, a uzytkownik (tj. autor instrukcji procedury) nie dopilnował by warunek ten został zapewniony.
WYMYŚL przykład.
Struktura deklaracji procedury jest podobna do struktury bloku.

Definicja 11.2. *Niech* $\Psi, \kappa_1, ..., \kappa_n$ *będą identyfikatorami,* $\mu_1, ..., \mu_n$ *będą słowami z trójelementowego zbioru* $\{\text{in}, \text{out}, \text{inout}\}$ ...

Deklaracja procedury Ψ ma następującą postać

$$\overbrace{\underbrace{\textbf{\textit{unit}}\ \Psi}^{nazwa}:\ \textbf{\textit{procedure}}\overbrace{(\mu_1\ \kappa_1 : T_1,\ \ldots,\mu_n\ \kappa_n : T_n)}^{lista\ parametrów\ formalnych};}^{nagłówek\ procedury}\underbrace{\mathbb{D}\ \textbf{\textit{begin}}\ \mathbb{I}\ \textbf{\textit{end}}\ \Psi}_{treść\ procedury}$$

Identyfikator Ψ jest *nazwą* zadeklarowanej procedury, Ciąg napisów $\mu_1\ \kappa_1 : T_1,\ \mu_n\ \kappa_n : T_n$ jest *listą parametrów formalnych* procedury Ψ. Identyfikator κ_i jest formalnym i-tym parametrem typu pierwotnego lub tablicowego T_i. Słowo μ_i okresla sposób przekazywania parametru aktualnego – objaśnimy to nieco dalej.
Lista deklaracji lokalnych $\mathbb{D}$, zarówno programu jak i procedury może zawierać deklaracje procedur i deklaracje zmiennych i stałych.
Struktura modułów programu. Zbiór bloków i procedur zadeklarowanych w programie wraz z relacją *decl* jest drzewem.

Przykład 11.3.

```
program Pr7;
  unit P2: procedure();
    unit Pw: procedure();
    begin
      call P3();
    end Pw;
  begin
    ...
    call Pw(0);
    ...
  end P2;
  ...
  unit P3: procedure();
  ...
  end P3;
begin
  ...
  call P2();
end
```

rys. graf moduły + decl

Ciąg instrukcji $\mathbb{I}$ może zawierać instrukcje znane nam wcześniej oraz instrukcje procedury. ...

obj.

Definicja 11.4. *Instrukcja procedury ma następującą budowę*

$$\textbf{\textit{call}}\ \Psi\ \underbrace{(\omega_1,\ldots,\omega_n)}_{lista\ par.aktualnych}$$

Każdy parametr aktualny ω_i ma być wyrażeniem typu T_i wymienionego w deklaracji procedury Ψ.
*Jeśli sposób przekazania i-tego parametru jest **out** lub **inout** to wyrażenie ω_i musi być zmienną (prostą lub indeksowaną) typu T_i.*

rys. moduły + call

Zbiór procedur zadeklarowanych w programie wraz z relacja *call* jest grafem wywołań.

11.3 Komputer $\mathcal{K}_7$

Komputer $\mathcal{K}_7$ zachowuje zdolność wykonywania poleceń znanych z wcześniejszych wersji abstrakcyjnego komputera $\mathcal{K}_6$. Nowa umiejętność to

Instrukcja procedury

Komputer oblicza parametry aktualne i składa je na stos. Następnie tworzy nową jednostkę dynamiczną tj. rekord aktywacji procedury i przenosi procesor do wykonywania instrukcji w tej jednostce. Return - instrukcja zakończenia obliczeń w rekordzie aktywacji procedury. Odesłanie obliczonych wartości do parametrów aktualnych określonych jako out lub inout.

11.4 Semantyka

Znaczenie instrukcji procedury nie jest oczywiste. Przez pewien czas przyjmowano, że instrukcja procedury da się wytłumaczyć przy pomocy tzw. reguły kopii, zobacz poniżej. W przypadkach bardziej złożonych ta reguła nie wystarcza. Ogólny przypadek instrukcji procedury objaśnimy opisując *protokół* współpracy jednostki dynamicznej zawierającej instrukcję procedury z rekordem aktywacji procedury. Reguła kopii – pierwsze przybliżenie.
Wykonanie instrukcji procedury jest równoważne wykonaniu instrukcji pewnego bloku skonstruowanego z parametrów aktualnych i treści procedury. Zamieniamy instrukcję procedury przez odpowiednio zbudowany blok. Najpierw przykład

Przykład 11.5. *Program zawiera deklarację pewnej procedury Psi i jedną lub więcej instrukcję procedury* ***call*** *Psi(...).*

```
program PrA;
  var x, y : integer, z : real;
  unit Psi : procedure(in a : integer, out b : real, inout c : integer);
    var u : integer;
  begin
  b := a + c; u := −11; c := a − u
  end Psi;
begin
  x := 7; z := x − 5;
  call Psi(x*4,y,z);
  writeln("z =", z, "y =", y)
end
```

Ten program jest równoważny programowi napisanemu poniżej

```
program PrAMod;
  var x, y : integer, z : real;
[   unit Psi : procedure( in a:integer,out b:real, inout c:integer );
      var u:integer;
    begin
      b:=a+c; u:=-11;  c:=a-u
    end Psi;                                                        ]
begin
  x := 7; z := x − 5;
{   block                                                           }  (∗ ≡ callPsi...∗)
      var a : integer, b : real, c : integer;
      var u : integer
    begin
      a := x ∗ 4; c := z;  (∗ pobranie parametrów in: a i c ∗)
      b := a + c; u := −11; c := a − u (∗ to jest treść Psi ∗)
      y := b; z := c     (∗odesłanie parametrów out: b i c, ∗)
    end
  writeln(“z =”, z, “y =”, y)
end
```

Kolejny przykład

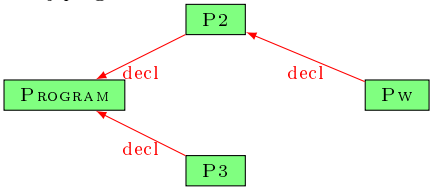
Przykład 11.6. *W tym przykładzie pokażemy, że może istnieć konflikt nazw: lokalna nazwa parametru formalnego i nielokalna zmienna występująca w parametrze aktualnym.*
Transmisja parametru aktualnego do parametru formalnego budzi w tym przypadku wątpliwości.
*a := (y+3)*a*
*Wartość wyrażenia (y+3)*a powinna być obliczona przed przekazaniem parametru do procedury. A przypisanie do parametru formalnego (wielkości lokalnej rekordu aktywacji procedury) już wewnątrz tego rekordu. Jak postąpić?*
*1. oblicz wartość parametru aktualnego i przypisz zmiennej pomocniczej (różnej od do tej pory zadeklarowanych zmiennych) np. aux := (y+3)*a,*
2. zmień nieco instrukcję procedury, np. call Psi(...,aux, ...)

W kolejnym przykładzie pokażemy, że strzałki DL i SL nie zawsze muszą mieć ten sam początek i koniec.

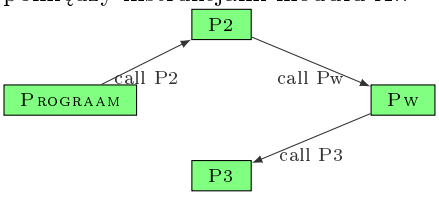
Przykład 11.7.

```
program Pr7;
  | unit P2: procedure();
  |   | unit Pw: procedure();
  |   | begin
  |   |   call P3();
  |   | end Pw;
  | begin
  |   ...
  |   call Pw(0);
  |   ...
  | end P2;
  ...
  | unit P3: procedure();
  | ...
  | end P3;
begin
  ...
  call P2();
end
```

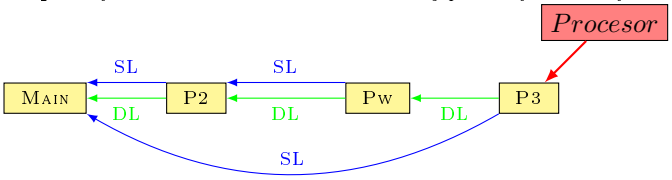
Statyczna struktura modułów tego programu jest widoczna na rysunku poniżej. Zielone prostokąty reprezentują procedury (lub bloki). Strzałka decl prowadzi od modułu A do modułu B gdy ten ostatni jest zadeklarowany w module B. W przypadku gdy moduł A jest blokiem, strzałka decl prowadzi do modułu zawierającego ten blok wśród swoich instrukcji.



Graf wywołań zawiera moduły programu. Strzałka call łączy moduł A z modułem B jeśli wśród instrukcji modułu A występuje instrukcja procedury call B(...) lub, w przypadku gdy moduł B jest blokiem, gdy blok ten występuje pomiędzy instrukcjami modułu A..



W pewnym momencie obliczeń istnieją cztery rekordy aktywacji.



Relacja decl pomiędzy modułami w statycznym grafie modułów daje podstawy do stworzenia relacji SL pomiędzy jednostkami dynamicznymi modułów.

Relacja call (graf wywołań procedur w Twoim obecnym stanie wiedzy o modułach) jest relacją odwrotną do relacji DL pomiędzy jednostkami dynamicznymi.

def. łańcucha dynamicznego

A oto reguła kopii. Instrukcja procedury **call** P(a,b,c) ma ten sam efekt co instrukcja bloku zbudowanego na podstawie treści deklaracji procedury P i postaci parametrów aktualnych a, b, c.

$$\textbf{call } P(a,b,c)\alpha \Leftrightarrow \left\{ \begin{array}{l} \textbf{block} \\ \text{oblicz wartości a, b i c,} \\ \text{złóż je na stos} \\ \textbf{block} \\ \text{treść tego bloku weź z deklaracji P;} \\ \text{modyfikując ją w ten sposób:} \\ \text{przypisz wartości ze stosu} \\ \text{a, b i c parametrom formalnym} \\ \textit{tu wstaw treść } P \\ \text{zakończenie – odebranie wyników} \\ \textbf{end} \\ \textbf{end} \end{array} \right\} \alpha$$

Problemy i pytania
Czy to musi być takie skomplikowane?
Zauważ, obliczenie wartośći parametrów ma miejsce tam gdzie jest instrukcja procedury, ale ciąg dalszy obliczeń ma miejsce w rekordzie aktywacji procedury P. Jeśli parametry aktualne są zmiennymi lub stałymi to reguła kopii jest prostsza:
call P(a, b, c) zastap przez {f1:=a;f2:=b; treść P; b:=f2;c:=f3 }
zakładam tu że w deklaracji procedury P, a jest in, b inout, c out. Jeśli w instrukcji procedury jest wyrażenie ω to instrukcję procedury **call** P(a, ω, c) zastap przez {z$\leftarrow \omega$; **call** P(a,z,c) }

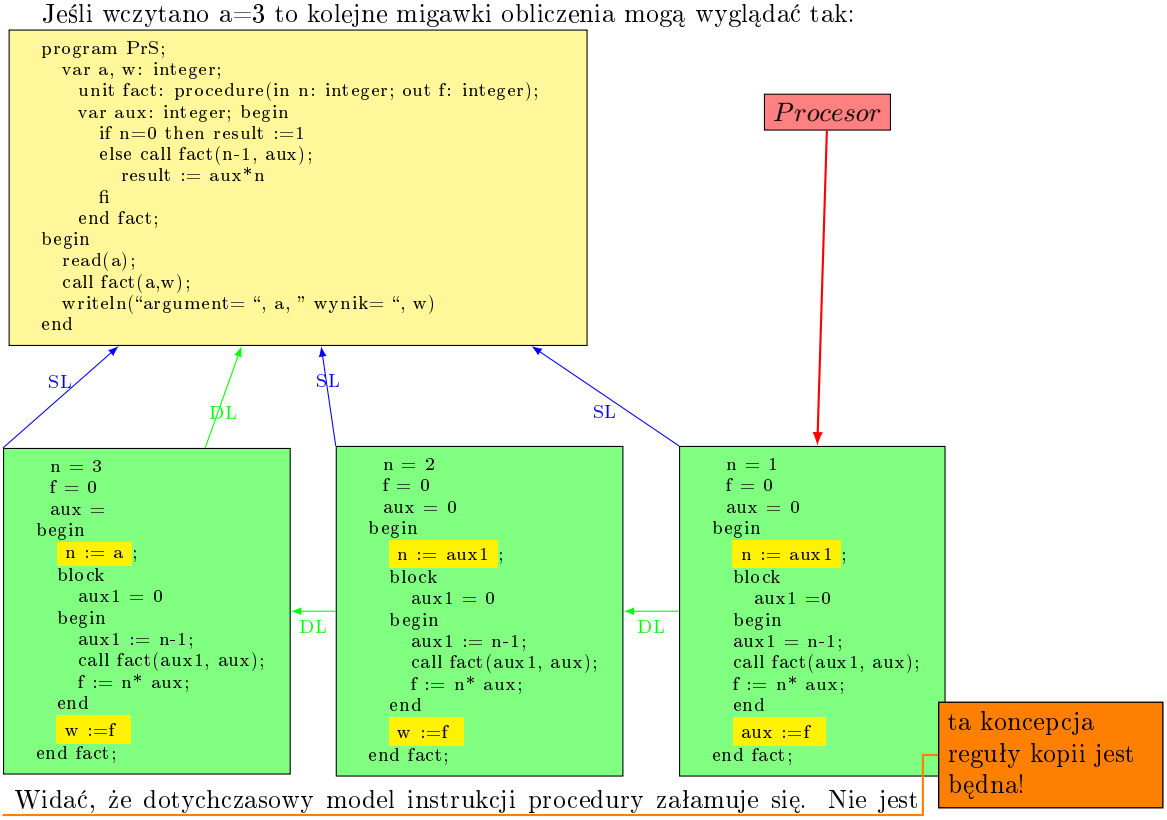
11.4.1 Procedury rekurencyjne

W tym miejscu przedstawimy dwa przykłady: procedurę rekurencyjną obliczania silni oraz procedurę ustawiającą 8 hetmanów na szachownicy.

Przykład 11.8.

```
program PrS;
  var a, w: integer;
  | unit fact: procedure(in n: integer; out f: integer);
  | var aux: integer; begin
  |   if n=0 then result :=1
  |   else call fact(n-1, aux); result := aux*n
  |   fi
  | end fact;
begin
  read(a);
  call fact(a,w);
  writeln("argument= ", a, " silnia= ", w)
end
```

Jeśli wczytano a=3 to kolejne migawki obliczenia mogą wyglądać tak:



Widać, że dotychczasowy model instrukcji procedury załamuje się. Nie jest poprawny. Jak to naprawić?

W kompilatorze instrukcja procedury jest realizowana według pwnego protokołu. Mówi się o *sekwencji wywołującej.*

instrukcja call P(a_1, ..., a_n) jest realizowana przez wykonanie ciągu następujących instrukcji

oblicz wartość parametru a_1 i włóż na stos

...

oblicz wartość parametru a_n i włóż na stos

przejdź do wykonywania (zmodyfikowanej) treści procedury P.

Po stronie procedury P:

utożsam stos z parametrami formalnymi (ten zabieg pozwala zaoszczędzić ciąg instrukcji

weź ze stosu i przypisz paramatrowi formalnemu f_{n-i} [dla i= 1, ..., n]

wykonaj treść procedury

powróć do jednostki dynamicznej wywołującej procedurę P.

odbierz wartości przekazane dla wyniku.

koniec protokołu Czytelnik zechce zauważyć, że parę zwrotów w powyższym protokole pozostało niejasnych. Twórcy kompilatorów wiedzą o co chodzi.

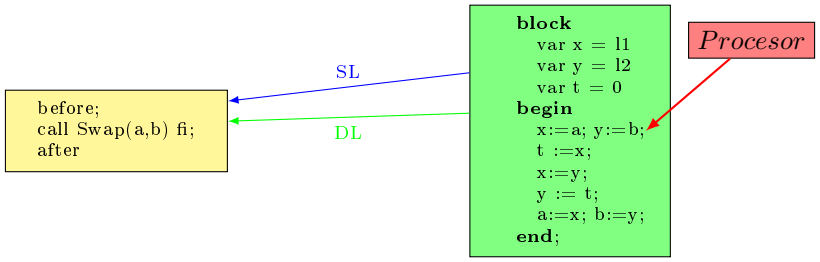
Jak należy rozumieć słowa *utożsamiamy stos wartości parametrów aktualnych z lista parametrów formalnych?*

Jak należy rozumieć zwrot *odbierz wartości parametrów formalnych oznaczone modyfikatorem out lub inout?*

11.4.2 Protokół call - realizacja instrukcji procedury

Następujący protokół realizuje polecenie call:

- W jednostce dynamicznej J wykonującej polecenie
  call P(a1, . . . , an)
  wystawiamy ciąg połowicznych instrukcji przypisania
  a1! . . . an!
  utwórz rekord aktywacji procedury P i przejdź do niego
- W rekordzie aktywacji procedury P:
  odbierz parametry aktualne f1? . . . fn?
  wykonaj instrukcje procedury P
  wyslij wyniki f1! ... fn!
  powróć do wykonywania poleceń w miejscu wystąpienia instrukcji procedury
- W jednostce dynamicznej J:
  odbierz wyniki a1? ... an?
  kontynuuj obliczenie (rekord aktywacji procedury P można i należy usunąć)



rysunek

11.4.3 Sprzeczność

przemyśl to!

W tym miejscu warto zwrócić uwagę na możliwość zadeklarowania procedur wzajemnie sprzecznych.

Część II

Programuj z klasą lub programowanie obiektowe

Rozdział 12

Moduły programu

Program jest zbiorem modułów zawartych w bloku programu głównego. Struktura programu może być bardzo skomplikowana. Jest to graf, którego węzłami są moduły. Pomiędzy modułami zachodzą dwie relacje:

- moduł M jest bezpośrednio zawarty w module K, lub inaczej mówiąc moduł M jest *zagnieżdżony* w module K, oznaczenie $M\ decl\ K$
- moduł M bezpośrednio dziedziczy z modułu N, lub inaczej mówiąc moduł M *rozszerza* moduł N, oznaczenie $M\ inh\ N$.

Program jest więc zbiorem $\mathcal{P}$ modułów z dwoma relacjami: $decl$ oraz inh.
Struktura $\langle \mathcal{P}, decl \rangle$ jest drzewem. Korzeniem drzewa jest moduł programu głównego (Main).
Struktura $\langle \mathcal{P}, inh \rangle$ jest lasem. [1]
A ponadto zachodzą następujące warunki:

- $inh(A) = bind(B\ \textbf{in}\ decl(A))$ – klasa A dziedziczy z tej klasy o nazwie B, która jest widoczna z klasy A, (zauważ, że w programie może znaleźć się wiele klas o tej samej nazwie). Może też zdarzyć się tak, że z miejsca w którym zadeklarowano klasę A nie widać żadnej klasy o nazwie B.
- żadna klasa nie dziedziczy samej siebie, bezpośrednio lub pośrednio.

to objaśnimy później

12.1 Rodzaje modułów

W programie mogą wystąpić moduły następujących rodzajów:

- blok, (*ang.* block)
- funkcja, (*ang.* function)
- procedura, (*ang.* procedure)

[1] Właściwie, należy to przyznać, każda klasa jest pochodną nieujawnionej klasy Obiekt. To w tej klasie znajdują się metody np. copy. To obiekty klasy Obiekt są argumentami struktury zarządzania pamięcią obiektów, np. kill. A więc, struktura $\langle \mathcal{P}, inh \rangle$ też jest drzewem.

- klasa, (*ang.* class)
- współprogram, (*ang.* coroutine)
- proces, (*ang.* process)
- moduł obsługi wyjątków (*ang.* handlers).

12.2 Role jakie odgrywają deklaracje modułów

Na moduł możemy spojrzeć w dwojaki sposób:

- moduł jest wzorcem według, którego tworzone są jednostki dynamiczne (instancje modułu). W trakcie wykonywania programu powstają i giną jednostki dynamiczne. Pomiędzy nimi zachodzą rozmaite relacje. Jedną z nich nazywamy zwykle SL jest ona dynamicznym odpowiednikiem relacji *decl* pomiędzy modułami. ...
- z metamatematycznego punktu widzenia moduły są definicjami. Np definicja funkcji definiuje funkcję i rozszerza w ten sposób alfabet dodając nowy funktor. Moduł klasy o nazwie C definiuje strukturę algebraiczną, nazwijmy ją $\mathfrak{C}$. Uniwersum tej struktury to zbiór obiektów tej klasy. Operacje i relacje są określone przez metody zadeklarowane w klasie.
  Przykład
  unit complex: class(Re,Im: real);...

12.3 Składnia modułu

W module możemy wyróżnić następujące składniki:

1. nazwę modułu N,
2. rodzaj modułu $\boldsymbol{K}$,
   $\boldsymbol{K} \in \{$block, function, procedure, class, coroutine, process, handlers $\}$,
3. listę parametrów formalnych *args*,
4. deklaracje lokalne *declarations*,
5. instrukcje *instructions*

Deklaracja modułu, na ogół, wygląda tak:

unit N: $\boldsymbol{K}$ (*args*);
  declarations
begin
  instructions
end N

12.4 Odstępstwa

Od powyżej naszkicowanej składni są trzy odstępstwa:

- moduł funkcji musi wymienić typ wyniku funkcji,
  po wyliczeniu parametrów formalnych, zob. przykład poniżej,
- moduł bloku nie ma nazwy ani listy parametrów formalnych,
  blok jest instrukcją,
- moduł **handlers** obsługi wyjątków ma trochę inną budowę,
  zajmiemy się tym później.

Jak zobaczymy później moduły mogą dziedziczyć z modułów klas. Składnia takiego modułu wygląda nieco inaczej. Omówimy to w trakcie wykładu o dziedziczeniu.

12.5 Przykład modułu procedury

```
unit swap: procedure (inout x,y: real);
  var pom: real
begin
  pom := x;
  x := y;
  y:= pom
end swap
```

Pewnie masz kilka pytań.

12.6 Przykład bloku

Blok zazwyczaj nie ma nazwy. Program główny jest blokiem. Program główny może zaczynać sie od słowa block lub od słowa program, w tym przypadku po słowie program występuje identyfikator, który nie ma żadnego znaczenia dla działania programu. To jest tylko nazwa.

```
program Bisekcja;
  var a, b, c, epsilon: real
begin
  while abs(b-a)>epsilon do
    c:=(a+b)/2;
    if f(a)*f(c)<0
    then  b:=c
    else a:=c
    fi
  od
end
```

```
block
  var a, b, c, epsilon: real
begin
  while abs(b-a)>epsilon do
    c:=(a+b)/2;
    if f(a)*f(c)<0
    then  b:=c
    else a:=c
    fi
  od
end
```

12.7 Blok - komentarze

Blok jest instrukcją. Możesz go wstawić wszędzie gdzie przewidziano miejsce dla instrukcji.
Program jest programem – najbardziej zewnetrznym modułem programu. Nie możesz go wstawić do innego programu.
Jeśli wolisz to blok może być programem. Nic na tym nie tracisz. Identyfikator po słowie program jest tylko nazwą programu. Nazwa ta nie jest wykorzystywana w celu innym niż identyfikacja tekstu programu.

12.8 Przykład modułu – funkcja

W module funkcji **MUSISZ** zadeklarować typ wyniku! (tu real)

```
unit sqrt: function(x:real): real ;
  var y,epsilon: real
begin
  while abs(x-y*y) > epsilon do
    y:=(y+x/y)/2;
  od;
  result := y;
  return
end sqrt;
```

Wynikiem jest wartość przypisana zmiennej systemowej result .
Uwaga! pamiętaj o typie wyniku.
Polecenie **return** pozwala zakończyć obliczenie w module funkcji lub procedury bez konieczności przechodzenia do ostatniej linii modułu
end sqrt.

12.9 Przykład procedury Bisection

12.10 Przykład modułu klasy

```
unit complex: class (Re, Im: real);
  var module: real;
  unit add: function(z:complex): complex;
  begin
    result:= new complex(Re+z.re, Im+z.Im)
  end add;
  unit mult: function(z:complex): complex;
  begin
    result:= new complex(Re*z.Re-Im*z.Im,Re*z.Im+Im*z.Re)
  end mult
begin
  module:=sqrt(Re*Re+Im*Im)
end complex;
```

Moduł klasy *complex* opisuje pewną strukturę algebraiczną.
Rozważmy następującą klasę

```
unit complex2: class (Mod, Arg: real);
  unit add: function(z:complex2): complex2;
  begin
    result:= new complex2(...)
  end add;
  unit mult: function(z:complex2): complex2;
  begin
    result:= new complex2(Mod*z.Mod, Arg+z.Arg)
  end mult
begin
end complex2;
```

Czy te dwie klasy definiują tę sama strukturę algebraiczną? Tak.
Czy potrafisz podać specyfikację tej struktury danych?
Jakie własności tej struktury danych sa niezbędne do dowodzenia własności algorytmów wykonywanych w tej strukturze?
Okazuje się, że oprócz aksjomatów ciała potrzebna jest nam własność: *byc ciałem charakterystyki zero*. Własność ta nie daje się wyrazić żadną formułą pierwszego rzędu. Ale jest ona wyrażalna formułą algorytmiczną

$$\neg\{x \leftarrow 1;\ \textbf{while}\ x \neq 0\ \textbf{do}\ x \leftarrow x+1\ \textbf{od}\}(x = 0)$$

12.10.1 Przykład zastosowania klasy complex

Pokazujemy przykład.

```
program ExampleComplex;
  unit complex: class ...
  var x,y,z,t,v:complex
begin
  z:=new complex(3.0, -4.5);
  t:=new complex(-21.9,21.9);
  x:=t.add(z.mult(new complex(3.2,4.3)));
  z:=x.mult(z);
end
```

Ile obiektów powstaje w trakcie wykonywania programu? Czy każdy obiekt jest wartością jakiejś zmiennej? Co z tym zrobić? Sformułuj swoje obserwacje podkreślając różnice pomiędzy obliczeniami na obiektach i obliczeniami na typie prostym real.

Start – rekord aktywacji Main

```
program ExampleComplex;
   complex: class ...
   complex u = none
   complex t = none
   complex x = none
begin
   u:= new complex(3.0, -4.5);
   t:= new complex(-21.9,21.9);
   x:= t.add(z.mult(new complex(3.2,4.3)));
   u:= x.mult(u);
end
```

obiekt u ∈ Complex

```
program ExampleComplex;
   complex: class ...
   complex u =
   complex t = none
   complex x = none
begin
   u:= new complex(3.0, -4.5);
   t:= new complex(-21.9,21.9);
   x:= t.add(u);
   u:= x.mult(u);
end
```

u=

```
real Re = 3.0
real Im = -4.5
add function ...
mult function ...
```

obiekt t ∈ Complex

```
program ExampleComplex;
   complex: class ...
   complex u =
   complex t =
   complex x = none
begin
   u:= new complex(3.0, -4.5)
   t:= new complex( 21.9,21.9)
   x:= t.add(u);
   u:= x.mult(u);
end
```

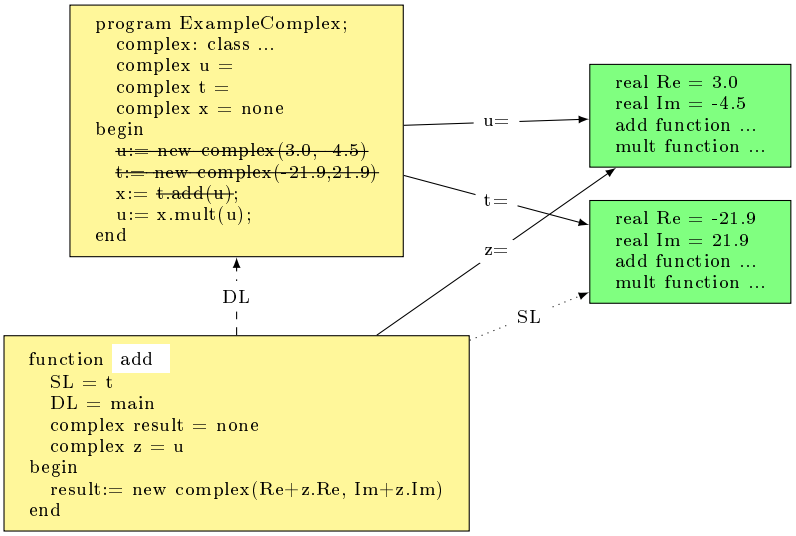
u=

```
real Re = 3.0
real Im = -4.5
add function ...
mult function ...
```

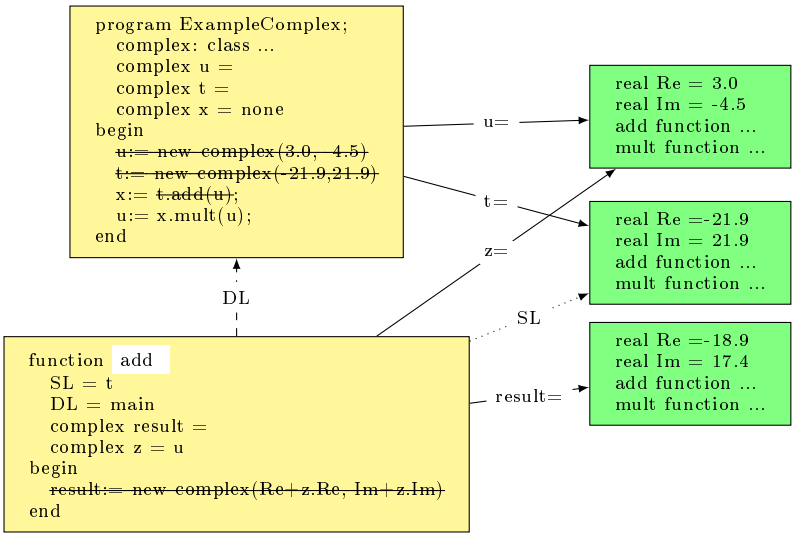
t=

```
real Re =-21.9
real Im = 21.9
add function ...
mult function ...
```

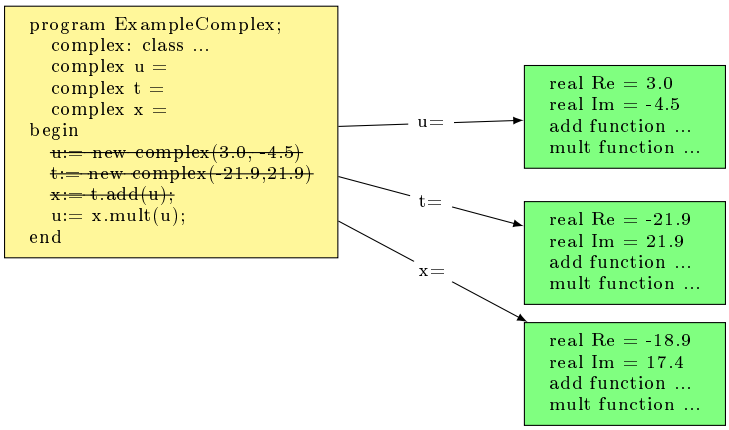
rekord aktywacji funkcji add



wynik funkcji add



x=u.add(t)



12.11 Przykład - klasa jako struktura danych

Klasy Punkt. Linia, Okrag zawarte w klasie GeometriaPlanarna.

```
unit Punkt: class(x,y:real);
   unit dist: function(p:Punkt):real; ...
   end dist; (* result = odleglosc do punktu p *)
end Punkt;
```

```
unit Linia: class(A,B,C:real);
   unit meets: function(l: Linia):Punkt; ...
   end meets; (* result= punkt przecięcia z linia l *)
   parallelTo: function(l:Linia): Boolean; ...
   end parallelTo; end Linia;
```

```
unit Okrag: class(s:Punkt, r:real);
   intersects: function(o:Okrag): Linia; ...
   end intersects; (* result= linia łącząca punkty przecięcia z okregiem o  *)
end Okrag;
```

12.11.1 klasa – GeometriaPlanarna

12.12 Przykład moduł współprogramu

```
unit Gracz: coroutine;
   var partner: Gracz;
begin
   ⟨ Inicjalizacja czyli konstruktor ⟩
   return;
   do
      ⟨ wykonaj ruch ⟩
      attach(partner);
   od
end Gracz
```

12.12.1 gra kółko i krzyżyk

```
program Kolko_i_krzyzyk;
   var Plansza: arrayof arrayof char;
   unit Gracz: coroutine ...
   var A,B: Gracz;
begin
   ⟨ Utworzenie planszy ⟩
   A := new Gracz;
   B:= new Gracz; B.partner:=A;
   A.partner := B;
   attach(A);
end
```

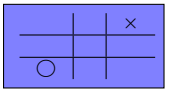
12.12.2 grają

początek programu – rekord aktywacji Main

```
program Kolko_i_krzyzyk;
  var Plansza: arrayof arrayof char;
  unit Gracz: coroutine ...
  var A,B: Gracz;
begin
  ⟨ Utworzenie planszy ⟩
  A := new Gracz;
  B:= new Gracz; B.partner:=A;
  A.partner := B;
  attach(A);
end
```

---

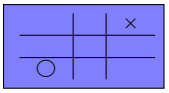
utworzenie pola gry



```
program Kolko_i_krzyzyk;
  var Plansza: arrayof arrayof char;
  unit Gracz: coroutine ...
  var A,B: Gracz;
begin
  ⟨ Utworzenie planszy ⟩
  A := new Gracz;
  B:= new Gracz; B.partner:=A;
  A.partner := B;
  attach(A);
end
```

---

tworzymy obiekt A współprogramu Gracz



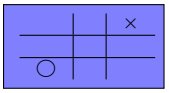
```
program Kolko_i_krzyzyk;
  var Plansza: arrayof arrayof char;
  unit Gracz: coroutine ...
  var A,B: Gracz;
begin
  ⟨ Utworzenie planszy ⟩
  A := new Gracz;
  B:= new Gracz; B.partner:=A;
  A.partner := B;
  attach(A);
end
```

A=

```
Gracz A:
do
   ruch;
   attach (B)
od
```

---

utworzenie obiektu B współprogramu Gracz



B=

```
Gracz B:
do
   ruch;
   attach (A)
od
```

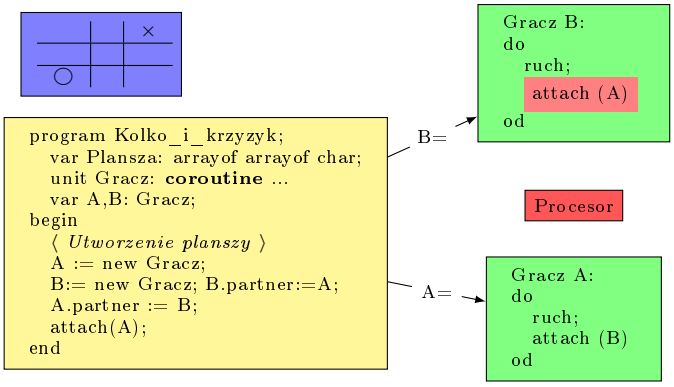
```
program Kolko_i_krzyzyk;
  var Plansza: arrayof arrayof char;
  unit Gracz: coroutine ...
  var A,B: Gracz;
begin
  ⟨ Utworzenie planszy ⟩
  A := new Gracz;
  B:= new Gracz; B.partner:=A;
  A.partner := B;
  attach(A);
end
```

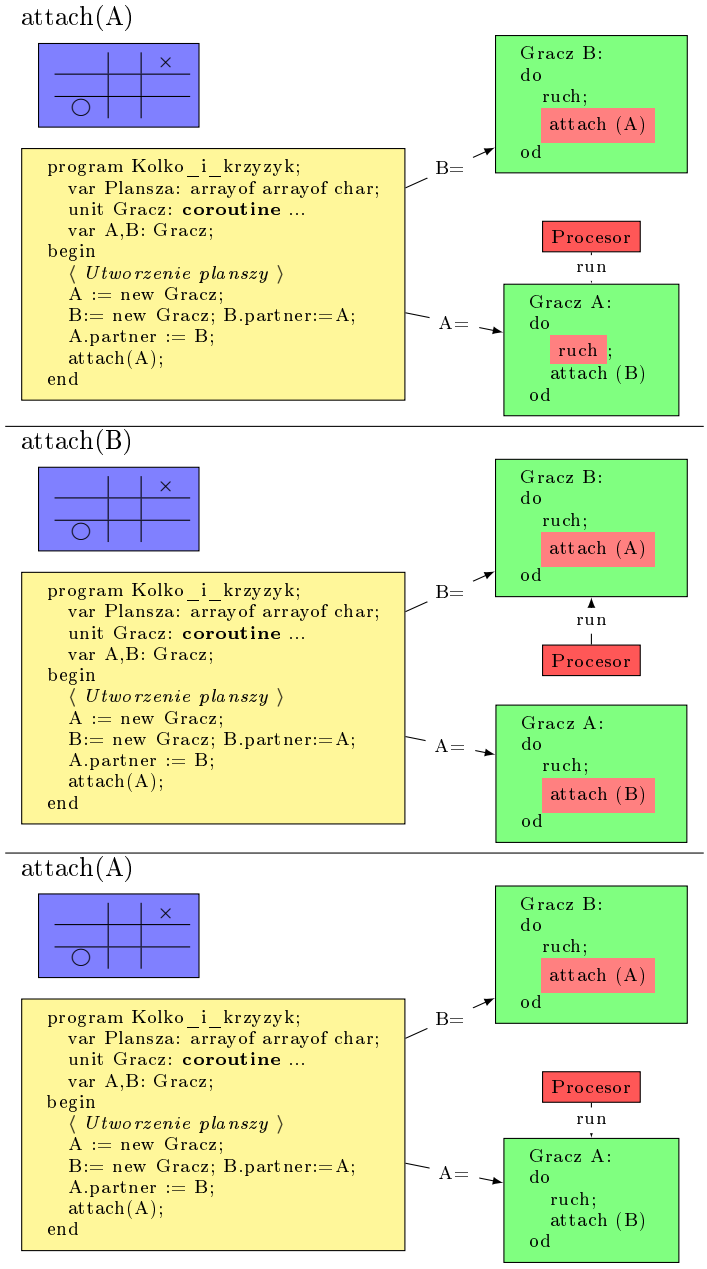
A=

```
Gracz A:
do
   ruch;
   attach (B)
od
```

---

utworzenie obrazka Procesor





12.13 Przykład modułu procesu

To jest tylko jeden prosty przykład procesu.

```
unit Agent: process( node : integer,...);
   unit meth: procedure( ); ... end meth;
begin
   ⟨ konstrukcja ⟩
   return;
   do
      accept meth
   od
end Agent
```

Omówić znaczenie parametru node, instrukcji accept.

Rozdział 13

Klasy i obiekty $\mathcal{L}_7$

W tej części zajmiemy się klasami i obiektami. Przedstawimy też dziedziczenie klas.

13.1 Klasa

Deklaracja klasy K jest definicją odpowiedniej struktury akgebraicznej S_K. W dalszym ciągu zobaczymy jak bardzo skomplikowane mogą być struktury opisywane przez klasy.

Składnia deklaracji klasy jest podobna do deklaracji procedury

unit K : **class**$(args)$; D **begin** I **end** K

Jak widać zamiast słowa **procedure** występuje słowo **class**.
Odpowiednio, zamiast instrukcji procedury **call**, w użyciu będzie słowo **new**. Wyrażenie new K(params) jest wyrażeniem obiektowym. Jego wartością jest pewien obiekt klasy K. Na rysunku dostrzeżesz jak wykonanie instrukcji przypisania x:= new K(params) skutkuje utworzeniem obiektu o (klasy K) i przypisaniem go jako wartości zmiennej x.

rys. obiekt

Klasy stanowią budulec z którego tworzymy programy obiektowe. W danym programie klasa jest niezmienna.
Klasa wyznacza nowy typ. Czyli strukturę algebraiczną. Taka struktura może być jednak bardziej skomplikowana od struktur badanych przez matematyków. Co prawda, klasa complex definiuje ciało liczb zespolonych. Ale moduł klasy pozwala zdefiniować bardzo skomplikowane struktury algebraiczne.
Klasa C umożliwia deklarowanie zmiennych typu C:
var z1,z2: C

deklaracja klasy to dodanie nowego typu, tzn. nowej struktury algebraicznej i jej teorii!

Niezmiennik systemu Loglan. *W trakcie obliczeń programu wartość każdej zmiennej jest albo wartością pewnego typu prostego albo obiektem pewnej klasy (lub tablicy) albo jest specjalną stałą* ***none****.*

Typ wyznaczony przez deklarację klasy C jest to zbiór wszystkich obiektów jakie spełniają relację **in** pomiędzy obiektami i klasą, wraz z operacjami na obiektach tego typu i relacjami pomiędzy obiektami tego typu.
Przykład

Zbiór $|complex|$ obiektów klasy complex z działaniami add i mult opisanymi w tej klasie.

$$\mathcal{C} = \langle |complex|, add, mult, = \rangle$$

stanowi algebrę, którą oznaczyliśmy $\mathcal{C}$.
No tak, ale czym są obiekty klasy complex?

13.2 Obiekty

Obiekty klas są jednostkami dynamicznymi – tzn. powstają w trakcie wykonywania programu. Po jego zakończeniu znikają.
Obiekty są wartościami zmiennych zadeklarowanych jako zmienne odpowiedniego typu opisanego klasą. Obiekt może zmieniać się w trakcie obliczeń. Jest jednostką dynamiczną.

13.3 Scenariusz obiektu

Obiekty:

- powstają - w efekcie obliczenia wartości wyrażenia `new C(params)`, i stają się wartośćia zmiennej odpowiedniego typu, por. `x:=new C(params)`
- są współdzielone przez różne zmienne – w efekcie wykonania instrukcji przypisania `y:=x`,
- są dostępne dla operacji:
  - odczytu wartości atrybutu,
  - zapisu (modyfikacji) wartości atrybutu,
  - usługi (serwisu),
- stają się niedostępne (stają się "śmieciami") gdy nie wskazuje na nie żadna zmienna,
- są usuwane np. przez polecenie `kill(x)`

Obiekt jako argument operacji.
Obiekt jako posiadacz stanu pamięci.
Przykłady
zmiana promienia okręgu
wstawienie elementu do kolejki

13.4 Uwagi o odśmiecaniu i defragmentacji

Odśmiecanie jest kosztownym procesem. W fazie "sweep" trzeba przejrzeć całą pamięć. Następna faza też jest kosztowna.
Czy nie można by zmniejszyc potrzeby odśmiecania przez częstsze kompresowanie wolnej pamięci? Wystarczyłoby by porzucając wykorzystany obiekt pomocniczy protokół ... wpisywał ten obiekt na listę wolnej pamięci. Jak to

zrobić?

Odśmiecanie nie jest koniecznym elementem systemu zarządzania obiektami! Jeśli pamiętasz nazwę obiektu, który za chwilę stanie się smieciem, to zrób kill!

Klasa GaussC – liczb zespolonych całkowitych

Karl Gauss rozważał zbiór liczb zespolonych całkowitych.

```
unit GaussC: class(re, im: integer);
  unit add: function(z: GaussC): GaussC;
  begin
    result:= new GaussC(re+z.re, im+z.im)
  end add;
  unit mult: function(z: GaussC): GaussC;
  begin
    result:= new GaussC(re*z.re-im*z.im, im*z.re+z.im*re)
  end mult;
end GaussC;
```

Struktura na którą składa się zbiór $|GaussC|$ obiektów tej klasy z działaniami *add* i *mult*

$$\mathfrak{C}_I = \langle |GaussC|, add, mult \rangle$$

Zadanie 13.1. *Sprawdź, że struktura $\mathfrak{C}_I$ jest pierścieniem.*

Popatrzmy na program wykorzystujący klasę GaussC.

```
program EuGaussC;
  unit GaussC: class ...
  unit norm: function(z: GaussC): real;
  begin
    result:=z.re*z.re+z.im*z.im
  end norm;
  unit div: function(y, z: GaussC): GaussC;
  begin
    result:=
  end div;
  unit rem: function(y,z: gaussC): GaussC;
  begin
    result:=
  end rem;
  unit equal: function(y: gaussC): Boolean;
  begin
    result:= re=y.re and im=y.im
  end rem;
  var z1, z2, z3, zero: GaussC;
begin
  zero := new GaussC(0,0);
  z1:= new GaussC();
  z2 := new GaussC();
  while not rem(z1,z2).equal(zero) do
    xyz ... XYZ
  od ;
end
```

Łatwo napisać klasę complex. Jakie własności ma opisana (zrealizowana) przez tę klasę algebra?
Operacje Add i Mult. OK.
Czy są łączne? przemienne? ...
Czy to jest ciało?
Czy jest to ciało Archimedesowe? Jaki porządek?
Liczby zespolone całkowite Gaussa. To jest pierścień. W tym pierścieniu algorytm Euklidesa ma zawsze obliczenie skończone!
Jesli wprowadzę porządek leksykograficzny ...
to będzie to ciało Archimedesowskie (?) NIE
porządek leksykograficzny (ani żaden inny porządek $<$) nie zapewnia własności takich jak
x$<$y implikuje dla każdego a x+a $<$y+a.

Mnożenie macierzy liczb zespolonych

Osiem różnych wersji
mnożenie zwykłe lub sprytne liczb zespolonych
mnożenie macierzy zwykłe lub Winograda
reprezentacja danych: macierze obiektów complex lub para macierzy Re oraz Im

Klasa GeometriaPlanarna

Język programowania zorientowany do rozwiązywania problemów geometrii cyrkla i linijki.

Zadanie Okrąg opisany na trójkącie

Zadanie Inwersja względem okręgu

Zadanie

Teza Algorytm geometrii cyrkla i linijki, który się nie zapętla nie potrzebuje instrukcji while.

odszukaj gdzie to jest napisane

Kolejki

Każdy spotkał się ze strukturą kolejek. Jaka to algebra?
Specyfikacja kolejek – niesprzeczność, zupełność.

13.5 Stosy

opisać stosy w tablicach, stosy-listy, stosy w N

aksjomaty stosów

13.6 Komputer $\mathcal{K}_7$

Komputer $\mathcal{K}_7$ tworzy obiekty, umożliwia dostęp do ich atrybutów, usuwa obiekty (kill).

13.6.1 Struktura danych do zarządzania obiektami

Obiekty są wartościami zmiennych (odpowiednich typów). Komputer $\mathcal{K}_7$ zapewnia przy tym zachowanie następującej własności Jeśli zmienna z jest zadeklarowana jako typu T (np. unit T; class ...) to jej wartością jest obiekt klasy T lub wartością jest **none**.

13.6.2 lokalizacja potrzebnej zmiennej ...

Komputer $\mathcal{K}_7$ umie znaleźć zmienną, która jest potrzebna do obliczenia wartości wyrażenia lub do wykonania instrukcji przypisania. Problem pojawia się gdy szukana zmienna nie jest zmienną lokalna.

13.7 obietnice ...

It turns out that the combination of inheritance and inner classes offers many interesting possibilities:
• it allows to obtain most of the effects of multiple inheritance c.f. [19, Chapter 10],
• instead of passing classes as parameters one can extend abstract inner classes which serve as counterparts of formal parameters [20 p. 176],
• provides a convenient way to express call back objects [19],
• allows to inherit certain patterns of architecture, e.g. a class pattern of the

model-view-controller system can be defined and extended by inheriting classes [2,19],
- allows to inherit protocols [2 p. 112–113],
- enables inheritance of a class put earlier into a tree-like library of classes,
- and many others

Rozdział 14

Dziedziczenie klas $\mathcal{L}_8$

Dziedziczenie klas zwane też rozszerzaniem (*ang.*extending) jest narzędziem potężnym, słabo rozumianym i rzadko w pełni wykorzystywanym. Deklaracja klasy wprowadza nowy typ danych, jest jego definicją. Dziedziczenie czyli rozszerzanie definicji klasy pozwala tworzyć hierarchie typów. Dwie deklaracje klas

$$\textbf{unit } A : \textbf{ class}(args_A);\ D_A \textbf{ begin } I_A \textbf{ end } A$$

oraz

$$\textbf{unit } B : \textbf{ class}(args_B);\ D_B \textbf{ begin } I_B \textbf{ end } B$$

mogą znaleźć się w relacji dziedziczenia, w taki oto sposób

$$\underbrace{\textbf{unit } B : A \textbf{ class}}_{klasa\ B \quad rozszerza\ A}\ (args_B);\ D_B \textbf{ begin } I_B \textbf{ end } B$$

W takiej sytuacji mówimy, że klasa B dziedziczy z klasy A lub klasa B *rozszerza* klasę A. W największym skrócie i z wieloma zastrzeżeniami, należy rozumieć, że klasa B jest faktycznie zadeklarowana w taki sposób

$$\textbf{unit } B : \textbf{ class}(args_A, args_B);\ D_A;\ D_B \textbf{ begin } I_A;\ I_B \textbf{ end } B$$

Mówimy, że relacja rozszerzania klas jest opisana przez *regułę konkatenacji* klas. Deklaracja klasy B jest wynikiem konkatenacji deklaracji klasy A i deklaracji klasy B. Przypominamy, podana powyżej wersja reguły konkatenacji klas jest znacznie uproszczona w stosunku do jej pełnej wersji.

14.1 Przykłady

hierarchia klas: przykład klasa rachunek

14.2 Wyznaczanie klasy dziedziczonej

14.3 Reguła konkatenacji klas

Regułę konkatenacji klas objaśnimy w kilku odsłonach pokazując kolejne szczegóły tej reguły.

14.3.1 Składanie deklaracji - warstwy

14.3.2 inner

14.3.3 metody wirtualne

14.3.4 qua -

14.3.5 blok prefiksowany

Przykład – blok wyznaczający środek okręgu opisanego na trójkącie.
pref GeoPlan block ... end
Przykład – algorytm mnożenia macierzy => blok A
Mnożenie macierzy w pierścieniach egzotycznych i zwykłych
Klasa *Complex* ...
Klasa *Egzo*1 add= minimum , mult= dodawanie
Klasa *Egzo*2 add= maximum , mult= minimum
Instrukcje bloku prefiksowanego w różnych otoczeniach
pref Complex **block** A **end** – mnożenie macierzy liczb zespolonych
pref Egzo1 **block** A **end** – najkrótsza droga od punktu do punktu
pref Egzo2 **block** A **end** – droga o przepustowości ...
W przypadku Egzo1 macierz zawiera informacje o długości drogi z punktu i do punktu j.
W przypadku Egzo2 macierz zawiera informację a_{ij}= wysokość wiaduktu na drodze od punktu i do punktu j.
Problem *jak w bloku A zadeklarować zmienną*? Jakiego ma być ona typu?
Należałoby powiedzieć, w bloku A, zmienna s jest typu opisanego w prefiksie.
Ale co to konkretnie znaczy?
A jeśli prefiks to GeoPlan to, który z typów zawartych w w tej klasie Okrąg, Linia, Punkt?
Jak to powiązać? by zapewnic pewna uniwersalność bloku A?

Może tak:
pref B(mójTyp) block A end
w deklaracji klasy B będzie parametr formalny T, w bloku A moge napisać **var** s : T.
A w nowym języku LEM można będzie podstawiac na miejsce T dowolny podtyp typu T. Wtedy wyrażenia s.add() i s.mult(...) będą mieć sens.

14.4 Relacja in

Parę słów o algorytmicznej wersji teorii mnogośći.

14.5 Struktura modułów

Wcześniej powiedzieliśmy, że struktura modułów każdego poprawnego programu jest grafem z dwoma relacjami: *decl* oraz *inh*. Wierzchołkami grafu są moduły programu. Relacja *decl* zachodzi pomiędzy modułem m i m' gdy moduł m

jest zadeklarowany bezpośrednio w module m'. *decl* jest funkcją okresloną dla każdego modułu oprócz programu głównego.

$$decl\colon \{Mod \setminus main\} \to Mod$$

A więc zbiór Mod modułów programu wraz z funkcją *decl* jest drzewem skończonym. Podobnie relacja *inh* jest funkcją określoną dla wszystkich modułów programu oprócz *main* i *Object*.
Jest to więc inne drzewo skończone z tym samym zbiorem wierzchołków.

Oprócz tych dwu funkcji okreslone sa tez dwie relacje: $call\colon Mod \to Mod$, dwa moduły znajduja się w relacji *call* gdy w module m istnieje instrukcja $callm'(...)$ wywołania metody ...
Funkcja $bind(occur - id, m) = m'$ wyznacza dla danego wystąpienia $occur - id$ identyfikatora id w module m taki moduł m', że ...
Funkcja *type* dla danego wystąpienia wyrażenia ω w module m wyznacza typ tego wyrażenia, a więc klasę definiująca ten typ.
Podsumowując algebraiczna struktura modułów programu jest grafem:

- Zbiór Mod modułów jest skończony. Sa to moduły wypisane w programie bezpośrednio lub zaimportowane z biblioteki klas.
- Funkcja *decl* ...
- Funkcja *inh* ...
- Funkcja *bind* ...
- Funkcja *type* ...
- Relacja *call* ...

Zachodzą następujące związki *(z pracy LSW)* wypisz aksjomaty

Definicja 14.1. *(A1) (baza indukcji 1). Dla każdej klasy K znaczeniem typu pustego ϵ jest Object.*

$$bind(\epsilon\ in\ K) \stackrel{df}{=} Object$$

(A2) (baza indukcji 2). Niech K będzie klasą. Znaczeniem identyfikatora (klasy) C jest klasa o nazwie C taka, że

$$bind(CinK) \stackrel{df}{=} (inh^i decl^j(K)).C$$

gdzie para $(j,i), j \geq 0, i \geq 0$, jest najmniejsza parą , w porzadku lekskograficznym par, taką, że klasa $(inh^i decl^j(K)).C$ jest określona. Pary liczb naturalnych sa porównywane zgodnie z porządkiem leksykograficznym tzn. że para (j,i) jest mniejsza niż para (q,p) jeśli $j < q$ lub $j = q\ i\ i < p$. Wartość wyrażenia $bind(C\ in\ K)$ jest nieokreślona w pozostałych przypadkach.
(B) (krok indukcyjny). Niech $X \neq \epsilon$. Dla każdej klasy K znaczenie typu postaci $X.C$ w klasie K jest wyznaczone w ten sposób

$$bind(X.C\ in\ K) \stackrel{df}{=} (inh^i(bind(X\ in\ K)).C$$

gdzie $i \geq 0$, jest najmniejszą liczbą naturalną taką, że klasa $(inh^i(bind(XinK)).C$ jest określona, znaczenie wyrażenia $bind(X.C\ in\ K)$ jest nieokreślone w pozostałych przypadkach.

Definicja 14.2. *Relacja zależenia dep*

$dep \stackrel{df}{=} K, bind(ext(K)|^i \ in \ decl(K)) : K \in Classes, 0 < i \geq length(ext(K)) forext(K) \neq \epsilon i = 0 forext($

14.6 Applications

It turns out that the combination of inheritance and inner classes offers many interesting possibilities:
• it allows to obtain most of the effects of multiple inheritance c.f. [19, Chapter 10],
• instead of passing classes as parameters one can extend abstract inner classes which serve as counterparts of formal parameters [20 p. 176],
• provides a convenient way to express call back objects [19],
• allows to inherit certain patterns of architecture, e.g. a class pattern of the model-view-controller system can be defined and extended by inheriting classes [2,19],
• allows to inherit protocols [2 p. 112–113],
• enables inheritance of a class put earlier into a tree-like library of classes,
• and many others

14.7 przyczynek do teorii

Wczesniej widzieliśmy, że deklaracje funkcji sa niczym innym niż definicjami.

Przyglądając się deklaracjom klas stwierdzamy, że są to definicje struktur algebraicznych, często będziemy mówić definicje struktur danych. W podręcznikach podstaw matematyki nie znaleźliśmy twierdzeń, które odpowiadały by na nasze pytania.

Uwaga.
Następująca deklaracja

```
unit T: class;
  var l,r: T;
end T;
```

może być pojmowana jako definicja typu drzewo binarne. Zob. []
Czy rzeczywiście? Czy istnieją niestandardowe modele? Modele czego? Czy jest to definicja kompletna?
Co zyskujemy powiadając "*najmniejszy model w kategorii wszystkich modeli powyżej definicji jest strukturą drzew binarnych.*"

Rozdział 15

Stosy

Ten rozdział poświęcamy strukturze stosów. Dwa są powody by obszerniej zająć się tematem stosów:

- ponieważ struktura ta ma wiele zastosowań, i
- ponieważ na przykładzie struktury stosów przedstawimy problemy wiążące się z specyfikowaniem klas i z przeprowadzaniem dowodów.

15.1 Specyfikacja stosów

Czym są stosy? Jak Twoim zdaniem powinna brzmieć odpowiedź na to pytanie?

Na ogół nie zawracamy sobie głowy tym problemem. W dawnej Polsce w encyklopedii o tytule "Nowe Ateny" pod hasłem koń czytamy *koń jaki jest każdy widzi*. I słusznie. W tamtych czasach, jeśli ktoś umiał czytać, to na pewno widział konia.

Spróbuj jednak zapisać definicję stosu. A przynajmniej spróbuj napisać wymagania W na to by klasa K mogła być uznana za klasę definiująca stosy liczb całkowitych.

Aha, nie jest to oczywiste. Programujący w Javie napiszą interfejs

```
interface StosI {
  void push(int e)
  void pop()
  int top()
}
```

lub coś podobnego i powiedzą Stos ma trzy metody: push, pop i top. Porównaj []. Świetnie.

Czy następująca deklaracja klasy jest poprawną implementacją tego interfejsu?

```
class Stos implements StosI {
  private int[8] tabl;
  private int i,j;
  void push(int e) {tabl[j++] = e }
  void pop() {i++ }
  int top() {return tabl[i]}
}
```

Czy powyższa klasa Stos implementuje interfejs StosI? Trzeba uznać, że tak,

ponieważ zawiera deklaracje trzech metod push, pop i top, a ponadto typy argumentów i wyników są zgodne dla każdej z tych metod. Kompilator Javy zaakceptuje słowo zdanie "class Stos implements (interface) StosI". Czy jednak jest to stos? Hmm! Przecież ta klasa implementuje kolejki FIFO! Czegoś zabrakło w naszej specyfikacji.
Zgadzamy się co do tego, że stosy muszą wykonywać trzy działania:

$push$: $int \times Stos \to Stos$

pop: $Stos \to Stos$

top: $Stos \to int$

Ale takie wyliczenie to za mało.
Powinniśmy jeszcze opisać efekty działania tych operacji. Tak jak to zrobiliśmy w rozdziale 3 Wyrażenia, opisując własności działań na typach prostych. Pamietasz?
Co więc należy podać jako własnośći (aksjomaty?) stosów?

$$top(push(e, s)) = e$$

Powyższą formułę czytamy: *operacja top zastosowana do wyniku operacji* $push(e, s)$ *zwraca element* e. Możemy ją poprzedzić kwantyfikatorem ogólnym.
Mamy więc pierwszy aksjomat stosów

$$\forall_s \forall_e \, top(push(e, s)) = e$$

Można tę formułę napisać w ortografii programowania obiektowego w taki sposób:

$$\forall_s \forall_e \, top(s.push(e)) = e$$

Dodajmy jeszcze podobną formułę

$$\forall_s \forall_e \, pop(push(e, s)) = s$$

oraz kolejne

$$empty(newStos)$$

$$\forall_s \forall_e \neg empty(push(e, s))$$

Mamy już cztery aksjomaty. Czy to wystarczy?

Most of us knows what stack is. At least players of the game canasta know. Any programmer used stacks at least once in his professional life. The shortest description is LIFO. Elements are put into stack and extracted. The LIFO means: Last In First Out.

Na chwilę odejdźmy od założenia, że elementami stosu są liczby naturalne. Przyjmijmy, że dana jest pewna klasa Elem i to właśnie obiekty tej klasy są elementami stosów. Operacja $push$ wkłada element e do stosu s a jej wynikiem jest stos $push(e, s)$. Inna operacja pop zdejmuje element ze stosu i zwraca stos $pop(s)$. Operacja top zwraca element $top(s)$. Dwie ostatnie operacje nie są zdefiniowane gdy argument s jest stosem pustym, tj. gdy zachodzi $empty(s)$.
Możemy to podsumować w ten sposób: struktura algebraiczna stosów ma swoje uniwersum na które składa się unia dwu zbiorów, zbioru elementów E i zbioru stosów S. Ponadto mamy trzy operacje: $push, pop, top$ oraz dwa predykaty $empty$ i równość $=$.

Tablica 15.1. Specyfikacja S1 struktury stosów

Sygnatura czyli interfejs	Komentarze
Uniwersum $= E \cup S$	
E	zbiór elementów
S	zbiór stosów
Operacje	
$push : E \times S \longrightarrow S$	włóż element e do stosu s
$pop : S \longrightarrow S$	zdejm wierzchołek stosu
wynik jest określony	wttw gdy $\neg empty(s)$
$top : S \longrightarrow E$	zwróć wierzhołek stosu
wynik jest określony	wttw gdy $\neg empty(s)$
$newStack :\longrightarrow S$	stos pusty
Relacje	
$empty : S \longrightarrow \{true, false\}$	czy stos jest pusty?
$=: E \times E \cup S \times S \longrightarrow \{true, false\}$	relacja równości
Aksjomaty	
s1) $\forall_{e \in E} \forall_{s \in S}\ \neg empty(push(e, s))$	wynik operacji push jest stosem niepustym
s2) $\forall_{e \in E} \forall_{s \in S}\ e = top(push(e, s))$	element ostatnio włożony na stos jest na wierzchołku stosu
s3) $\forall_{e \in E} \forall_{s \in S}\ s = pop(push(e, s))$	po wykonaniu operacji push, operacja pop odtwarza stos
s4) $empty(newStack)$	nowy stos jest pusty

Zbiór S jest pojmowany zazwyczaj jako zbiór wszystkich obiektów jakie można utworzyć na podstawie klasy S, podobnie pojmowany jest zbiór E.
Zauważ, że w tej tablicy posłużyliśmy się ortografią odmienną od przyjętej w Javie dla zapisywania interfejsów. Nie ma to jednak większego znaczenia ponieważ, łatwo można przetłumacyć interfejs zapisany w jednej konwencji do drugiej.

Tabela 15.2 zawiera dwie implementacje specyfikacji S1 z tablicy 15.1. W lewej kolumnie umieściliśmy klasę Stos implementującą specyfikację S1. Prawa kolumna zawiera *matematyczny* model tej specyfikacji.

Tablica 15.2 Dwa modele specyfikacji S1

Model dany przez klasę Stos	Model matematyczny
unit Elem: **class**; ... **end** Elem;	$E = \{a, b, c, ...\}$
unit Stos: **class**;	S = set of all finite sequences
unit Linkage: **class**(e:elem, next: Linkage);	over alphabet E,
end Linkage	the empty sequence λ .
var Linkage topv;	$newstack = \lambda$
unit push: **function**(Elem e, Stos s):Stos;	$push(e, \{e_1, ..., e_n\}) = \{e, e_1, ..., e_n\}$
begin	
result := new Stos;	
result.topv := new Linkage(e, s.topv);	
end push;	
unit top: **function**(Stos s): Elem;	$top(\{e_1, ..., e_n\}) = e_1$
begin	$top(\lambda)$ nieokreślone
if (s.topv=null) **then raise** EmptyStack	
else result := s.topv.el **fi**	
end top	
unit pop: **function**(Stos s): Stos;	$pop(\{e_1, e_2, ..., e_n\}) = \{e_2, ..., e_n\}$
begin	$pop(\{e_1\}) = \lambda$
if (s.topv=null) **then raise** EmptyStack	$pop(\lambda)$ nieokreślone
else result := **new** Stos;	
result.topv := s.topv **fi**	
end pop;	
unit empty: **function** (Stos s): Boolean;	$empty(s) \equiv s = \lambda$
begin	znak = oznacza relację identyczności
result := (s.topv=null);	
end empty;	
unit equal: **function** (Stos s1,s2): Boolean;	
var aux, aux1, aux2: Boolean;	
begin	
aux:=true;	
aux1:=empty(s1); aux2:=empty(s2);	stosy są równe wttw gdy
while not(aux1 or aux2 or aux) **do**	mają te same elementy
aux := (top(s1) = top(s2));	na tych samych pozycjach.
s1 := pop(s1); aux1 := empty(s1);	
s2 := pop(s2); aux2 := empty(s2);	
od	
result := (aux1 and aux2 and aux);	
end equal;	
end Stos;	
signal EmptyStack;	

Model matematyczny nazywać będziemy modelem standardowym stosów. Dla dowolnego zbioru E można skonstruować model standardowy bazując na zbiorze E. Wszytkie takie modele są podobne. Nie muszą jednak być izomorficzne. Wystarczy rozważyć dwa modele standardowe, jeden zbudowany nad zbiorem E_1 i drugi nad zbiorem E_2, przy czym zbiory te są różnej mocy, $card(E_1) \neq card(E_2)$.

Wielu autorów przyjmuje formuły zawarte w tabeli 15.1 jako specyfikację stosów np. [11], [1]. Jednak ten zbiór formuł nie mówi całej prawdy o stosach.

Wynika to z następującego lematu.

Lemat 15.1. *Formuła*

$$(\forall s \in S)\ \neg empty(s) \Rightarrow\ \ s = push(top(s), pop(s))$$

jest niezalezna od aksjomatów s1 - s4.
Formuła ta mówi: dla każdego niepustego stosu s, wynik operacji push włozenia elementu top(s) do stosu pop(s) jest stosem s.

Dowód. Rozwazmy strukturę I_2, opisaną w Tablicy 15.3. Nietrudno sprawdzić, że jest to model aksjomatów s1 - s4, tj. wszytkie cztery formuły są prawdziwe w strukturze I_2. Zobaczmy, że formuła wymienion w lemacie nie jest prawdziwa w tej strukturze. Rozpatrzmy stos $s = \{e_1, e_2, e_3, ..., e_n\}$ taki, że $e_1 \neq e_2$. Oczywiście $top(s) = e_1$ and $pop(s) = \{e_3, ..., e_n\}$. Ale $push(top(s), pop(s) = \{e_1, e_1, e_3, ..., e_n\} \neq s$. □

Tablica 15.3 Model I_2

$E = \{a, b, c\}$
S = zbiór wszystkich skończonych ciągów znaków ze zbioru E, włączając pusty ciąg λ.
$push(e, \{e_1, e_2, ..., e_n\}) = \{e, e, e_1, e_2, ..., e_n\}$
$top(\{e_1, ..., e_n\}) = e_1$
$top(\lambda)$ niekreślony
$pop(\{e_1, e_2, e_3, ..., e_n\}) = \{e_3, ..., e_n\}$
$pop(\{e_1\}) = pop(\{e_1, e_2\}) = \lambda$, $pop(\lambda)$ nieokreślony

Upoważnia nas to do przedstawienia pełniejszej specyfikacji stosów, zob. Tablicza 15.4.

Tablica 15.4 Specyfikacja stosów S2

Sygnatura	taka sama jak w $S1$
Aksjomaty	
aksjomaty s1 - s4 oraz	
s5) $(\forall s \in S)\ \neg empty(s) \Rightarrow$ $s = push(top(s), pop(s))$	dla każdego niepustego stosu s wynikiem operacji $push$ na elemencie $top(s)$ i stosie $pop(s)$ jest stos s

Ktoś może pomyslec, im więcej formuł (dodamy do specyfikacji) tym lepiej. To się jednak może skończyć źle. W wyniku mozna otrzymać specyfikację sprzeczną.
Spójrzmy na następujący przykład S3 w tabeli 15.1.

Tablica 15.5 Specyfikacja stosów S3

Sygnatura
podobnie jak w specyfikacji S1, powiększona o dwie stałe a, b typu E
Aksjomaty
aksjomaty s1 - s5 oraz
sQ) $\neg empty(s) \Rightarrow push(e, pop(s)) = pop(push(e, s))$
oraz aksjomat
s2E) $a \neq b$

Twierdzenie 15.2. *Zbiór formuł* $\{s1 - s5, sQ, s2E\}$ *jest zbiorem sprzecznym.*

Dowód. Aksjomat s2E) stwierdza, że zbiór elementów E ma conajmniej dwa elementy a i b różne. Załóżmy, że $s \in S$ jest stosem niepustym. Mamy wtedy:

(1)	$s_1 \stackrel{df}{=} push(a, s)$	z definicji
(2)	$s_2 \stackrel{df}{=} push(b, s)$	z definicji
(3)	$s = pop(s_1)$	z (1) na mocy s3)
(4)	$s_2 = push(b, pop(s_1))$	z (2) i (3), s jest niepusty
(5)	$s_2 = pop(push(b, s_1))$	z (4) na mocy sQ)
(6)	$s_2 = s_1$	z (5) na mocy s3)
(7)	$b = top(s_2) = top(s_1) = a$	z (6) na mocy s2)

Sprzeczność! A więc specyfikacja S3 jest sprzeczna. □

Wniosek 15.3. *Specyfikacja S3 nie ma żadnej implementacji.*

Skąd wiadomo, że tak jest? Wynika to z twierdzenia o pełności rachunku programów. Gdyby istniała jakaś implementacja tego zbioru aksjomatów, to musiałaby równocześnie spełniać dwie formuły $a = b$ i $\neg(a = b)$.

sprzeczność→ ALARM

Stwierdzenie, że specyfikacja pozwala wyprowadzić zarówno pewną formułę α jak i jej negację (a więc, że jest sprzeczna) to sygnał alarmowy. W żadnym przypadku nie należy podejmować się zlecenia na wytworzenie oprogramowania, które ma spełniać wymagania sprzeczne. A jeśli nie jesteśmy pewni, że specyfikacja, która przecież jest załącznikiem do umowy o dzieło, jest wolna od sprzeczności, to wpiszmy do umowy odpowiedni kodycyl stanowiący o wysokości odszkodowania dla zleceniobiorcy za utracony czas i inne szkody.

Wracamy do specyfikacji S2. Po dokładniejszej analizie zauważamy, że można dodac do tej specyfikacji nieskończony zbiór dodatkowych formuł. Wszytkie te formuły mają strukturę zgodną ze schematem indukcji (strukturalnej) dla stosów. W ten sposób dochodzimy do kolejnej specyfikacji S4, por. Tablica 15.6.

Tablica 15.6 Specyfikacja stosów S4

Sygnatura
taka sama jak w S1
Aksjomaty
aksjomaty s1 - s5 oraz wszystkie formuły o następującym schemacie IS
$\alpha(s/s_0) \wedge \{(\forall_{s\in S}(\forall_{e\in E}(\alpha(s) \Rightarrow \alpha(s/push(e,s)))\} \Rightarrow \forall_{s\in S}\alpha(s)$
gdzie α jest dowolną formułą pierwszego rzędu i $s_0 = newStack$

Schemat indukcji powiada: jeśli formuła $\alpha(x)$ jest prawdziwa dla stosu pustego $\alpha(x/s_0)$ i jeśli dla dkażdego stosu s i dla każdego elementu e, $\alpha(x/s)$ implikuje $\alpha(x/push(e, s))$ to dla każdego stosu s zachodzi formuła $\alpha(x/s)$.

The formula does not say that there are not pathological stacks. One may say: *we shall consider only standard stacks, i.e. the stacks obtained from the empty stack in finite number of operations push.* But how to express this property as an axiom? Instead, one may say: *I am going to consider only programmable models of specification S4.* Even adding such extra requirement we can not eliminate pathological stacks. In fact papers [25, 21] prove that there exist pathological

models of specification S4. Pathological means here that there are stacks which can be popped without end and no empty stack results.

Twierdzenie 15.4. *Istnieje programowalny model specyfikacji S4 taki, że dla pewnego stosu s_1 program*

$$\textbf{while } \neg\, empty(s_1) \textbf{ do } s_1 := pop(s_1) \textbf{ done}$$

nie kończy obliczeń.

Model taki, nazywamy nieosiągalnym. W oczywisty sposób jest to model patologiczny. Dla dowodu zobacz prace [25, 21]. Druga z tych prac przynosi dwa kolejne fakty

Twierdzenie 15.5. *Niech E będzie zbiorem skończonym. Niech $\mathfrak{S}$ oznacza strukturę stosów nad zbiorem E. Zbiór formuł pierwszego rzędu prawdziwych w strukturze $\mathfrak{S}$ jest rozstrzygalny.*

Wydaje się, że jest to dobra wiadomość. Dobrze jest miec procedurę rozstrzygania o prawdziwości formuł. Jednak, okazuje się, że jest to zła wiadomość. Wynika to z kolejnego twierdzenia.

Twierdzenie 15.6. *Każda rozstrzygalna teoria pierwszego rzędu $\mathcal{T}$ posiada model programowalny i nieosiągalny, a więc patologiczny.*

Wydaje się, że jesteśmy w impasie. Że nie można skonstruować aksjomatyzacji dla struktur danych o nieskończonym zbiorze. Okazuje się, że logika algorytmiczna przychodzi tu z pomocą (zanotujmy niespełniona obietnicę [10]). Rozważmy mianowicie następująca specyfikację S5. Schemat indukcji zostaje tu zastąpiony przez pojedynczą formułę algorytmiczną, por. Tabela 15.7.

Tablica 15.7 Specyfikacja S5

Sygnatura
taka sama jak w S1
Aksjomaty
aksjomaty s1 - s5 i
s6) $\forall_{s\in S}$ **while** $\neg empty(s)$ **do** $s := pop(s)$ **done true**
ten program nie zapętla się, tzn. każdy stos jest skończony

Poniżej Następujące twierdzenie o reprezentacji [24]

Twierdzenie 15.7. *Każdy model specyfikacji S5 jest izomorficzny z pewnym standardowym modelem stosów.*

Twierdzenie to mówi, że specyfikacja S5 uchwyciła wszystkie własności struktury algebraicznej stosy. Dowolny model zbioru aksjomatów S5 jest izomorficzny ze biorem skończonych ciągów elementów ze zbioru E, a operacje push, pop i top są określone tak jak w tablicy 15.2.
Zauważyłeś, że jeden z aksjomatów stwierdza, że dla każdego stosu s program wymieniony w aksjomatach zawsze kończy obliczenie. Ta własnośc okaże się bardzo przydatna w dowodach poprawności innych algorytmów.

Widzieliśmy różne specyfikacje stosów. Pora by je porównać, zob. poniższą tablice 15.8.

Tablica 15.8 Porównanie specyfikacji S1 - S5

Specyfikacja	Uwagi
S_1	informacja o stosach niekompletna, np. formuła s5 jest niezależna od $\{s1, s2, s3, s4\}$ S_1 ma zaskakujące modele por. implementacja I_2
S_2	jeśli $card(E) = k, k \in N$, to teoria pierwszego rzędu S_2 jest rozstrzygalna, niekompletna informacja, dopuszcza modele patologiczne
S_3	specyfikacja sprzeczna, por. twierdzenie 15.2 nie istnieje *żaden* model
S_4	rozstrzygalna, niekompletna informacja dopuszcza implementacje patologiczne
S_5	informacja kompletna, każdy model jest izomorficzny z pewnym modelem standardowym algorytmiczna teoria jest nierozstrzygalna.

Uwaga 15.8. *Zbiór formuł pierwszego rzędu prawdziwych w strukturze danych stosów nad skończonym biorem zbiorem E elementów jest rozstrzygalny. Równocześnie specyfikacje S2 i S4 mają modele niestandardowe. W modelach tych polecenie s:=pop(s) może byc powtarzane dowolnie wiele razy i nie doprowadza do stosu pustego empty(s).*

15.2 Implementacje stosów

Znane są dwie implementacje stosów: stos jako tablica i wskaźnik, oraz stos jako lista. Przedstawimy jeszcze jedną, mniej znaną implementację stosów – stosy jako liczby.

15.2.1 Stosy jako tablice

15.2.2 Stosy jako listy

Ta implementacja – lepiej mówić *ten model*, została opisana w tabeli 15.2. Klasa Stosy jaką tam zamieściliśmy jest definicją struktury algebraicznej $\mathfrak{S}_l$.

$$\mathfrak{S}_l \stackrel{df}{=} \langle \{|Elem| \cup |Stos|\}, push, pop, top, empty, equal \rangle$$

Zbiór $|Elem|$ jest zbiorem obiektów o, które spełniają relację o *is Elem*. Podobnie, zbiór $|Stos|$ jest zbiorem obiektów o, które spełniają relację o *is Stos*. Oba zbiory nie są zbiorami istnejącymi w trakcie wykonywania jakiegokolwiek programu. Myślimy o nich jako o abstrakcji, zbiorach potencjanych obiektów. Deklaracje funkcji stanowią definicje

dowód poprawności implementacji por ks. AL

15.2.3 Stosy jako liczby naturalne

Niech zbiór E elementów będzie skończony. W szczególnym przypadku gdy liczba elementów równa jest 10, możemy utożsamiać stosy z liczbami naturalnymi l, takimi, że $l \geq 10$. Stos pusty jest liczbą 10. Niech stos s będzie reprezentowany przez liczbę n. Wtedy wynik operacji $push(e, s)$ będzie reprezentowany

przez liczbę $n * 10 + e$. W tym przypadku rozważamy następująca klasę.

```
unit Stosy10: class;
  const k=10;
 signal Emptystack;
 signal WrongElem;
 unit Elem: class(l: integer);
 begin
  if l<0 orif l>k-1 then raise WrongElem fi
 end Elem;
-------------------------------------------------
 unit Stos: class(l: integer);
  unit push: function(e:Elem, s:Stos):Stos;
  begin
    result:=new Stos((s.l-k)*k+e.l+k+1);
  end push;
  unit pop: function(s:Stos):Stos;
  begin
    if not empty(s) then
     result:= new Stos( ((s.l-k-1)div k)+k )
    else
     raise EmptyStack
    fi
  end pop;
  unit top: function(s:Stos):Elem;
  begin
    if not empty(s) then
     result:= new Elem( (s.l-k-1)mod k )
    else
     raise EmptyStack
    fi
  end top;
  unit empty: function(s:Stos):Boolean;
  begin
    result:=s.l=k
  end empty;
  unit equal: function(s1,s2: Stos):Boolean;
  begin
   result:= s1.l=s2.l
  end equal;
 begin
  if l<k then l := k fi
 end Stos
end Stosy10;
```

Klasa *Stosy*10 jest definicją struktury algebraicznej $\mathfrak{S}_{10}$. Na uniwersum tej struktury składają sie dwa zbiory $|Elem|$ oraz $|Stos|$ opisane przez klasy *Elem* i *Stos*. Działania tej struktury są zdefiniowane przez metody (tj. funkcje) $push, pop, top, empty, equal$.

$$\mathfrak{S}_{10} \stackrel{df}{=} \langle \{|Elem| \cup |Stos|\}, push, pop, top, empty, equal \rangle$$

Zbiór $|Elem|$ zawiera dziesięć obiektów jakie można utworzyć obliczając wartość wyrażenia new Elem(i), i=0 1 ... 9. Zbiór $|Stos|$ jest nieskończonym zbiorem obiektów klasy *Stos*. Działanie
$push\colon |Elem|\times|Stos| \to |Stos|$ zdefiniowaliśmy podając deklarację funkcji *push*. Można łatwo sprawdzić każdy z aksjomatów stosów. Czy dla każdych $e \in Elem$ i $s \in Stos$ jest prawdą, że

$$top(push(e, s) = e$$

Wynik operacji $push(e, s)$ jest obiektem typu *Stos* dla którego wartość atrybutu l jest równa $(s.l - k) * k + e.l + k + 1$. Zastosujmy do tej liczby działanie *top*. A więc obliczmy wartość wyrażenia $((s.l - k) * k + e.l + k + 1 - k - 1)$**mod** k. Widać że jest to $e.l$. I to się zgadza.
I tak po kilku krokach sprawdzimy że każdy z aksjomatów stosów jest prawdziwy w implementacji opisanej przez klasę Stosy10.

Zadanie 15.1. *Sprawdź pozostałe aksjomaty.*

Wskazówka. Możesz się zastanawiać w jaki sposób sprawdzić że dla każdego stosu s program **while** not empty(s) **do** s:= pop(s) **od** zakończy obliczenia tj. nie zapętli sie? Przypomnij sobie odpowiedni aksjomat liczb całkowitych.

Zadanie 15.2. *Opisz zbiór $|Stos|$. Czy jest to drzewo? Skończone?*

15.2.4 Implementacja specyfikacji S4

W tym miejscu podajemy dwie implementacje specyfikacji S4 por.15.1 zrealizowane jako klasa Stosy13 i klasa Stosy14. W obu klasach znajdujemy klasę Nat. klasa Nat zawarta w klasie Stosy13 różni się od klasy Nat zawartej w klasie Stosy14.
Poniżej znajdziesz specyfikację SNat.

A w tej tabeli są dwie różne implementacje specyfikacji SNat.

To jest implementacja I4.

```
unit Stosy13: class; (* implementuje specyfikację S4 *)
  const k=10;
  signal Emptystack;
  signal WrongElem, NegativeNat;
  unit Nat: class(l: integer);
    unit add: function(n: Nat): Nat;
    begin
      result:= new Nat(l+n.l)
    end add;
    unit zero: function : Nat;
    begin
      result:= new Nat(0)
    end zero;
  begin
    if l < 0 then raise NegativeNat fi;
  end Nat;
-------------------------------------------------
  unit Elem: class(l: Nat);
  begin
   if l.l<0 orif l.l>k-1 then raise WrongElem fi
  end Elem;
-------------------------------------------------
  unit Stos: class(l: Nat);
    unit push: function(e:Elem, s:Stos):Stos;
    begin
      result:=new Stos((s.l.l-k)*k+e.l.l+k+1);
    end push;
    unit pop: function(s:Stos):Stos;
    begin
      if not empty(s) then
       result:= new Stos( ((s.l.l-k-1)div k)+k )
      else
       raise EmptyStack
      fi
    end pop;
    unit top: function(s:Stos):Elem;
    begin
      if not empty(s) then
       result:= new Elem( (s.l.l-k-1)mod k )
      else
       raise EmptyStack
      fi
    end top;
    unit empty: function(s:Stos):Boolean;
    begin
      result:=s.l.l=k
    end empty;
    unit equal: function(s1,s2: Stos):Boolean;
    begin result:= s1.l.l=s2.l.l
    end equal;
  begin
   if l.l<k then l.l := k fi
  end Stos
end Stosy13;
```

Implementacja I5 rózni się od poprzedniej inną definicją klasy Nat.

```
unit Stosy14: class;
  var k: Nat, signal Emptystack, WrongElem, NegativeNat;
  unit Nat: class(i,l,m: integer);
    unit add: function(n: Nat): Nat;
    begin result:= new Nat(i+n.i,l*n.m+n.l*m ,m*n.m)
    end add;
    unit zero: function : Nat;
    begin result:= new Nat(0, 0, 1)
    end zero;
  begin
    if l < 0 then raise NegativeNat fi;
  end Nat;
-------------------------------------------------------------
  unit Elem: class(l: Nat);
  begin    if l.l<0 orif l.l>k-1 then raise WrongElem fi
  end Elem;
-------------------------------------------------------------
  unit Stos: class(l: Nat);
    unit push: function(e:Elem, s:Stos):Stos;
    begin
      result:=new Stos((s.l.l-k)*k+e.l.l+k+1);
    end push;
    unit pop: function(s:Stos):Stos;
    begin
      if not empty(s) then
       result:= new Stos( ((s.l.l-k-1)div k)+k )       else
       raise EmptyStack       fi
    end pop;
    unit top: function(s:Stos):Elem;
    begin
      if not empty(s) then
       result:= new Elem( (s.l.l-k-1)mod k )
      else
       raise EmptyStack
      fi
    end top;
    unit empty: function(s:Stos):Boolean;
    begin
      result:=s.l.l=k
    end empty;
    unit equal: function(s1,s2: Stos):Boolean;
    begin
     result:= s1.l.l=s2.l.l
    end equal;
  begin
    if l.l<k then l.l := k fi
  end Stos
begin
  k:= new Nat(10,0,1);
end Stosy14;
```

15.2.5 typ formalny czy dziedziczenie ?

Porównać

W tym miejscu porównamy dwa sposoby wprowadzania typu element do struktury stosów:
- element jako typ formalny – poprawnie, ale ...
- element jako typ abstrakcyjny, niekompletny, do dalszego sprecyzowania poprzez dziedziczenie, wada – dopuszcza rózne rozszerzenia.
Ale można się przed tym zabezpieczyc zamykając dziedziczenie zob, typ człowiek w podręczniku Loglanu.

Przykład 15.1. *Typ El jako parametr formalny.*

```
unit Stosy: class(type El, function equal(e1, e2:El):Boolean);
   unit Stos: class; ... end Stos;
   push: function(e:El, s: Stos): Stos;
   end push;
   pop
   end pop;
   ...
end Stosy;
```

W takim podejściu zanim użyjesz generatora new Stosy lub bloku prefiksowanego Stosy(...) block .. end;
musisz utworzyć typ ELA – parametr aktualny i funkcję eq – funkcję charakterystyczna relacji równośći, a więc funkcja eq ma miec odpowiednie własności.
Teraz
Stosy(ELA,eq) block ... end
powinno zadziałać poprawnie – algorytm wewnątrz tego bloku realizowany w środowisku Stosy(ELA,eq). Tzn. elementy maja być typu ELA i do porównywana elementów wykorzystuje się funkcję eq.

A więc na użytkownika klasy Stosy nakłada się obowiązki ...
Inaczej mozna tak

Przykład 15.2. *Typ El jako atrybut abstrakcyjnej klasy Stosy.*

```
unit Stosy: class;
   unit El: class;
      unit virtual eq: function(e1, e2: El):Boolean; ... end eq
   end El;
   unit Stos: class; ... end Stos;
   push: function(e:El, s: Stos): Stos;
   end push;
   pop
   end pop;
   ...
end Stosy;
```

W takim podejściu trzeba rozszerzyc deklaracje

```
unit MojeStosy: Stosy class;
  MojeEl: El class;
    unit virtual eq: function(e1,e2: MojeEl): Boolean; ... end eq;
  MojeEl;
end MojeStosy;
```

Tu też musze pamiętac czego oczekuje się od funkcji eq.
Wada: jak zapanować nad tym co będzie wstawiane do stosu?

Drobna modyfikacja powyższego podejścia polega na zabronieniu rozszerzenia typu MojeEl w sposób dowolny.

Przykład 15.3. *Typ zabezpieczony przed ...?*

```
unit Stosy: class;
  unit El: class;
    unit virtual eq: function(e1, e2: El):Boolean; ... end eq
  end El;
  unit Stos: class; ... end Stos;
  push: function(e:El, s: Stos): Stos;
  end push;
  pop
  end pop;
  ...
end Stosy;
```

W takim podejściu trzeba rozszerzyć deklaracje

```
unit MojeStosy: Stosy class;
  MojeEl: El class;
    unit virtual eq: function(e1,e2: MojeEl): Boolean; ... end eq;
  begin
    if not (this El is MojeEl) then raise ExceptionEB fi;
  MojeEl;
end MojeStosy;
```

Teraz do kolejki wchodza tylko obiekty typu MojeEl.

Rozdział 16

Kolejki

Struktura kolejek występuje w tak wielu zastosowaniach komputerów i w rozmaitych elementach systemów operacyjnych, baz danych, protokołach komunikacyjnych, etc, że musimy się jej przyjrzeć bliżej.
Struktura kolejek jest bliska strukturze stosów. Widzieliśmy wcześniej 15.1, że łatwo zaimplementować interfejs stosów realizując w istocie system kolejek. Dzieje się tak i w drugą stronę: po podaniu interfejsu I kolejek można podać klasę będącą modelem struktury stosów, która będzie implementować ten interfejs.
Czym jest więc struktura kolejek?

Definicja 16.1. *Każda struktura algebraiczna spełniająca następujące warunki*

(U) Uniwersum struktury jest sumą dwu rozłącznych zbiorów E i Q, $E \cap Q = \emptyset$,

(S) Sygnatura struktury zawiera operacje f, g, h i relacje $e, =_E, =_Q$ takie, że
$f\colon E \times Q \to Q$
$g\colon Q \to Q$
$h\colon Q \to E$
oraz
$em\colon Q \to \{true, false\}$
$=_E\colon E \times E \to \{true, false\}$
$=_Q\colon Q \times Q \to \{true, false\}$

(A) Aksjomaty. Struktura zapewnia prawdziwość następujących formuł
s1) $\forall_{e\in E}\forall_{s\in Q}\ \neg em(f(e,s))$
s2) $\forall_{e\in E}\forall_{s\in Q}\ em(s) \Rightarrow h(f(e,s)) =_E e$
s3) $\forall_{e\in E}\forall_{s\in Q}\ em(s) \Rightarrow g(f(e,s)) =_S s$
s4) $\forall_{e\in E}\forall_{s\in Q}\ \neg em(s) \Rightarrow h(s) =_E h(f(e,s))$
s5) $\forall_{e\in E}\forall_{s\in Q}\ \neg em(s) \Rightarrow f(e, g(s)) =_Q g(f(e,s))$
s6) $\forall_{s\in Q}\{\textbf{while}\ \neg em(s)\ \textbf{do}\ s := g(s)\ \textbf{od}\}\ em(s)$

jest strukturą kolejek (FIFO).

Zazwyczaj zbiór E nazywamy zbiorem elementów a zbiór Q zbiorem kolejek. Operacje f, g, h nazywane są zazwyczaj: $put, get, first$, po polsku: *wstaw, usuń, pierwszy*. Funkcja boolowska em sprawdza czy kolejka jest pusta. Można łatwo sprawdzić, że dwie znane implementacje kolejek spełniają tę definicję.

Przykład 16.2. *Rozważmy następującą klasę*

```
unit KolejkiT: class;
  unit Element: class ;
  end Element;
  unit Kolejka: class;
    var T: arrayof Element, i,j: integer;
  begin
    array T dim (1:50);
  end Kolejka;
  signal EmptyError, FullError;
  put: function(e: Element, s: Kolejka): Kolejka;
  begin
    if j<>50 then s.T(s.j) := e; s.j := s.j+1; result:=this Kolejka
    else raise FullError fi;
  end put ;
  first: function(s: Kolejka): Element;
  begin
    if not em(s) then result := s.T(s.i);
    else raise EmptyError fi
  end first ;
  get: function(s: Kolejka): Kolejka;
  begin
    if not em(s) then s.i := s.i+1; result:=this Kolejka
    else raise EmptyError fi
  end get ;
  em: function(s: Kolejka): Boolean;
  begin
    result := s.i=s.j
  end em ;
end KolejkiT;
```

Bardziej "obiektowo" wygląda ta sama idea kolejki zapisana w ten sposób.

Przykład 16.3. *Zamiast klasy KolejkiT można rozważać klasę KolejkiT2. W tym przypadku mówimy o metodach put, get, first i em klasy Kolejki. Zauważ, że obliczanie wartości wyrażeń obiektowych wymaga przekazywania mniejszej liczby parametrów.*

```
unit KolejkiT2: class;
  unit Element: class ;
    unit virtual equal: function(e: Element): Boolean; ... end equal
  end Element;
  unit Kolejka: class;
    var T: arrayof Element, i,j: integer;
    put: procedure(e: Element );
    begin
      if j<>50 then T(j) := e; j := j+1
      else raise FullError fi;
    end put ;
    first: function: Element;
    begin
      if not em(s) then result := T(i);
      else raise EmptyError fi
    end first ;
  get: procedure;
  begin
      if not em(s) then i := i+1;
      else raise EmptyError fi
  end get ;
  em: function : Boolean;
  begin
      result := (i=j)
  end em ;
  begin
    array T dim (1:50);
  end Kolejka;
  signal EmptyError, FullError;
end KolejkiT2;
```

W tym przypadku trzeba zmienic ortografię wymagań. Np. warunek $\neg em(s) \Rightarrow first(s) =_E first(put(e,s))$ *zapiszemy tak* $\neg s.em \Rightarrow s.first.equal(s.put(e).first)$ *Nietrudno zauważyć, że tak zmienione aksjomaty kolejek są spełnione przez klasę KolejkiT2.*

Z taką klasą KolejkiT2 można pokusić się o jej zastosowanie.

Przykład 16.4. *Ten blok wykorzystuje klasę KolejkiT2 (i równocześnie rozwija klasę Element) w celu ...*

```
KolejkiT2 block
  Elem: Element class;
    unit virtual equal: function; ... end equal;
  end Elem;
  k1, k2: Kolejka, e:Elem;
begin
  k1:=new Kolejka; k2:= new Kolejka;
  e:=new Elem(...);
  k1.put(e);
  k2.get;
end
```

uzupełnić

Zauważmy, że z pary obiektów klasy Kolejki możemy stworzyć stos. (Podobnie z pary obiektów klasy Stos możemy zbudować kolejkę.) Te związki ukazują więź łączącą pojęcia stosu i kolejki.

Przykład 16.5. *Kolejki jako listy*

Te modele teorii kolejek LIFO są algorytmicznie nieodróznialne tzn. dla dowolnej formuły algorytmicznej α, formuła ta jest spełniona w implementacji

(tj. modelu) KolejkiT wtedy i tylko wtedy gdy jest spełniona w modelu KolejkiL. Sformułujemy twierdzenie wzmacniające tę obserwację. Niech E będzie dowolnym zbiorem wyposażonym w relację równości $=_E$ spełniająca aksjomaty równości: zwrotność, przechodniość i antysymetrię. Rozważmy zbiór $Fseq(E)$ skończonych ciągów elementów ze zbioru E. (Ciąg pusty $\emptyset \in Fseq(E)$.
Para $\langle E, Fseq(E)\rangle$ rozpatrywana z następującymi operacjami jest modelem teorii kolejek.

Niesprzeczność.
Model standardowy kolejek
Tw. o reprezentacji
Aksjomatyzacja pierwszego rzędu ma modele patologiczne.

Rozdział 17

Zbiory skończone

W wielu programach operacjami dominującymi sa operacje na zbiorach skończonych. Komputer w naturalny sposób operuje na liczbach. Jednak większość oprogramowania wymaga przechowywania i odszukiwania informacji. Informacja ta jest oczywiście zawarta w pewnym zbiorze skończonym. W zależności od zadania jakie program ma rozwiązywać zbiór skończony będzie implementowany

Ostatnio przyjęło się mówić o kontenerach (*ang.* container) jako pojemnikach mieszczących skończone zbiory obiektów[1]. Zastanówmy się jak ma wyglądać specyfikacja pojęcia kontener.

17.1 Specyfikacja kontenerów

Po doświadczeniach ze strukturą stosów spróbujemy podać definicję aksjomatyczną struktury kontenerów.

Definicja 17.1. *Kontenerem nazywamy strukturę algebraiczną $\mathcal{K}$, której uniwersum składa się z dwu rozłącznych zbiorów E i S, $E \cap S = \emptyset$. W strukturze tej mamy następujące operacje i, d, a oraz predykaty r, p, c, q.*

$$\mathcal{K} = \langle E \cup S; i, d, a, r, p, c, q \rangle$$

Sygnatura tej struktury jest wyliczona poniżej

$i\colon E \times S \to S$

$d\colon E \times S \to S$

$a\colon S \to E$

Relacje

$r\colon E \times E \to B_0$

$p\colon S \to B_0$

$c\colon E \times S \to B_0$

$q\colon S \times S \to B_0$

[1] W latach 1970 używano słowa dictionary [?]

Aksjomaty struktury kontenerów są następujące

k1) $\forall_{e\in E}\,\forall_{s\in S}\,(c(e,i(e,s))\wedge\forall_{e'\in E\wedge e'\neq e}(c(e',s)\Leftrightarrow c(e',i(e,s)))$

k2) $\forall_{e\in E}\,\forall_{s\in S}\,(\neg c(e,d(e,s))\wedge\forall_{e'\in E\wedge e'\neq e}(c(e',s)\Leftrightarrow c(e',d(e,s)))$

k3) $\forall_{s\in S}(p(s)\Leftrightarrow\forall_{e\in E}\,\neg c(e,s))$

k4) $\neg(p(s)\Rightarrow c(a(s),s))$

k5) $\forall_{e\in E}\,\forall_{s\in S}\;\; c(e,s)\Leftrightarrow$

$$\left\{\begin{array}{l}\textbf{block}\\ \quad\textbf{var}\ bool:Boolean, s1:S\\ \textbf{begin}\\ \quad s1:=s;\ bool:=false;\\ \quad\textbf{while}\ \neg bool\wedge\neg p(s1)\ \textbf{do}\\ \quad\quad e1:=a(s1);\\ \quad\quad bool:=r(e1,e);\\ \quad\quad s1:=d(e1,s1)\\ \quad\textbf{od}\\ \textbf{end}\end{array}\right\}\ bool$$

k6) $\forall_{s\in S}\,\{\textbf{while}\ \neg p(s)\ \textbf{do}\ s:=d(a(s),s)\ \textbf{od}\}\ p(s)$

k7) $q(s,s')\Leftrightarrow\forall_{e\in E}(c(e,s)\Leftrightarrow c(e,s'))$

k8) $\forall_{e\in E}\ r(e,e)$

k9) $\forall_{e,e'\in E}\ r(e,e')\Leftrightarrow r(e',e)$

k10) $\forall_{e,e',e''\in E}\ (r(e,e')\wedge r(e',e''))\Rightarrow r(e,e'')$

Tak jak zwykle nasuwają się dwa pytania

1. Czy istnieją jakieś kontenery? Czyli, czy zbiór formuł k1) - k10) jest niesprzeczny?
2. Czy definicja kontenera wyklucza niepożądane struktury? Czy modelami zbioru aksjomatów są tylko takie struktury jakie akceptuje nasza intuicja?

Zacznijmy od następującego twierdzenia

Twierdzenie 17.1. *Zbiór aksjomatów k1) – k10) jest niesprzeczny.*

Dowód. Dowód polega na pokazaniu modelu dla tego zbioru. Niech E będzie dowolnym zbiorem, niech $=_E$ oznacza relację identyczności w zbiorze E.

Rozpatrzmy zbiór $Fin(E)$ wszystkich skończonych podzbiorów zbioru E. Przyjmiemy następującą interpretację symboli sygnatury kontenera:

$i(e,s)$	$d(e,s)$	$a(s)$	$p(s)$	$c(e,s)$	$q(s,s')$
$s\cup\{e\}$	$s\setminus\{e\}$	$a(s)\in s$	$s=\emptyset$	$e\in s$	$s=_S s'$

Łatwo widać, że wszystkie aksjomaty kontenerów są w tej interpretacji prawdziwe. Wyjaśnienia wymaga jedynie interpretacja operatora a, funkcja ta nazywana bywa selektorem. Istnienie funkcji $a\colon (Fin(E)\setminus\emptyset)\to E$ wynika z następującego twierdzenia teorii zbiorów, które tradycyjnie nazywane jest *Aksjomatem wyboru dla zbiorów skończonych*

Tw. ACF *Dla każdego zbioru X istnieje funkcja f przyporządkowująca każdemu niepustemu, skończonemu podzbiorowi zbioru X, jeden element $x\in X$.* por. [26]

Dowód twierdzenia ACF jest efektywny, nie wymaga pewnika wyboru.
Podsumowując, stwierdzamy, że zbiór formuł k1) - k10) jest niesprzeczny. □

Struktura algebraiczna opisana powyżej nazywana będzie *modelem standardowym.*

Wspomnieliśmy wcześniej, że znanych jest wiele różnych modeli teorii kontenerów.
Tablice
Jeśli wiadomo, że zbiór E jest pewnej niedużej mocy, powiedzmy $card(E) < 1000$ to każdy podzbiór zbioru E można zapisać w tablicy Boolowskiej o 1000 elementach. Operacje wstawiania, usuwania i wybierania elementu będą szybkie, ich koszt nie przekroczy $c * 1000$, gdzie c jest pewną stałą. Podobnie będzie ze sprawdzaniem czy element e należy do zbioru s i czy zbiór s jest pusty. Ten sposób przedstawiania zbiorów występuje w języku Pascal.

Zadanie 17.1. *Napisz klasę implementująca strukturę kontenerów w tablicach i oszacuj koszt każdej operacji.*

Kłopot występuje gdy liczność zbioru E jest dużo większa lub gdy równocześnie chcemy przechowywac wiele podzbiorów zbioru E. Wtedy koszt operacji staje się zbyt wielki.
Jeżeli w działaniach na podzbiorach zbioru E nie występuje operacja $mb(e, s)$ sprawdzania przynależności elementu e do zbioru s i ponadto wiadomo, że operacja usuwania elementu $del(e, s)$ odnosić się będzie zawsze do tego elementu e, który do zbioru s został wstawiony najdawniej, to najlepiej przedstawić kontener w postaci *kolejki* **??**.

Zadanie 17.2. *Napisz klasę implementującą strukturę kontenerów w kolejkach i oszacuj koszt każdej operacji.*

Stosy warto przyjąć jako implementację kontenera gdy operacja $mb(e, s)$ ogranicza się do sprawdzenia czy element e jest równy wierchołkowi stosu i gdy wiadomo, że operacja usuwania elementu zawsze odnosić się będzie do elemntu ostatnio wpisanego do stosu.

Zadanie 17.3. *Napisz klasę implementująca strukturę kontenerów w stosach i oszacuj koszt każdej operacji.*

Hash tables

Liczba różnych modeli pojęcia kontenera jest nieograniczona.
Skąd wiadomo, że nie istnieje model zbioru formuł k1 – k10) patologiczny, nieodpowiadający naszej intuicji? Na to pytanie odpowiedzi udziela następujące

Twierdzenie 17.2. *Każda struktura algebraiczna $\mathcal{K} = \langle E \cup S; i, d, a, r, p, c, q\rangle$ będąca modelem kontenera jest izomorficzna z modelem standardowym nad zbiorem E.*

Dowód. Dla dowodu należy pokazać, że istnieje taka funkcja $f\colon S \to Fin(E)$, która jest różnowartościowa i **na**, i taka, że spełnione są następujące równości

$f(i(e,s)) = f(s) \cup \{e\}$
$f(d(e,s)) = f(s) \setminus \{e\}$
$a(f(s)) = a(s)$
$mb(e,s) \Leftrightarrow e \in f(s)$
...
Definiujemy funkcję f w ten sposób
$f(s) \stackrel{df}{=} \{e \in E : mb(e,s)\}$
Funkcja f jest różnowartościowa ponieważ ...
Funkcja f odwzorowuje zbiór S na zbiór $Fin(E)$ ponieważ ...
Funkcję a: definiujemy ...
Łatwo sprawdzic, że zachodzą równości ... □

Istnieje wiele struktur spełniających tę definicję. Programistów interesują struktury, które zaimplementowano jako klasy. W niemal każdym podręczniku *Algorytmów i struktur danych* znajdziesz mnóstwo przykładów struktury kontener.

Programy wykorzystujące strukturę kontenera będą działać szybciej jeśli koszt operacji wymienionych w powyższej definicji jest mały. Powstało wiele implementacji struktury kontener. Na ogół, implementacje te wykorzystują strukturę drzewa i wtedy koszt pojedynczej operacji jest proporcjonalny do logarytmu liczby elementów zawartych w kontenerze.

17.2 Definicja instrucji forall

Programiści odczuwają potrzebę instrukcji podobnej do polecenia for, ale pozwalającej na powtórzenie pewnych instrukcji dla każdego elementu z pewnego skończonego zbioru S. W tym rozdziale przedstawimy pewien sposób realizacji polecenia foreach. Container może zawierać taką klasę.

```
unit forall:class;
        var e: Elem;
begin
        if not empty() then
            e := first();
            while not at_last() do
                INNER;
                e := next()
            od
        fi
end forall;
```

Zastosowanie
Poniższa instrukcja bloku prefiksowanego powtórzy <moje instrukcje> dla każdego elementu e znajdującego się w bieżącej instancji containera. Jest wskazane by intrukcje te posługiwały się zmienną nielokalną e.
Jest również wskazane by instrukcje te nie zmieniały tej zmiennej - dokładniej by nie wykorzystywały poleceń first() ani next().
Co z instrukcjami ins oraz del? Mogą one zmienić strukturę kontenera i spowodować pominięcie lub powtórzenie iterowanej instrukcji.

```
pref forall block
begin
<moje instrukcje>
end
```

Ta instrukcja jest równoważna następującej instrukcji.

```
block
  var e: Elem;
begin
  if not empty() then
  e := first();
  while not at_last() do
    <moje instrukcje>;
    e := next()
  od
  fi
end (* forall *) ;
```

Jeśli chcemy udowodnić, że instrukcja <moje instrukcje> zostanie powtórzona dla każdego elementu znajdujacego się w kontenerze - w chwili rozpoczęcia instrukcji forall
to musimy wykazać, że spełnione są pewne dodatkowe warunki:
jakie?
(* ********** *)

Trzeba zbadać inne warianty:
- unit forall: class(**inout** a:Elem, c: Container); ...
lub
- unit forall: class(a: Elem); ...

end Container;

17.3 Antoniego Kreczmara kontener obiektów

VLP – maszyna wirtualna Loglanu zarządza obiektami klas i tablic, jakie powstają podczas wykonywania programu. System $\mathcal{AK}$ obiektów różni się od kontenerów omawianych wcześniej.

Definicja 17.2. *System obiektów Loglanu jest to struktura algebraiczna*

$$\mathcal{AK} = \langle F \cup S \cup \{none\}; r, a, i, d, m, k, e \rangle$$

taka, że jej uniwersum jest unią rozłącznych zbiorów Fr i St oraz {**none**}. *Operacje sa następujące:*

$r : S \to F$

$a : S \to F$

$i : F \times S \to S$

$d : F \times S \to S$

$m : F \times S \to \{true, false\}$

$k : F \times S \to S$

$e \in S$

We shall consider variables of type F - for frames, usually denoted by $f, f', ...$ and of type S - for states,usually denoted by $s, s', ...$
The set of functors and predicates of the theory's language consists of:
and two constants none $\notin \{F \cup S\}$ and $eS \in S$. The value of any variable f of type F is a frame, or none.

***Axioms** specific of the theory $\mathcal{ATHM}$ are given below*

HM_1) $\forall_{s \in S}\ \neg m(r(s), s)$
For every state s, operation $res(s)$ returns a new frame, not an element of s

HM_2) $\forall_{f \in F}\ \neg m(f, e)$
the initstate e is the empty set of frames

HM_3) $\forall_{s \in S} \left\{ \begin{array}{l} \textbf{while } s \neq e \textbf{ do} \\ \quad s := d(a(s), s) \\ \textbf{od} \end{array} \right\} (s = eS)$

For every state s, the program while (above) terminates, hence, every state is a finite set of frames

HM_4) $\forall_{s \in S}\ s \neq e \Rightarrow m(a(s), s)$
For every non-empty state, function amb returns a member of the state s

HM_5) $\forall_{f \in F} \forall_{s \in S} \{s' := i(f, s)\}(m(f, s') \wedge \forall_{f' \in F}(f' \neq f \Rightarrow m(f', s) \Leftrightarrow m(f', s'))$
operation ins adds frame f to the state s

HM_6) $\forall_{f \in F} \forall_{s \in S} \{s' := d(f, s)\}(\neg m(f, s') \wedge \forall_{f' \in F}(f' \neq f \Rightarrow m(f', s) \Leftrightarrow m(f', s'))$
operation del deletes frame f from the state s

HM_7) $m(f, s) \Leftrightarrow \left\{ \begin{array}{l} \textbf{begin} \\ \quad s1 := s;\ bool := false; \\ \quad \textbf{while } s1 \neq e \wedge \neg bool \\ \quad \textbf{do} \\ \quad\quad f1 := a(s1); \\ \quad\quad \textbf{if } f = f1 \textbf{ then } bool := true \textbf{ fi}; \\ \quad\quad s1 := d(f1, s1); \\ \quad \textbf{od} \\ \textbf{end} \end{array} \right\} bool$

*This formula defines the relation member in terms of operations a, d and e. It is **not** an implementation however. We postulate that the implemented cost should be constant.*

HM_8) *The operation kill is characterised by the axioms of the following scheme.*
The index k may be any natural number $k > 0$, let $1 \leq i \leq k$.

$$\underbrace{((f_1 = ... = f_k) \wedge m(f_1, s))}_{precondition} \Rightarrow \underbrace{[s' := k(f_i, s)]}_{\text{statement}} \underbrace{(f_1 = ... = f_k = \textbf{none})}_{postcondition}$$

Any formula of this form is an axiom, it tells that operation kill in one move nullifies all the references to the object pointed by the variable f_i. And indeed, in the system of Kreczmar the cost of the operation kill is constant.

HM9) $\forall_{s\in S}\forall_{f\in F} m(f,s) \Rightarrow f \neq none$

HM10) $\forall_{s,s'\in S}\, s =_S s' \Leftrightarrow \forall_{f\in F}(m(f,s) \Leftrightarrow m(f,s'))$

17.3.1 Properties of the specification

One can investigate the properties of the specification itself. We are able to state an important metatheorem about the system of axioms in HM. The following theorem was not formulated in [**?**]. H. Oktaba proved a theorem on consistency for a similar set of axioms [**?**], basically it was the set {HM_1 - HM_7 }.

Twierdzenie 17.3. *(on consistency of the set {HM1-8})*
The system of axioms HM_1 – HM_8 has a model.

For a sketch of the proof see the Appendix A. The model constructed in the proof will be called the *standard model.*

H. Oktaba proved another important fact:

Twierdzenie 17.4. *(representation theorem)*
Every two models of the axioms HM_1 – HM_7 are isomorphic, up to implementation of operations amb and res, to the standard model.

The meaning of the theorem is: *the operations ins, mb, are precisely described in the set of axioms, when the description of properties of operations res and amb was left understated. This was done intentionally, for there are various possibilities how to treat the released memory and how to choose where(address) to allocate a new object.*. Everybody agrees that the operations *res* and *amb* may be implemented in several versions. Many different implementation of *res* (respectively *amb*) operation allows to prove that the requirements mentioned in the axioms HM_1 and HM_4 are satisfied. Therefore, it is not easy to prove the representation theorem for the axioms HM_1 – HM_8. One must decide on his choice of the *res* operation – how it will be implemented in one system, while other systems may prefer different solution. podaj przykłady

17.3.2 Variations of axiom's system

Are the simplifications we made important? One can easily observe two points:

- One can consider a slightly different operation reserve - with a parameter *appetite* defining the size required for an object. This can be easily done by modification of the signature $res : S \times N \to F$ and leads to a new (consistent) set of axioms.
- Another extension of our system HM is defined when one describes the internal structure of an object. (The structure is determined by the declaration of class). This extension is also consistent.

Till now we needed not to introduce an operation of garbage collection. In our abstract version the set of Frames is isomorphic with the set of natural numbers.To make our theory more realistic we should introduce a postulate that the set of frames is finite.In this case a need arises of garbage collection.

One can ask how to express the property *the set Fr of frames is finite*? The answer is easy:
HM_9) $\quad \exists_{s_0 \in St} \forall_{f \in Fr} mb(f, s_0)$
Which reads: *the set of frames is equal to some state, hence Fr is a finite set.*
The set of the axioms $\mathrm{HM}_1 - \mathrm{HM}_9$ is inconsistent. However, it is quite easy to repair it. We leave this as an exercise. **Hint.** Introduce a predicate $full$, a dual to the predicate *empty*.

Rozdział 18

Kolejki priorytetowe

Pojęcie *kolejki priorytetowe* jest zbliżone do pojęcia kontenera. Mówimy o kolejkach priorytetowych gdy zbiór E elementów jest wyposażony w dwie relacje: $=_E$ – relację równości oraz relację $<$ porządku w zbiorze E.
Kolejki priorytetowe mają niewiele wspólnego z kolejkami w zwykłym sensie tj. kolejkami FIFO. Nie są znane implementacje kolejek priorytetowych inne niż w drzewach.

18.1 Pojęcie kolejki priorytetowej

Definicja 18.1. *Kolejką priorytetową nazywamy strukturę algebraiczną $\mathcal{KP}$, której uniwersum składa się z dwu rozłącznych zbiorów E i S, $E \cap S = \emptyset$.*

$$\mathcal{KP} = \langle E \cup S; i, d, m, r, \leq, p, c, q \rangle$$

W strukturze tej mamy następujące operacje i, d, m oraz predykaty $r, <, p, c, q$. Sygnatura operacji jest wyliczona poniżej

$i \colon E \times S \to S$

$d \colon E \times S \to S$

$m \colon S \to E$

Relacje

$r \colon E \times E \to B_0$

$\leq \colon E \times E \to B_0$

$p \colon S \to B_0$

$c \colon E \times S \to B_0$

$q \colon S \times S \to B_0$

Aksjomaty struktury kolejek priorytetowych

k1) $\forall_{e\in E}\,\forall_{s\in S}\,(c(e,i(e,s)) \wedge \forall_{e'\in E\wedge e'\neq e}(c(e',s) \Leftrightarrow c(e',i(e,s)))$

k2) $\forall_{e\in E}\,\forall_{s\in S}\,(\neg c(e,d(e,s)) \wedge \forall_{e'\in E\wedge e'\neq e}(c(e',s) \Leftrightarrow c(e',d(e,s)))$

k3) $\forall_{s\in S}(p(s) \Leftrightarrow \forall_{e\in E}\,\neg c(e,s))$

k4) $\forall_{s\in S}(\neg p(s) \Rightarrow c(m(s),s))$

k5) $\forall_{e\in E}\,\forall_{s\in S}\,(c(e,s) \Rightarrow \neg(e \leq m(s)))$

k6) $\forall_{e\in E}\,\forall_{s\in S}\;\; c(e,s) \Leftrightarrow$

```
{ block
    var bool : Boolean, s1 : S
  begin
    s1 := s; bool := false;
    while ¬bool ∧ ¬p(s1) do
      e1 := m(s1);
      bool := r(e1, e);
      s1 := d(e1, s1)
    od
  end } bool
```

k7) $\forall_{s\in S}\,\{\textbf{while}\ \neg p(s)\ \textbf{do}\ s := d(m(s),s)\ \textbf{od}\}\ p(s)$

k8) $\forall_{s,s'\in S}\,q(s,s') \Leftrightarrow \forall_{e\in E}(c(e,s) \Leftrightarrow c(e,s'))$

k9) $\forall_{e\in E}\;r(e,e)$

k10) $\forall_{e,e'\in E}\;r(e,e') \Leftrightarrow r(e',e)$

k11) $\forall_{e,e',e''\in E}\;(r(e,e') \wedge r(e',e'')) \Rightarrow r(e,e'')$

k12) $\forall_{e\in E}\;e \leq e$

k13) $\forall_{e,e'\in E}\;e \leq e' \wedge e' \leq e \Rightarrow r(e,e')$

k14) $\forall_{e,e',e''\in E}\;(e \leq e' \wedge e' \leq e'') \Rightarrow e \leq e''$

Koniec definicji kolejki priorytetowej.

18.2 Niesprzeczność, tw. o reprezentacji, modele

Ta specyfikacja kolejek priorytetowych jest niesprzeczna.

Twierdzenie 18.1. *Zbiór formuł k1) – k14) posiada model.*

Dowód. Dowód tego twierdzenia przebiega podobnie do dowodu twierdzenia 17.1. Niech E będzie dowolnym zbiorem, uporządkowanym przez relację $\leq_E$, niech $=_E$ oznacza relację identyczności w zbiorze E.

Rozpatrzmy zbiór $Fin(E)$ wszystkich skończonych podzbiorów zbioru E. Przyjmiemy następującą interpretację symboli sygnatury kontenera:

$i(e,s)$	$d(e,s)$	$m(s)$	$r(e,e')$	$e \leq e'$	$p(s)$	$c(e,s)$	$q(s,s')$
$s \cup \{e\}$	$s \setminus \{e\}$	$min(s)$	$e =_E e'$	$e \leq_E e'$	$s = \emptyset$	$e \in s$	$s =_S s'$

Przy takiej interpretacji symboli każda formuła ze zbioru k1) – k14) jest prawdziwa. □

Strukturę algebraiczną $E \cup Fin(E)$ z działaniami opisanymi w powyższej tabelce będziemy nazywać *standardowym modelem kolejek priorytetowych* wyznaczonym przez zbiór E.

Twierdzenie 18.2. *Każda kolejka priorytetowa $\mathcal{B}$ jest izomorficzna z standardowym modelem kolejek priorytetowych wyznaczonym przez zbiór elementów struktury $\mathcal{B}$.*

Dowód. Dowód przebiega w sposób podobny do dowodu twierdzenia 17.2. □

18.3 Zastosowania

Oba znane algorytmy budowania drzewa rozpinającego grafu: algorytm Kruskala i algorytm Prima wykorzystuja strukturę kolejki priorytetowej, algorytm Chartresa znajdowania kolejnej liczby pierwszej, algorytm Dijkstry znajdowania najkrótszj ścieżki w grafie, zarządzanie zdarzeniami w klasie Simulation,

Rozdział 19

Drzewa BST

Przyjmujemy, że dany jest zbiór E uporządkowany przez relację $<$. W praktyce programistycznej mamy najczęściej do czynienia ze zbiorem rekordów, takim, że klucze przypisane rekordom są uporządowane przez pewną relację zwrotną, przechodnią i antysymetryczną *less*.

Drzewa binarnych poszukiwań są znane od połowy XX wieku. Stosowanie drzew BST umożliwia szybkie wykonanie następujących operacji.

Każda implementacja drzewa BST pozwala zaimplementować kolejkę priorytetową.
PLAN

1. definicja (algebry) struktury drzew BST,
2. aksjomaty
3. model standardowy - S-wyrażenia
4. model klasa Node,
5. własności struktury BST
6. implementacja kolejek PQ

19.1 Struktura drzew BST?

Niech E oznacza zbiór uporządkowany przez relację $\leq_E$, zwrotną przechodnią i antysymetryczną. Równość $=_E$ elementów zbioru E ...

Definicja 19.1. *Strukturą drzew binarnych poszukiwań nazywamy system algebraiczny*

$$\mathcal{BST} = \langle E \cup BST \cup \{none\}; v, l, r, n, ul, ur; m, \leq, = \rangle$$

gdzie
v jest jednoargumentową operacją typu $(BST \to E)$
l, r są jednoargumentowymi operacjami typu $(BST \to \{BST \cup \{none\}\})$
n jest jednoargumentową operacją typu $(E \to BST)$
ul, ur są dwuargumentowymi operacjami typu $(BST \times BST \to BST)$
m jest funkcją charakterystyczną relacji $m\colon E \times BST \to B_0$

*Ponadto, dla dowolnego $e \in E$ i dowolnych drzew $n_1, n_2 \in BST$ prawdziwe są następujące własności (*AKSJOMATY*):*

bst1) $v(n(e)) = e \wedge l(n(e)) = none \wedge r(n(e)) = none$

bst2)
$$S: \left\{ \begin{array}{l} \textbf{begin} \\ \quad n1 := n; continue := true; \\ \quad \textbf{while } (\mathrm{n1} \neq \underline{none}) \wedge continue \\ \quad \textbf{do} \\ \quad\quad \textbf{if } \mathrm{e} = \mathrm{n1.val} \textbf{ then } continue := false \\ \quad\quad \textbf{else} \\ \quad\quad \textbf{if } e < n1.val \textbf{ then } n1 := n1.l \\ \quad\quad \textbf{else } n1 := n1.r \textbf{ fi} \\ \quad\quad \textbf{fi}; \\ \quad \textbf{done} \\ \textbf{end} \end{array} \right\} true$$

bst3)
$$m(e,n) \Leftrightarrow M: \left\{ \begin{array}{l} \textbf{begin} \\ \quad n1 := n; result := \underline{false}; \\ \quad \textbf{while} \neg result \wedge\ n1 \neq \underline{none} \\ \quad \textbf{do} \\ \quad\quad \textbf{if } e = n1.v \textbf{ then } result := \underline{true} \\ \quad\quad \textbf{else} \\ \quad\quad\quad \textbf{if } e < n1.v \textbf{ then } n1 := n1.l \\ \quad\quad\quad \textbf{else } n1 := n1.r \textbf{ fi} \\ \quad\quad \textbf{fi} \\ \quad \textbf{done} \\ \textbf{end} \end{array} \right\} result$$

bst4) $m(e, n.l) \Rightarrow e < v(n)$

bst5) $m(e, n.r) \Rightarrow v(n) < e$

bst6) $(n = n' \equiv (n = none = n') \vee (n.v = n'.v \wedge n.l = n'.l \wedge n.r = n'.r))$

bst7)
$$\big(n.r = n'' \wedge n.v = e \wedge (\left\{ \begin{array}{l} K: \textbf{begin} \\ \quad n2 := n'; \\ \quad \textbf{while } n2.r \neq none \\ \quad \textbf{do} \\ \quad\quad n2 := n2.r \\ \quad \textbf{done}; \\ \quad bol := n2.v < n.v \\ \textbf{end} \end{array} \right\} bol \vee n' = none)\big)$$
$$\Rightarrow \{n3 := ul(n', n)\}(n3.r = n'' \wedge n3.v = e \wedge n3.l = n')$$

bst8)
$$\big(n.l = n'' \wedge n.v = e \wedge (\left\{ \begin{array}{l} L: \textbf{begin} \\ \quad n2 := n'; \\ \quad \textbf{while } n2.l \neq none \\ \quad \textbf{do} \\ \quad\quad n2 := n2.l \\ \quad \textbf{done}; \\ \quad bol := n.v < n2.v \\ \textbf{end} \end{array} \right\} bol \vee n' = none)\big)$$
$$\Rightarrow \{n3 := ur(n', n)\}(n3.r = n' \wedge n3.v = e \wedge n3.l = n'')$$

bst9) *aksjomaty liniowego porządku* $\leq$ *i równości* $=$

bst10) *aksjomaty o Exception - do wymyslenia*

Będziemy także badać zbiór konsekwencji przyjętego układu aksjomatów.

Definicja 19.2. *Algorytmiczną teorią drzew binarnych poszukiwań nazywać będziemy zbiór konsekwencji wyznaczony przez zbiór formuł* $\mathcal{B}st = \{bst1, \ldots, bst10\}$.

$$\mathcal{ATBST} = \mathcal{C}(\mathcal{B}st).$$

Lemat 19.1. ***M*** *oznacza program występujący w aksjomacie bst2. Własność stopu tego programu jest twierdzeniem teorii* $\mathcal{ATBST}$.

$$\mathcal{ATBST} \vdash \mathbf{M}\,\mathrm{true}$$

Dowód. Łatwo zauważyć, że program **M** ma obliczenie skończone wtedy i tylko wtedy gdy obliczenie skończone ma program **S**. Z twierdzenia o pełności wynika istnienie dowodu dla formuły **M***true*. Można jednak łatwo wykazać istnienie dowodu na innej drodze. Zauważmy, że dla każdej liczby naturalnej $i \in N$ poniższa implikacja jest tautologią.

$$\left\{ \begin{array}{l} n1 := n; continue := true; \\ \left(\begin{array}{l} \textbf{if } (\mathrm{n1} \neq \underline{none}) \wedge continue \\ \quad \textbf{then} \\ \quad\quad \textbf{if } \mathrm{e} = \mathrm{n1.val} \textbf{ then } continue := false \\ \quad\quad \textbf{else} \\ \quad\quad \textbf{if } e < n1.val \textbf{ then } n1 := n1.l \\ \quad\quad \textbf{else } n1 := n1.r \textbf{ fi} \\ \quad\quad \textbf{fi}; \\ \quad \textbf{fi} \end{array} \right)^i \end{array} \right\} true \Rightarrow$$

$$\left\{ \begin{array}{l} n1 := n; result := false; \\ \left(\begin{array}{l} \textbf{if } (\mathrm{n1} \neq \underline{none}) \wedge \neg result \\ \quad \textbf{then} \\ \quad\quad \textbf{if } \mathrm{e} = \mathrm{n1.val} \textbf{ then } result := true \\ \quad\quad \textbf{else} \\ \quad\quad \textbf{if } e < n1.val \textbf{ then } n1 := n1.l \\ \quad\quad \textbf{else } n1 := n1.r \textbf{ fi} \\ \quad\quad \textbf{fi}; \\ \quad \textbf{fi} \end{array} \right)^i \end{array} \right\} true \qquad (19.1)$$

Dowód tego można przeprowadzić przez indukcję. Dla $i = 0$ mamy oczywiscie

$$\vdash \{n1 := n;\ continue := true\} true \Rightarrow \{n1 := n;\ result := false\} true$$

Załóżmy, że istnieje dowód dla implikacji 19.1... Zapiszmy jej schemat

czytelniej proszę

$$\vdash \{N;\ (if\ \gamma\ then\ M\ fi)^i\} true \Rightarrow \{K;\ (if\ \delta\ then\ L\ fi)^i\} true$$

Wykorzystując wyłącznie rachunek zdań sprawdzamy że tautologią jest też formuła postaci

$$\vdash \{N;\ (if\ \gamma\ then\ M\ fi)^{i+1}\} true \Rightarrow \{K;\ (if\ \delta\ then\ L\ fi)^{i+1}\} true$$

Stąd wnioskujemy, że dla każdego $i \in N$ jest tautologią formuła

$$\vdash \{N;\ (if\ \gamma\ then\ M\ fi)^{i+1}\}true \Rightarrow \{K;\ (while\ \delta\ do\ L\ od)\}true$$

czytelniej

W dowodzie wykorzystujemy aksjomat Ax21 logiki algorytmicznej 8.1. Wiedząc, że dla każdego $i \in N$ formuła o powyższej postaci jest tautologią możemy zastosować regułę R3 8.1 i otrzymujemy że implikacja **??** jest tautologią. Dowód kończy zastosowanie reguły odrywania R1. □

19.2 Modele. Tw. o niesprzeczności

Opisana powyżej teoria ATBST ma wiele modeli. Poniżej przytoczymy dwa takie modele. Sformułujemy i udowodnimy twierdzenia: o niesprzeczności zbioru aksjomatów bst1) – bst10) oraz twierdzenie o reprezentacji.

Model standardowy

Przykładem struktury drzew BST jest zbiór S-wyrażeń z odpowiednio określonymi operacjami.

Przykład 19.3. *Niech E będzie zbiorem uporządkowanym przez relację $<$. Zbiór S-wyrażeń nad zbiorem E jest to najmniejszy zbiór wyrażeń S taki, że*

e) dla każdego $e \in E$ napis $(()e())$ należy do zbioru S,

c) jeśli napisy t_1 i t_2 należą do zbioru S, $e \in E$ i ponadto największy element zbioru E występujący w napisie t_1 jest mniejszy od e oraz najmniejszy element zbioru E występujący w napisie t_2 jest większy od e, to napis $(t_1 e t_2)$ należy do zbioru S

W zbiorze S-wyrażeń określamy następujące działania

$$\begin{aligned}
new(e) &= (()e()) \quad dla\ dowolnego\ e \in E \\
val(t_1\, e\, t_2) &= e \\
left(t_1\, e\, t_2) &= t_1 \\
right(t_1\, e\, t_2) &= t_2 \\
upl(t, (t_1\, e\, t_2)) &= \begin{cases} (t\, e\, t_2) & wttw\ (t\, e\, t_2)\ jest\ elementem\ zbioru\ S \\ nieokreślony & w\ pozostałych\ przypadkach \end{cases} \\
upr(t, (t_1\, e\, t_2)) &= \begin{cases} (t_1\, e\, t) & wttw\ (t_1\, e\, t)\ jest\ elementem\ zbioru\ S \\ nieokreślony & w\ pozostałych\ przypadkach \end{cases}
\end{aligned}$$

jest strukturą drzew BST.

Zauważ, że każde wyrażenie ze zbioru BST mozna narysować jako drzewo.

rysunek

Przykład 19.4.

Twierdzenie 19.2. *(o niesprzeczności)*
Zbiór aksjomatów bst1) – bst8) posiada model.

Dowód. Rozważmy następująca interpretację

n	l	r	v	ul	ur
new	left	right	val	upl	upr

□

Strukturę opisaną powyżej nazywać będziemy modelem standardowym teorii BST.

Twierdzenie 19.3. *(o reprezentacji)*
Każdy model teorii BST jest izomorficzny z pewnym modelem standardowym.

Model obiektowy

W programach, struktura drzew binarnych poszukiwań jest implementowana na podstawie następującej lub podobnej deklaracji klasy.

```
unit Node: class(e: E);
  var l,r: Node
end Node
```

gdzie E jest nazwą pewnej klasy opisującej typ obiektów klasy E. Zakładamy, że deklaracja tej klasy zawiera dwie deklaracje funkcji definiujące: relację $\underset{E}{\leq}$ porządku w zbiorze obiektów typu E, oraz relację równości $\underset{E}{=}$ w tym samym zbiorze. Z tą niepozorną deklaracją klasy wiąże się następująca struktura danych

rys. obiektu Node

$$\langle E \cup Node, v, l, r, new, ul, ur, isnone, \underset{E}{\leq}, \underset{E}{=} \rangle$$

gdzie ...
Własności klasy Node są zbyt liberalne na nasz gust. Pozwalają na stworzenie dowolnego grafu. Zaradzimy temu modyfikując powyższą deklarację.
Zmodyfikowana klasa Node, modyfikator private (tj. hidden) nie pozwoli na dowolne manipulacje na atrybutach klasy node. Oferowane metody *ul* i *ur* zapewniają, że wynik modyfikacji atrybutu będzie znowu drzewem BST.

ALGORYTM 19.5.

```
unit Node: class(e: E);
  (* private *)hidden e,l,r;
  var l,r: Node
  unit cl : function: Node;
  begin
    result := l
  end cl;
  unit ul: procedure(n1: Node);
  begin
    if max(n1) < e then l:=n1 else raise SigA fi
  end ul;
  unit cr : function: Node;
  begin
    result := r
  end cr;
  unit ur: procedure(n1: Node);
  begin
    if min(n1) > e then r:=n1 else raise SigA fi
  end ur;
  unit max : function(n: Node): E;
    var n2: Node
  begin
    n2:=n;
    while n2.r ≠ none do n2 :=n2.r od;
    result := n2.e
  end max;
  unit min : function(n: Node): E;
    var n2: Node
  begin
    n2:=n;
    while n2.l ≠ none do n2 :=n2.l od;
    result := n2.e
  end min
end Node
```

Zadanie 19.1. *Czytelnik zechce sprawdzić, że wszystkie formuły bst1 – bst10) są prawdziwe przy takiej interpretacji.*

19.3 Rozszerzenie struktury BST

Strukturę BST wzbogacimy o kilka definicji, uzasadniając ich poprawność.

Definicja 19.6. *Operacja min zwraca najmniejszą wartość zapisaną w drzewie*

n.

$$min(n) \stackrel{df}{=} \left\{ \begin{array}{l} \textbf{if } n \neq none \\ \textbf{then} \\ \quad n1 := n \ ; \\ \quad \textbf{while } n1.l \neq none \textbf{ do } n1 := n1.l \textbf{ od} \\ \textbf{else raise SigRefToNone} \\ \textbf{fi} \end{array} \right\} n1.v$$

Lemat 19.4. *Dla dowolnego n takiego, że $n \neq none$, wartość $min(n)$ jest określona.*

Dowód. Dla dowodu wystarczy zaobserwować, że w dowolnym modelu teorii ATBST, każde obliczenie poniższego programu

$$\textbf{while } n1 \neq none \textbf{ do } n1 := n1.l \textbf{ od}$$

jest udane i skończone. Jest to konsekwencja twierdzenia o reprezentacji 19.3. Fakt ten można też udowodnić formalnie wyprowadzając formułę stopu z aksjomatu bst1. □

Lemat 19.4 pozwala nam bezpiecznie dodać deklarację funkcji *min* do pozostałych deklaracji klasy *Node*. Kolej na definicję operacji member (programiści czasami nazywają ją contains). Aksjomat bst3 jest definicją funkcji boolowskiej sprawdzającej czy element $e \in E$ należy do drzewa n.
Lemat 19.1 upewnia nas, że można w bezpieczny sposób posługiwać się tą definicją, czytaj operować funkcją member zadeklarowaną w klasie ...

Kolejny lemat pokazuje związek pojęć min i member.

Lemat 19.5. *Następująca formuła jest twierdzeniem teorii $\mathcal{ATBST}$*

$$\mathcal{ATBST} \vdash \forall_{n \in N} \forall_{e \in E} (member(e, n) \Rightarrow min(n) < e$$

i wobec tego formuła ta jest prawdziwa w każdym modelu algorytmicznej teorii $\mathcal{ATBST}$ drzew binarnych poszukiwań.

Opiszemy teraz dwie operacje porzednika *pred* i następnika *succ*.
Operacja $succ(n, r)$ dla zadanego węzła n drzewa o korzeniu r wyznacza węzeł n' taki, że przypisana mu wartość $n'.v$ jest najmniejszą z wartości większych od $n.v$, zapisanych w drzewie r.

$$succ(n, r) \stackrel{df}{=} n' \text{ takie, że } m(n'.v, r) \wedge n'.v = Min\{e : m(e, r) \wedge n.v < e\}$$

Podobnie definiujemy operację *pred*

$$pred(n, r) \stackrel{df}{=} n' \text{ takie, że } m(n'.v, r) \wedge n'.v = Max\{e : m(e, r) \wedge e < n.v\}$$

A oto algorytm $SUCC$ wyznaczania następnika. Zauważ, że wykorzystujemy funkcję $father$ zwracająca ojca danego węzła w drzewie, $father(root) = none$. Funkcję tę można z łatwością zaprogramować lub zapisać jej wykres w węzłach drzewa dodając pole $father$ obok pól $left, right$ i odpowiednio zmieniając algorytmy.

Algorytm 19.7.

$$SUCC: \left\{ \begin{array}{l} \textbf{if } n.right \neq none \\ \textbf{then} \\ \quad result := Min(n.right) \\ \textbf{else} \\ \quad y := father(n);\ x := n; \\ \quad \textbf{while } y \neq none \wedge x = y.right \textbf{ do} \\ \quad\quad x := y;\ y := father(y) \\ \quad \textbf{done}; \\ \quad result := y \\ \textbf{fi} \end{array} \right\} \tag{19.2}$$

Można sprawdzić, że algorytm ten poprawnie oblicza wartość funkcji $succ(n, r)$.

Lemat 19.6. *Jeśli n jest węzłem drzewa r, to po wykonaniu algorytmu SUCC zmienna result ma wartość równą $succ(n, r)$*

$$m(n.v, r) \Rightarrow \{SUCC\}(result = succ(n, r))$$

Dowód. Dowód nie jest trudny. Należy rozpatrzyć dwa przypadki:
a) gdy węzeł n ma prawe poddrzewo $n.right$, wtedy następnikiem jest węzeł zawierający najmniejszy element tego poddrzewa.
b) w przeciwnym wypadku następnikiem węzła n jest taki węzeł y, przodek węzła n, który jest najbliższy węzłowi n i taki, że y ma lewego syna. Tzn.

$$y = father^j(n) \text{ gdzie } j = \mu i(father^i(n) = y \wedge y.l \neq none)$$

zachodzi przy tym równość $n.v = max\{e : m(e, y.l) \wedge e > n.v\})$. □

Działanie wstawiania elementu e do drzewa o korzeniu n opisuje następujący algorytm $INSERT$.

Algorytm 19.8.

$insert(e, n) \stackrel{df}{=}$

$$M: \left\{ \begin{array}{l} n1 := n;\ bol := false;\ res := n1; \\ \textbf{while } \neg(n1 = none \vee bol) \textbf{do} \\ \quad n2 := n1; \\ \quad \textbf{if } e = n1.v \textbf{ then } bol := true \\ \quad \textbf{else} \\ \quad\quad \textbf{if } e < n1.v \textbf{ then } n1 := n1.l \textbf{ else } n1 := n1.r \textbf{ fi} \\ \quad \textbf{fi} \\ \textbf{done}; \\ \textbf{if } \neg bol \textbf{ then} \\ \quad aux := \textbf{new } Node(e); \\ \quad \textbf{if } n2 = none \textbf{ then } n1 := aux \textbf{ else} \\ \quad\quad \textbf{if } e < n2.v \textbf{ then } n2.l := aux \textbf{ else } n2.r := aux \textbf{ fi} \\ \textbf{fi} \end{array} \right\} n1 \tag{19.3}$$

Lemat 19.7. *Niech M będzie programem z poprzedniej definicji 19.8. Poniższa formuła jest prawdziwa w każdym modelu $\mathfrak{M}$ teorii $\mathcal{ATBST}$ drzew binarnych poszukiwań.*

$$\mathcal{ATBST} \models \forall_{n \in BST} \forall_{e \in E}(\{M\}\, m(e, n1) \wedge (\forall_{e' \neq e} m(e', n) \Leftrightarrow M\, m(e', n1)))$$

Dowód. Każde dwa modele o tym samym zbiorze elementów E są izomorficzne. Wystarczy więc, jeśli sprawdzimy, że formuła ta jest prawdziwa w jakimkolwiek modelu teorii $\mathcal{ATBST}$ (nie korzystając przy tym z żadnych założeń o zbiorze elementów E).
Możemy jednak postąpić inaczej. Udowodnimy, że formuła ta jest twierdzeniem teorii $\mathcal{ATBST}'$, jaka powstaje gdy do teorii $\mathcal{ATBST}$ dodajemy nowy symbol operacji *insert* i definiujący ją aksjomat tj. definicje 19.8.
Zauważmy, że $Strue \Rightarrow Mtrue$ jest tautologią.
Z kolei albo

$$((m(e,n) \wedge Mbol)$$

lub

$$(\neg m(e,n) \wedge M(n1 = none \wedge pred(n2).v < e < succ(n2) \wedge father(n1) = n2))$$

Czyli program M ustala czy element e należy do drzewa n i nadaje zmiennej *bol* odpowiednią wartość, ponadto, w przypadku gdy, po wykonaniu programu M zmienna *bol* ma wartość false, obliczona przez program M wartość zmiennej $n1$ wskazuje gdzie należy wstawić nowy węzeł $n(e)$ jako odpowiedniego syna węzła $n2$.

Rozpatrzmy formuły postaci ... □

Algorytm 19.9.

$delete(e, n) \stackrel{df}{=}$

```
Δ{ Λ| n1 := n; bol := false; res := n1;
   |  while ¬(n1 = none ∨ bol)do
   |    n2 := n1;
   |    if e = n1.v then bol := true
   |    else
   |      if e < n1.v then n1 := n1.l else n1 := n1.r fi
   |    fi
   |  done;
   |
   |  if bol
   |  then
   |    if e < n2.v then tolft := true else tolft := false fi;
   |    if n1.l = none ∧ n1.r = none
   |    then
   |      if tolft then n2.l := none else n2.r := none fi
   |    else
   |      if n1.l = none
   |      then
   |        if n1 = n
   |        then res := n1.r
   |        else
   |          if tolft then n2.l := n1.r
   |          else n2.r := n1.r fi
   |        fi
   |      else
   |        if n1.r = none                                          } res
   |        then (∗n1 has the left son only∗)
   |          if n1 = n
   |          then (∗found in the root∗)
Ψ|            res := n1.l
   |          else
   |            if tolft
   |            then n2.l := n1.l
   |            else n2.r := n1.l
   |            fi
   |             (∗n2 ommited n1 to son∗)
   |          fi
   |        else {n1 has two sons}
   |          n4 := n1.r;
   |          while n4.l ≠ none
   |          do
   |            n5 := n4;
   |            n4 := n4.l
   |          done (∗ n4.v = min(n1.r) ∗)
   |          n5.l := n4.r;
   |          n1.v := n4.v (∗ i.e. n1.v ← min(n4.r) ∗)
   |        fi(∗ n1.r = none? ∗)
   |      fi(∗ n1.l = none ∗)
   |    fi(∗ n1.l = none ∧ n1.r = none ∗)
   |  fi(∗bol∗)
```

Lemat 19.8. *Dla każdego elementu e i dla każdego drzewa BST n, program* Δ *występujący w definicji działania delete kończy obliczenie.*

Dowód. Dla dowodu wystarczy zauważyć, że wykonywanie (jedynej) instrukcji Λ zawierającej instrukcję while, która występuje w tym programie kończy się po skończonej liczbie kroków.
Można to wyprowadzic formalnie z aksjomatu bst2 (por. dowód lematu 19.1) □

Lemat 19.9. *Dla każdego elementu e i dla każdego drzewa BST n po wykonaniu instrukcji* $\{n1 := n; bol := false; res := n1; \Lambda\}$ *zachodzi dokładnie jedna z poniższych formuł*

i) $\neg bol$ *gdy element e nie występuje w drzewie n,*

lub

ii) $bol \wedge n1.v = e \wedge (n2.left = n1 \vee n2.right = n1)$

Dowód. Dowód pozostawiamy czytelnikowi □

Następny lemat stwierdza poprawnośc algorytmu Ψ względem warunku początkowego α : $bol \wedge n1.v = e \wedge (n2.left = n1 \vee n2.right = n1)$ i warunku końcowego β : $\neg m(e, n) \wedge ...$

Lemat 19.10.

$$\mathcal{ATBST} \vdash \alpha \Rightarrow \Delta\beta$$

Dowód. □

Łącząc powyższe spostrzeżenia stwierdzamy, że aksjomat k2) kolejek priorytetowych jest twierdzeniem algorytmicznej teorii drzew BST $\mathcal{ATBST}_{ext}$ wzbogaconej o zestaw definicji *declar*

jeśli zinterpretować nazwy ...

Lemat 19.11.

$$\mathcal{ATBST}_{ext} \vdash \forall_{e \in E} \forall_{s \in S} (\neg c(e, d(e, s)) \wedge \forall_{e' \in E \wedge e' \neq e} (c(e', s) \Leftrightarrow c(e', d(e, s))))$$

Udowodnić lemat stanowiący, że program S zawsze kończy obliczenie tj. bst2.

19.4 Implementacja kolejek priorytetowych

W tej sekcji udowodnimy, poprawność implementacji kolejek priorytetowych, jaka opisana jest w poniższej klasie PQS.
Poniżej prezentujemy moduł klasy PriorityQueues, który implementuje strukturę kolejek priorytetowych na bazie klasy Node realizującej strukturę drzew binarnych poszukiwań. Dyskusja przeprowadzona w poprzednim podrozdziale pozwala nam zapewnić użytkownika o poprawności tej implementacji. Udowodniliśmy bowiem, ze kazdy aksjomat kolejek priorytetowych jest prawdziwy w strukturze obiektów typu Node.

klasa

```
unit PriorityQueues : class (type E; function less(e, e' : E) : Boolean);
(* the class PriorityQueues implements a family of abstract data types.
A data type is determined by its class of elements and an ordering relation less.
User! make sure that less fulfills the axioms of order *)
  unit node : class (v : E);
    variable l, r : node;
  end node;

  unit min : function (n : node) : E;
  begin
    while n.l ≠ none do n := n.l od;
    result := n.v
  end min;

  unit member : function (e : E, n : node) : Boolean;
    variable n1 : node, bool : Boolean;
  begin
      n1 := n;
      bool := false;
      while n1 ≠ none ∧ ¬ bool do
          if n1.v=e then
            bool := true
          else
            if less(e, n1.v) then
              n1 := n1.l
            else
              n1 := n1.r
            fi
          fi
      od;
      result := bool
  end member;


  unit empty : function (n : node) : Boolean :
  begin
    result := (n = none)
  end empty;
```

```
unit insert function (e : E, n : node) : node;
   variable n1, n2, n3 : node, bool : Boolean;
begin
  n1 := n; n3 := n; bool := false;
  while ¬ n1 = none ∧¬ bool
  do
    n2 := n1;
    if e = n1.v then bool := true
    else
       if less(e, n1.v) then n1 := n1.l else n1 := n1.r fi
    fi
  od;
  if ¬ bool
  then
     aux:=new node(e);
     if n3 = none then n3 := aux
     else if less(e, n2.v) then n2.l := aux else n2.r := aux fi
     fi
  fi;
  result := n3
end insert;



unit delete : function(e : E, n : node) : node;
variable n1, n2, n3, n4, n5 : node, bool1, leftson : Boolean;
begin
n1 := n; n3 := n; bool1 := false;
(* search e *)
while ¬ n1 = none ¬ bool1
do
n2 := n1;
if e = n1.v
then
bool1 := true
else

if less(e, n1.v)
then
n1 := n1.l
else
n1 := n1.r
fi
fi
od

if bool1
then e is found in n1, n2 is the father of n1
leftson := less(e, n2.v)
leftson n1 is the left son of n2;
```

```
if n1.l = none n1.r = none
then n1 is a leaf
if leftson then n2.l := none
else n2.r := none fi;
else n1 is not a leaf
if n1.l = none
then n1 has the right son only
if n1 = n
then
n3 := n1.r
else
if leftson
then
n2.l := n1.r
else
n2.r := n1.r
fi (* since now n2 is the father of the only son of n1 *)
fi n = n1?
else n1 has the left son only
if n1.r = none
then n1 has the left son only
if n1 = n
then found in the root
n3 := n1.l
else
if leftson
then
n2.l := n1.r
else
n2.r := n1.r
fi (* since now n2 is the father of the only son of n1 *)
fi
else n1 has two sons
n4 := n1.r;
while n4.l none
do
n5 := n4;
n4 := n4.l
od; (* n4 is the least element in the left subtree of n1 *)
n5.l := n4.r; (* i.e. we omitted the n4 node *)
n1.v := n4.v (* the value found in the n4 is put in n1 node *)
fi n1.r = none?
fi n1.l = none?
fi n1.l = none n1.r = none?
fi bool1?
result := n3
end delete

end PriorityQueues
```

Taki moduł klasy PriorityQueues implementuje strukturę kolejki priorytetowej wyznaczonej przez pewien typ T i zadaną w nim relację porządku *less*. Jeśli więc mamy klasę o strukturze podanej poniżej i towarzyszącą jej funkcję

```
unit T: class ...
   virtual unit equal: function(x:T): Boolean;

   end equal;
end T;
unit less: function(x:T): Boolean;
...
end less;
```

Przy czym funkcja less zapewnia prawdziwość następujących warunków:

s) $\forall x, y \; in \; T(less(x,y) \vee less(y,x))$

z) $\forall x \; in \; T less(x,x)$

p) $\forall x, y, z \; in \; T((less(x,y) \wedge less(y,z)) \Rightarrow less(x,z))$

a) $\forall x, y \; in \; T((less(x,y) \wedge less(y,x) \Rightarrow x.equal(y))$

a funkcja *equal* spełnia trochę inny zestaw warunków.

Zadanie 19.2. *Napisz te warunki.*

I masz ponadto blok, w którym zapisano pewien algorytm abstrakcyjny, posługujący się zmiennymi typu T i zmiennymi typu *node* oraz działaniami *insert*, *member*, *delete*, *min* etc.
Na przykład blok

$APr:$

```
block
   var e,e1:E, n,n1,n2:node
begin
   e:= new T(...); ???
   n := new node(e);
   n1 := insert(e1,n);
...
end
```

To łącząc te cztery moduły: T, less, PriorityQueues i APr w jedną instrukcję bloku prefiksowanego klasą otrzymujemy pożyteczny(?) program

```
pref PriorityQueues(T,less) block
   var e,e1:T, n,n1,n2:node
begin
   e:= new T(...);
   n := new node(e);
   n1 := insert(e1,n);
...
end
```

Powyższa instrukcja bloku

A module which implements PriorityQueues can be used many times for different abstract programs. In order to do so one should prefix the block containing the abstract program with the following line

pref PriorityQueues (an actual type E, an actual procedure of comparison)
block
...an abstract program, see.section.5.1
end of prefixed block, see.[9].

Adding just one line which fixes the name of the implementing module and its actual parameters causes that the operations of data structure of priority queues are realized in accordance with the meaning given by the implementing module PriorityQueues.

Rozdział 20

Kopce

Struktura kopców znajduje wiele zastosowań. Omówimy parę ważnych przykładów: algorytm sortowania przez kopcowanie *heapsort* i zastosowanie kopców do efektywnej implementacji struktury kolejek priorytetowych gwarantującej pesymistyczny czas operacji $O(log\ n)$.

20.1 Definicja

Czym jest kopiec? Student udzieli szybkiej odpowiedzi - kopiec to drzewo binarne, doskonałe, w którym każda ścieżka jest uporządkowana niemalejąco. Bardzo dobra odpowiedź. A jak to przetłumaczyć na język programistów? Poszukujemy definicji podobnej do definicji stosów, kolejek, drzew BST. Trzeba określić z jakimi operacjami mamy do czynienia i jakie mają one właściwości.

Definicja 20.1. *Struktura algebraiczna* $\mathcal{K}$

$$\mathcal{K} = \langle E \cup N \cup K \cup \{none\}; r, o, l, p, f, v, \leq, m \rangle$$

jest strukturą kopców wtedy i tylko wtedy gdy wyliczone powyżej funkcje mają liczbę i typy argumentów podane w poniższym zestawieniu

$$\begin{array}{lll} r\colon K \to N, & o\colon K \to N, & l\colon N \to N, \\ p\colon N \to N, & f\colon N \to N, & v\colon N \to E, \\ \leq\colon E \times E \to B_0, & m\colon N \times K \to B_0, & \end{array}$$

Ponadto prawdziwe są własności k1) – k11), *zob. poniżej.* popraw

k1) $\forall_{k \in K}\ \{n := o(k);$ **while** $n \neq r(k)$ **do** $n := f(n)$ **od**$\}(n = r(k))$

k2) $\forall_{k \in K} \forall_{n \in N}\ m(n,k) \Leftrightarrow \{n' := n;$ **while** $n' \neq r(k)$ **do** $n' := f(n')$ **od**$\} true$

k3) $\forall_{k \in K}\ (n = o(k)) \Rightarrow (l(n) = none \wedge p(n) = none \wedge (nie\ istnieje\ na\ prawo))$

k) $\forall_{k \in K} \forall_{n \in N}\ \Big(m(n,k) \Rightarrow \left\{ \begin{array}{l} n1 := n;\ n2 := o(k);\ dl := true; \\ \textbf{while}\ n1 \neq r(k) \wedge n2 \neq r(k)\ \textbf{do} \\ \quad n1 := f(n1);\ n2 := f(n2); \\ \quad \textbf{if}\ n1 \neq r(k) \wedge n2 = r(k)\ \textbf{then}\ dl := false\ \textbf{fi} \\ \textbf{od} \end{array} \right\} dl \Big)$

k4) $\forall_{n,n' \in N}\, f(n') = n \Rightarrow (l(n) = n' \vee p(n) = n')$

k5) $\forall_{n,n' \in N}\, l(n') = n \Rightarrow f(n) = n'$

k6) $\forall_{n,n' \in N}\, p(n') = n \Rightarrow f(n) = n'$

k7) $\forall_{k \in K} \forall_{n,n' \in N}\, (n = f(n') \wedge n \neq f(o(k))) \Rightarrow (l(n) \neq none \wedge p(n) \neq none)$
$\forall_{k \in K} \forall_{n,n' \in N}\, (n \neq o(k)) \Rightarrow (l(f(n)) \neq none \wedge p(f(n)) \neq none)$

k8) $\forall_{k \in K} \forall_{n \in N}\, (n \neq r(k) \wedge m(f(n), k)) \Rightarrow (v(f(n)) \leq v(n))$

k9) aksjomaty porządku $\leq$, zwrotność

k10) przechodniość

k11) antysymetria

k1y) $sub(n, n') \stackrel{df}{=} \{n1 := n;\ while\ n1 \neq n'\ do\ n1 := f(n1)\ od\}(n1 = n')$

k1z) $onleft(n, n') \stackrel{df}{=} \exists h(sub(n, h.left) \wedge sub(n', h.right))$

Pytanie. Czy te aksjomaty posiadają model? a może są sprzeczne? Odpowiedzi na to pytanie udziela następujące

Twierdzenie 20.1. *Powyższy układ aksjomatów posiada model, a więc jest niesprzeczny.*

Dowód. Powszechnie znaną realizacją kopca jest tablica. O strukturze E elementów nie musimy niczego zakładać, przyjmijmy więc że elementami są liczby naturalne z ich naturalnym porządkiem. Kopcem k będzie więc tablica A liczb naturalnych. Węzłami będą zmienne – elementy tablicy $A[1], \ldots, A[n]$. Interpretujemy nazwy działań w następujący sposób

$r(k)$	$o(k)$	$l(A[i])$	$p(A[i])$	$f(A[i])$	$v(A[i])$	$\leq$	$m(A[i], A)$
$A[1]$	$A[n]$	$A[2*i]$	$A[2*i+1]$	$A[i \div 2]$	$val(A[i])$	$\leq$	$1 \leq i \leq n$

Dokładniej, korzeniem kopca k jest zmienna $A[1]$, wartością funkcji $o(k)$ jest zmienna $A[n]$. Wartości funkcji l, p, f są opisane w powyższej tabelce, z następującym zastrzeżeniem, jeśli wartość wyrażenia $2*i, 2*i+1, i \div 2$ wykracza poza zakres $1 \ldots n$, to przyjmujemy, że wartość funkcji jest none. Wartością wyrażenia $v(A[i])$ jest liczba przypisana zmiennej $A[i]$. Po kolei sprawdzamy prawdziwość każdego aksjomatu w tej interpretacji i przekonujemy się, że wszystkie własności k1) – k12) są prawdziwe.
A więc istnieje model teorii $\mathcal{ATK}$ algorytmicznej teorii kopców i wobec tego, teoria ta jest niesprzeczna. □

Niech tablica A będzie odpowiednio duża tj. $lower(A) \lneq l \leq u \lneq upper(A)$. Warto zadać sobie pytanie: czy fragment tej tablicy $A[l], \ldots, A[u]$ jest kopcem?

Zadanie 20.1. *Zastanów się nad następującymi zadaniami. Odpowiedzi okażą się pomocne w zrozumieniu następnego podrozdziału.*

1. *Sformułuj warunki niezbędne na to byśmy mogli taki fragment tablicy A traktować jako kopiec.*

2. *Załóżmy, że fragment $A[l], \ldots, A[u]$ tablicy A jest lasem kopców. Rozważmy fragment $A[l-1], A[l], \ldots, A[u]$. Co można o nim powiedzieć? Co trzeba poprawić w tym fragmencie tablicy?*

3. *Przyjmijmy to samo założenie co poprzednio. Rozpatrujemy teraz fragment $A[l], \ldots, A[u], A[u+1]$. Czy można o nim powiedzieć, że jest kopcem? Co trzeba poprawić?*

Kolejne pytanie jakie staje przed nami, brzmi: czy każda struktura kopców jest izomorficzna ze strukturą kopców w tablicach? Na to pytanie udziela odpowiedzi następujące

Twierdzenie 20.2. *(o reprezentacji) Każda struktura kopców $\mathcal{K}$ jest izomorficzna ze strukturą $\mathcal{KT}$ kopców zapisanych w tablicach o tym samym zbiorze elementów E.*

Uwaga. Powinniśmy odróżniać strukturę kopców od zbioru kopców.
Powyższe aksjomaty opisują obiekty, które kwalifikują się do nazwy kopiec. Ale na kopcach wykonywać można operacje, które zwracają inne kopce.
Trzeba to podkreślić i uzupełnić. **Koniec uwagi**

20.2 Kopce w tablicach

Tu pokażemy jak można wykorzystać dziedziczenie by "*wyciągnąć wspólną część algorytmu przed nawias*".

20.2.1 Wstęp

Antoni Kreczmar zaprogramował w Loglanie algorytm heapsort wykorzystując możliwość dziedziczenia klasy przez procedurę.

Zacznijmy od przypomnienia czym jest kopiec.

Definicja 20.2. *Kopiec jest to drzewo binarne wyważone, tzn. takie że dla pewnej liczby naturalnej k, każdy liść tego drzewa jest na poziomie k lub k-1 i ponadto dla każdego węzła n tego drzewa wartość wpisana w tym węźle jest nie mniejsza niż wartości znajdujące sie w poddrzewie węzła n i jeśli węzeł n tego drzewa ma syna to jego lewy brat ma dwu synów.* popraw

W naszym przypadku kopiec będzie zapisany w tablicy.

Lemat 20.3. *Niech A bedzie tablicą n-elementową $A[1], A[2], \ldots, A[n]$. Jeżeli dla każdego k, $1 \leq k \leq n$ zachodzi $A[k] \leq A[k div 2] \leq A[k div 4] \leq \cdots \leq A[1]$ to tablica A reprezentuje kopiec.*

Dowód. Łatwo zauważyć, że tablica A może być pojmowana jako drzewo binarne. Korzeniem drzewa jest element $A[1]$. Dla każdego $i, 1 \leq i \leq n$, element $A[i]$ jest ojcem synów: lewego $A[2i]$ oraz prawego $A[2i+1]$, pod warunkiem, że elementy te istnieją w tablicy A tzn. że $2i \leq n$ i odpowiednio $2i+1 \leq n$. Nietrudno zauważyć, że drzewo jest wyważone. Warunek wymieniony w tezie lematu gwarantuje, że dla każdego węzła $A[l]$ wszystkie węzły poddrzewa $A[l]$ zawierają elementy nie mniejsze od $A[l]$. □

20.2.2 Konstruowanie algorytmu

Zacznijmy od analizy następującego kodu

Kopcuj:

```
i :=l; j := 2*i; x :=A(i);
while j<=p do
  if j<p and A(j)<A(j+1)
  then
    j:=j+1
  fi;
  if x>=A(j) then exit fi;
  A(i):=A(j); i:=j; j:=2*i;
done;
A(i) := x;
```

Niech dane będą warunki: warunek wstępny
α: *tablica A na miejscach* $l+1,\ldots,p$ *reprezentuje las kopców*,
oraz warunek końcowy
β: *tablica A na miejscach* $l, l+1,\ldots,p$ *reprezentuje las kopców.*

Lemat 20.4. *Program Kopcuj jest poprawny ze względu na warunek początkowy* α *i warunek końcowy* β

$$\alpha \Rightarrow \{\text{Kopcuj}\}\,\beta.$$

Dowód. Gdy $j > p$ to $A[l]$ jest liściem, $A[l]$ jest też korzeniem. A więc fragment tablicy $A[l], A[l+1],\ldots,A[p]$ jest lasem kopców. Rozważmy przypadek gdy $j \leq p$. Zacznijmy od obserwacji, że po wykonaniu instrukcji
if j<p and_if A[j] < A[j+1] then j := j+1 fi
zachodzi $A[j] = max\{A[2i], A[2i+1]\}$. W przypadku gdy x >A[j] to z założenia indukcyjnego x jest większe od wszystkich węzłów poddrzewa A[j], a więc i od wszystkich węzłow poddrzewa A[j-1]. Nie ma nic więcej do roboty i opuszczamy (exit) petlę. Jeśli tak nie jest to kładąc A[i]:=A[j]; i:= j; j:=2*i; sprowadzamy nasz problem do problemu przesiewania w poddrzewie drzewa początkowego. Po pewnej liczbie powtórzeń algorytm zatrzyma się ponieważ kolejne drzewo będzie liściem. □

Rozpatrzmy teraz następujący kod

BS:

```
l:= (upper(A) div 2)+1;
do
  l:= l-1;
  if l=lower(A) then exit fi;
  Kopcuj
od
```

Lemat 20.5. *Zakładam, że lower(A)=1.Program BS buduje kopiec z elementów tablicy A.*

Dowód. Po wykonaniu instrukcji l:=(upper(A)div 2)+1 fragment tablicy A: A[l], A[l+1], ...,A[upper(A)] jest lasem drzew. W każdej iteracji pętli do ...od powiększamy fragment zawierający las kopców. Powtarzanie zakończy się gdy l=lower(A). Wtedy las kopców jest jednym kopcem o korzeniu A[1]. □

Teraz rozważmy kod

ST:
```
p:= upper(A); l:=lower(A);
do
   x:=A[l]; A[l]:=A[p]; A[p]:= x;
   p:=p-1;
   if p=lower(A) then exit fi;
   Kopcuj
od
```

Lemat 20.6. *Jeśli tablica A jest kopcem, to program ST sortuje elementy tablicy A w porządku niemalejącym.*

Dowód. W pierwszym kroku iteracji zamieniamy miejscami A[l] i A[p], ponadto zmniejszamy wartość p. Ostatnim elementem tablicy A jest więc element największy. Gdy l=p to tablica A jest uporządkowana niemalejąco i można zakończyć działanie programu ST. W przeciwnym przypadku może być tak, że fragment A[l], ..., A[p] tablicy A nie jest kopcem. Ale po wykonaniu polecenia Kopcuj fragment ten będzie kopcem. Kolejny co do wielkości element trafi na miejsce p w tablicy A. Proces ten kończy się i z chwilą jego zakończenia tablica A jest uporządkowana niemalejąco. □

Co dalej? Jak skonstruować procedurę heapsort?
Można zaproponować trzy rozwiązania:

- Treścią procedury mogą stać się programy BS i ST, w których powtarza się program Kopcuj.
- Treść procedury może polegać na wywołaniu po kolei procedur BudujKopiec i Sortuj, każda z tych procedur wywołuje procedurę Kopcuj.
- Na treść procedury heapsort składają się klasa Przesiej, procedura BudujKopiec rozszerzająca klasę Przesiej i procedura Sortuj też rozszerzająca klasę Przesiej.

Zestawmy te trzy możliwości obok siebie

```
unit heapsortA: procedure
      <deklaracje>
begin
      BS;
      ST
end heapsortA
```

```
unit heapsortB: procedure
      unit Kopcuj: procedure
      end Kopcuj
      unit BudKop: procedure
      ... call Kopcuj ...
      end BudKop
      unit Sortuj: procedure
      ... call Kopcuj ...
      end Sortuj
begin
      call BudKop;
      call Sortuj;
end heapsortB
```

```
unit heapsortC: procedure
      unit Przesiej:
        class ...
      unit BudKop:
        Przesiej procedure
      unit Sortuj:
        Przesiej procedure
begin
      call BudKop;
      call Sortuj;
end heapsortC
```

Każda z tych propozycji pozwoli skonstruować poprawną procedurę heapsort.

Która droga prowadzi do najlepszego rozwiązania? Procedura heapsortB ma największy koszt. Podczas wykonywania instrukcji BudKop i Sortuj instrukcja call

Kopcuj będzie wykonywana wielokrotnie. Wielokrotnie trzeba będzie tworzyć rekord aktywacji tej procedury. Procedura heapsortC pozwala uniknąć tego narzutu. Zobaczymy jak. Ale najbardziej wydajna jest chyba procedura heapsortA. Tyle, że wymaga to od nas dwukrotnego napisania kodu Kopcuj, wewnatrz programu BS i wewnątrz programu ST. Nie ma w tym nic złego tak długo jak długo nie trzeba wprowadzać zmian w kodzie Kopcuj – musimy tylko pamiętać, że trzeba to zrobić w dwu miejscach!

Dokończmy konstrukcję procedury heapsortC. Tworzymy klasę Przesiewanie.

```
unit Przesiewanie: class;
    var koniec: boolean
begin
    do
        inner ;
        if koniec then exit fi;
        i :=l; j := 2*i; x :=A(i);
        while j<=p do
            if j<p and A(j)<A(j+1)
            then
                j:=j+1
            fi;
            if x>=A(j) then exit fi;
            A(i):=A(j); i:=j; j:=2*i;
        done;
        A(i) := x;
    done
end przesiewanie;
```

i procedury dziedziczące klasę Przesiewanie: budujKopiec i sortuj.

```
unit budujKopiec: Przesiewanie procedure;
begin
        koniec := (l=lower(A)); l :=l-1;
end budujKopiec;
```

Nie trzeba długo dowodzić, że prawdziwą treść procedury budujKopiec stanowi treść klasy Przesiewanie w której pseudoinstrukcję inner zastąpiono dwoma instrukcjami z deklaracji procedury budujKopiec. czyli należy pamiętać, że dzieki regule konkatenacji powinniśmy przyjąć, że deklaracja procedury budujKopiec to

```
unit budujKopiec: procedure;
    var koniec: boolean
begin
    do
        koniec := (l=lower(A)); l :=l-1;
        if koniec then exit fi;
        i :=l; j := 2*i; x :=A(i);
        while j<=p do
            if j<p and A(j)<A(j+1)
            then
                j:=j+1
            fi;
            if x>=A(j) then exit fi;
            A(i):=A(j); i:=j; j:=2*i;
        done;
        A(i) := x;
    done
end budujKopiec;
```

Wcześniej przekonaliśmy się, że ten algorytm przekształca tablicę A w kopiec. Podobnie można sprawdzić, że instrukcja call sortuj uporządkuje tablicę A w ciąg niemalejacy.

```
unit sortuj: Przesiewanie procedure;
begin
    d:= lower(A); koniec:=(p=d);
    x:=A(d); A(d) := A(p); A(p) := x;
    p := p-1;
end sortuj;
```

Po przyjęciu tych trzech deklaracji łatwo uzupełnić treść procedury heapsortC tak jak to widać poniżej.

```
unit heapsortC: procedure(A: arrayof integer);
    var i, j, l, p, x: integer;
    unit Przesiewanie: class ...
    unit budujKopiec: Przesiewanie procedure ...
    unit sortuj: Przesiewanie procedure ...
begin
    l := upper(A) div 2 +1;
    p := upper(A);
    call budujKopiec;
    call sortuj
end heapsortC
```

20.2.3 Zadania

1. Napisz kompletny program, który wczytuje zadany ciąg liczb i sortuje go.
2. Napisz program zawierający procedury heapsortB i heapsortC. Tworzy (dość dużą) tablicę liczb, np. wybierając liczby pseudolosowe i porządkuje ją dwukrotnie, raz przy pomocy heapsortB i drugi raz przy pomocy heapsortC. Porównaj czasy działania tych algorytmów.

3. Czy potrafisz zmienić procedurę heapsort tak by porządkowała tablice elementów typu T?

20.2.4 Kompletny algorytm

Poniżej przytaczamy kompletny tekst procedury heapsort (w wersji C). Z poprzednich rozważań wynika, że algorytm ten poprawnie porządkuje zadaną tablicę A.

```
unit heapsort: procedure(A: arrayof integer);

    var i,j,l,d,p,x: integer

    unit Przesiewanie: class;
        var koniec: boolean
    begin
        do
            inner ;
            if koniec then exit fi;
            i :=l; j := 2*i; x :=A(i);
            while j<=p do
                if j<p and A(j)<A(j+1)
                then
                    j:=j+1
                fi;
                if x>=A(j) then exit fi;
                A(i):=A(j); i:=j; j:=2*i;
            done;
            A(i) := x;
        done
    end Przesiewanie;

    unit budujKopiec: Przesiewanie procedure;
    begin
        koniec := (l=lower(A)); l :=l-1;
    end budujKopiec;

    unit sortuj: Przesiewanie procedure;
    begin
        d:= lower(A); koniec:=(p=d);
        x:=A(d); A(d) := A(p); A(p) := x;
        p := p-1;
    end sortuj;

   (* tu zaczynają się instrukcje procedury heapsort *)
begin
    l:= upper(A) div 2 +1;
    p:= upper(A);
    call budujKopiec;
    call sortuj;
end heapsort;
```

20.3 Czy to jest kolejka priorytetowa

Tu pokażemy jak sprawdzić czy zadana klasa K spełnia specyfikację S i dowieść swojej racji.

20.3.1 Czy to jest kolejka priorytetowa?

Poniżej znajdziesz pełny tekst klasy PQS napisanej wiele lat temu przez W.M. Bartol i D. Szczepańską. Jest to część większego programu symulacji pracy oddziału bankowego [27] [28].

```
unit PQS : class; (* priority queues system *)
    unit PQ: class;
        var last,root:node;
        unit min: function: elem;
        begin
            if root=/= none then result:=root.el else raise ReftoNone fi;
        end min;
        unit insert: procedure(r:elem);
            var x,z:node;
        begin
            x:= r.lab;
            if last=none then
                root:=x; root.left, last:=root
            else
                if last.ns=0 then
                    last.ns:=1; z:=last.left; last.left:=x;
                    x.up:=last; x.left:=z; z.right:=x;
                else
                    last.ns:=2; z:=last.right; last.right:=x;
                    x.right:=z; x.up:=last; z.left:=x;
                    last.left.right:=x; x.left:=last.left; last:=z;
                fi
            fi;
            call correctUp(r)
        end insert;
        unit delete: procedure(r: elem);
            var x,y,z:node;
        begin
            x:=r.lab; z:=last.left;
            if last.ns =0 then
                y:= z.up;
                if y=none then root:=none else y.right:= last fi;
                last.left:=y; last:=y;
            else
                y:= z.left; y.right:= last; last.left:= y;
            fi;
            z.el.lab:=x; x.el:= z.el; last.ns:= last.ns-1;
            r.lab:=z; z.el:=r;
            (* the following three instructions were added during our verification *)
            z.left.right :=none; z.ns:=0; z.left, z.right, z.up := none;
            if x.less(x.up) then
                call correctUp(x.el)
            else
                call correctDown(x.el)
            fi;
        end delete;
        unit correctDown: procedure(r: elem);
            var x,z: node, t: elem, fin, log: Boolean;
        begin
            z := r.lab;
            while not fin
            do
                if z.ns = 0 then fin :=true
                else
                    if z.ns=1 then x := z.left
                    else
                        if z.left.less(z.right) then x:= z.left else x:=z.right fi
```

```
                fi;
                if z.less(x)
                then
                    fin := true
                else
                    t:= x.el; x.el :=z.el; z.el:=t;
                    z.el.lab :=z; x.el.lab:=x
                fi;
            fi;
            z:=x;
        od
    end correctDown;
    unit correctUp: procedure(r: elem);
        var x,z: node, t: elem, fin, log: Boolean;
    begin
        z := r.lab;
        x:= z.up;
        do
            if x=none then log:=true else log:=x.less(z) fi;
            if log then exit fi;
            t:=z.el; z.el:=x.el; x.el:=t;
            x.el.lab:=x; z.el.lab:=z; z:=x; x:z.up;
        od
    end correctUp;
end PQ;

unit node: class (el:elem);
    var left,right,up: node, ns:integer;
    unit less: function(x:node): boolean;
    begin
        if x= none then
            result:=false
        else
            result:=el.less(x.el)
        fi;
    end less;
end node;
unit elem: class(prior:real);
    var lab: node;
    unit virtual less: function(x:elem):boolean;
    begin
        if x=none then
            result:= false
        else
            result:= prior ≤ x.prior
        fi;
    end less;
begin
    lab:= new node(this elem);
end elem;
end PQS (* priority queues system *);
```

20.3.2 Introduction

This paper presents a proof that the class PQS is a correct implementation of priority queues data structure. The class (see Appendix) is written in an object-oriented programming language. The specification (see Table 1) consists of a few algorithmic formulas. Formally, the proof of correctnes of the class w.r.t. the specification resembles a proof that a given algebraic structure $\mathbb{A}$ is

a *model* of some axiomatic theory $\mathcal{T}$. Literature knows many examples of the proofs of correctnes of algorithms with respect to their pre- and post-conditions. However, the union of proofs of a class' methods (i.e. algorithms) does not result in the proof of the class correctness.

Each class can be considered as a description of an algebraic structure (a data structure). The universe of the structure is the set of all potential objects of the class, its functions and relations are defined by class methods.

Dealing with a class we are confronted with several properties, sometimes called invariants of the class (B. Meyer in Eiffel [20]), that must be valid for all objects of the class. Moreover, an external characterization of the class requires some properties that express corellations between different methods. All these properties will be called a specification of a class.

Our message has several layers:

- first, we offer a discussion of the methods of software's verification.
- second, we propose to think what a software's verification is,
- finally, we need to know what a class specification looks like?

Two views and two practices are colliding in production of software. One view is that programs should be accompanied by solid arguments demonstrating the correctness and the completeness of software. The practice associated with this view consist in proving properties of software. At present, we observe that there are more and more cases when the industry demands the proofs of correctness of programs. It is especially true of these cases where security must be guaranteed. Another view is followed by the majority of the individual programmers and the big software companies. They are of the opinion that testing of programs is the sufficient activity before the software is delivered to clients. Hence we have two practices that seem to exclude one another: testing or proving. In the presented paper we argue[1] that the two approaches may be synthesized to a completely different scheme of practice. We propose to replace the term *testing* by another word experimenting. Testing limits itself to the execution of program and the comparison of its effects with the predicted, supposedly correct results. For the majority of people testing is the practice of searching bugs in software. Experimenting has larger horizons, it covers not only searching of errors (aka counter-examples) but also gathering the positive evidence. Sometimes during experiments we begin to believe that objects obey certain rules and later we try to find more evidence confirming our beliefs. During experiments one is going to execute program with different data, to collect results and to present them in graphical mode, in tables, in data bases, etc. It is suggested that the experiments were done with some plan. Next, one should analyze the gathered experience and search some regularities. This process should lead to the formulation of lemmas and propositions. After this is done, there is the time of proving the hypotheses. Obviously, the process sketched above may need to be iterated for different reasons, e.g. when our program is modified. We consider the method given below

$$experiment \rightarrow observe \rightarrow formulate\ hypotheses \rightarrow prove.$$

[1] following to some extent the ideas of Georg Hegel who used to say that from a pair: thesis and antithesis we should make a synthesis

as a proper approach to the production of the high integrity software [22], [5], [6], [3].

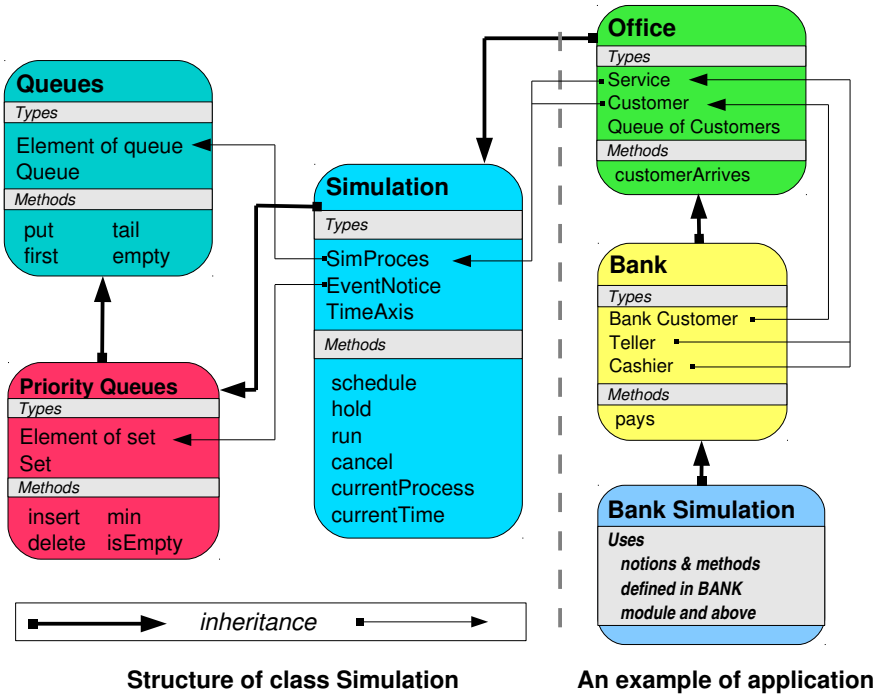
How to define the goals of software's verification? Assuming that a specification of a class is given in the form of a set of formulas, the verifier agent should prove that every object of the analyzed class will satisfy every formula of the set, whenever any method of the class has been completed.

We are proposing the methodology which consists of algorithmic logic AL and an environment SpecVer – a plugin into the Eclipse IDE. A SpecVer project can be developed from its initial phase of specifying algorithms and classes, through implementing them in an object programming language, to the phase of verification of implementation against its specification.

We present an almost complete case study which consists of a specification $ATPQ$ of priority queues, a class `PQS`, and the verification of the thesis that the class `PQS` correctly implements the specification $ATPQ$.

Class PQS may be used in several applications. For example, it is a part of a bigger program Simulation of a Bank Department. The structure of the simulation program is shown in Figure 1. The program contains 6 external classes, 13 innner classes and 20 methods. The relation between inner classes and the containing them external classes are shown on the diagram. The relation of inheritance is also shown on the diagram. The arrows lead from one class to its direct superclass. The thick arrows start at external classes. The thin arrows lead from an inner class K to inner class I inherited by the class K. We conceive the five external classes as implementations of data structures: Class `BankDepartment` extends the structure of `Office`. Class `Office` is based on the class `Simulation`, it uses also the data structure of `FIFOQueues`. The class `Simulation` relies on the class `PriorityQueues`. These relations are shown on the diagram by thick arrows. (As no one class may inherit two classes, we decided to make the class `FIFOQueues` the base class of the class `PriorityQueues`.) The diagram shows also the inheritance relation between inner classes. This allows us to introduce more subtle relations between classes. For example, any object of class `Customer` is queueable since the class `Customer` inherits from the class `Simprocess` which in turn inherits from the class `ElemFIFO`. An object of class `Customer` may be activated and made passive several times since the class `SimProcess` is a coroutine. The value of unique variable `ExperimPlan` is a set of `EventNotices`. During the execution of our simulation experiment the variable `ExperimPlan` has various sets of `EventNotices` as its value. An object of the class `EventNotice` is a pair: $\langle s.t \rangle$, where s is a `SimProcess` object and t is a time. Objects of class `EventNotice` are inserted and/or deleted from the set `ExperimPlan`. `EventNotices` are ordered by a relation *less*. The class `Simulation` is to guarantee that the active object of the experiment, in our case it will be either a customer or a teller object, will be active iff its activation time is the minimum of all times of `EventNotices` in `ExperimPlan` set. Encapsulating two inner classes `SimProcess` and `EventNotice` and the variable `ExperimPlan` of type `PQ` makes the process of coding of the class `Simulation` simpler.

The term high integrity software was introduced in [5]. For us the high integrity programming means the activity which involves specification of software modules, implementation the modules (i.e. classes and methods) and verification of the modules against their specification. We shall use the structure shown in Figure 1 to illustrate what has to be done.



Rysunek 20.1: Moduły programu symulującego bank

For each external class C three documents should be produced:

- A specification S of the class C. Specification is a set of formulas which express the properties of methods and invariants of objects of the class. Each specification should enjoy two properties:
  - Consistency
    There is no implementation of an inconsistent specification. An inconsistent specification has no sense.
  - Completeness
    An incomplete specification allows various models. Not all of them are desired. A complete specification brings enough information on properties of objects of class to distinguish between a desired and an improper implementation, and therefore can be used as a criterion of acceptance of a class. It is sufficient to produce the proofs or verification reports.
- The file containing the class C itself. Usually this file contains all the methods and inner classes of the class C.
- The verification report. This file should contain arguments that soothe our conscience and convince the user of our class. The arguments used may be more formal - having form of a mathematical proof or may recall a dialogue. Rarely a formal proof is needed. The verification report should be rather an evidence of analysis and it should serve to convince its reader

> that the conclusions of the report are sound. A verification report is of good quality if it is an intersubjective experience. Surely, a formal proof of a correctnes has this quality, but frequently it is not readable by a human being. A balance between two extremities is needed.

In earlier papers we have presented the work on specifications of classes [22]. Specifications are subjects of studies and analyses. In other words, one should visualize a process of development and amelioration of a specification.

This paper illustrates the process of verification of a class C against a specification S. In the most cases implementation of a class precedes the process of verification. Sometimes, the verification can be done simultaneously with the production of the needed class. In another paper we shall exemplify the (rare) case when a class, the Simulation class, is systematically elaborated together with the proof of its correctness. For this we need only the specification of the base class `PQS` and the target class `Simulation`.

We propose to make an experiment. The reader will look at the appendix and try to give arguments that the class PQS correctly implements a priority queues system. Those who forgot what a priority queue is, may find its axiomatic definition in Table 1 below. We ask the reader how much of time he/she needs to convince someone that the class `PQS` implements the abstract data type of priority queues.
We did the following:

1. Experiments - we executed methods of the class by hand and have drawn some pictures.
2. Observations - we analyzed the pictures and searched for some regularities.
3. Conclusions - we formulated several hypotheses (lemmas) and propositions.
4. Proving - we proved the lemmas.
5. Finally - putting together the facts, we proved the correctness theorem.

20.3.3 Priority Queues Specification

Before implementing a data structure one should write down its specification. The first part of the specification – the signature – enlists the sorts, the operations and the relations. The second part enlists the properties, also called the axioms, Table 1 contains the specification of the abstract data type of priority queues [24, p.154]. Remark that besides the formulas of first-order logic we use algorithmic formulas [24]. An example of algorithmic formula is the axiom (a2). Its structure is as follow:

$$\langle program\ P\rangle\langle formula\ \alpha\rangle.$$

The meaning of the whole formula is after execution of program P the formula α is valid". In our example the formula (a2) takes value true if and only if the program halts. Hence, including such formula among the axioms of our specification, means "the program P always terminates". The semantic meaning of the axiom (a2) is the set q is finite. The axiom (a8) is in fact an explicit, algorithmic definition of the relation *member*.

Algorithmic formulas allow to express the semantic properties of programs such as termination, correctness, etc. Almost 40 years ago we proved that the calculus is sound and complete [24]. It means that we have choice between showing that some semantic property of a program is valid or proving the formula that expresses the property. We gave several examples of specification of abstract data types as algorithmic theories. $ATPQ$ is one of such examples.

Table 1. Specification $ATPQ$ of priority queues.

Signature	Comments
Sorts	$Universe = E \cup PQ$
E	set of elements
PQ	set of priority queues
Operations	let $e \in E$ and $q \in PQ$
$insert : E \times PQ \longrightarrow PQ$	put e into q
$delete : E \times PQ \longrightarrow PQ$	delete e from q
$min : PQ \longrightarrow E$	find the minimum element
$empty : PQ \longrightarrow \{true, false\}$	is a priority queue q empty?
$member : E \times PQ \longrightarrow \{true, false\}$	does $e \in q$?
$\leq : E \times E \longrightarrow \{true, false\}$	the ordering relation

Axioms

(a1) *The set E ofelements is linearly ordered by the relation* $\leq$.
(a2) [**while** *not* $empty(q)$ **do** $q := delete(min(q), q)$ **done**] $true$
This axiom says for all q program halts, i.e. the priority queue q is finite
(a3) $[q1 := insert(e, q)]\{member(e, q1) \wedge (\forall_{e1 \neq e}\ member(e1, q1) \Leftrightarrow member(e1, q))\}$
(a4) $[q1 := delete(e, q)]\{\neg member(e, q1) \wedge (\forall_{e1 \neq e}\ member(e1, q1) \Leftrightarrow member(e1, q))\}$
(a5) $empty(q) \Rightarrow (\forall_{e \in E}\ \neg member(e, q))$
(a6) $\neg empty(q) \Rightarrow (\forall_{e \in E}\ member(e, q) \Rightarrow min(q) \leq e))$
The operation min finds the least element of the set q.
(a7) $[e := min(q)]true \Leftrightarrow \neg empty(q)$
Axiom (a7) says the result of expression $min(q)$ is defined iff $\neg empty(q)$
(a8) $member(e, q) \Leftrightarrow$ **begin**
s1 := q; result := false;
while not empty(s1) and not result **do**
if e =min(s1) **then** result:=true **fi**;
s1 := delete(min(s1),s1)
done
end result

$ATPQ$ is an acronym of Algorithmic Theory of Priority Queues. The theorems of the theory are the formulas provable by the calculus of algorithmic logic [24] from the axioms of $ATPQ$.

Uwaga 20.7. *In the literature, the frequent choice when speaking on the abstract data type of priority queues, is the operation deletemin, instead of the operation delete. Our choice was different and deliberate. With the present set of operations we are able to construct the specification which enjoys the property of being almost complete.*

It was shown in [24] that the algorithmic theory of priority queues has a metalogical property known as the representation meta-theorem: Every model

of $ATPQ$ is isomorphic to the standard model of priority queues. [2]
Let us add a few words on the $ATPQ$ specification. An important fact states that any implementation of the axioms of $ATPQ$ where a concrete set E was given, is isomorphic to the structure of finite subsets of the set E with the operations

$$insert(e, s) = \text{increase the set } s \text{ by adding element } e \text{ to it},$$

and

$$delete(e, s) = \text{supprime the element } e \text{ from the set } s.$$

The consequences of the theorem are of general methodological nature:

- the specification of $ATPQ$ is complete. If a new formula is added then either it is a logical consequence of the axioms or it leads to contradiction, or it expresses the properties of the elements only.
- the specification can be used as the criterion of correctness of a proposed implementation,
- the proofs of properties of programs that use this data structure may be based on axioms of priority queues listed in Table 1. No other properties of priority queues are ever needed.

20.3.4 Experiments

Our work on verification of the class PQS began by experimenting. The experiments consisted in executing (by hand) operations insert and delete, drawing pictures, and observing the changes in the constellation of objects of type node. Figure 2 shows a few snapshots of our experimentation. In fact, we did twice as much drawings. The reader may wish to continue our experiments on his own, e.g. by inserting the element $e6$ or deleting the element $e2$ after insertion of the element $e5$ was done. It is wortwhile to observe that the snapshot after execution of $delete(e2)$ will show the picture which is essentially the same as after insertion $e1, e2, e3, e4$ with following difference: Instead of $e2$ we find $e5$, and the pair of objects $\langle$ $e2$, its companion node object $\rangle$ is an isolated (not connected) part of the graf. During the experiments we neglected the ordering relation between elements.

Our first impression, when looking at the drawings of Figure 2, is a complete chaos. Slowly we commence to distinguish some parts and we begin to perceive some regularities. First, we remark that in each of 6 pictures there is exactly one object of type `PQ` which has two fields: *root*, *last*. Next, we remark that objects of type `Elem` and of type `Node` come in pairs, each pair is connected by *.el* and *.lab* arrows. Our next observation is that the arrows *.up* form lists of objects of type `Node` (no cycles). The meaning of arrows $.left$ and $.right$ is less evident that it may be suggested by their names. On the diagram some of them are solid and some of them are dotted. This comes as the result of analysis, the arrows $.left$ and $.right$ pay two roles. We see that the solid arrows $.left$ and $.right$ go against to the arrows *.up*. While the observation that there is no cycle in paths composed of *.up* arrows may lead to the statement that on each

[2] Algorithmic theories (e.g. [30], [24]) were studied since 70's of XX century.

diagram we have a tree of node objects, the present observation states that the tree is a binary one.
Our observations need to be properly formulated and proved. This will be done in the next section. Before that, one may execute further experiments, e.g. by executing the command $delete(e2)$.

20.3.5 Observations and Lemmas

We shall study the class `PQS`, see the Appendix. We can assume that a usage of `PQS` consists in a finite sequence of creation of `newElem`() objects and calls: `call q.insert(e), call q.delete(e')`.
Following the intuition gained from Figure 2 we shall introduce the notion of observable states. The initial state s_0 is the graph consisting of exactly one object o of type `PQ`, $s_0 = \{new\ \texttt{PQ}\}$ and no edges.

Definicja 20.3. *(of the observable states) The set $\mathcal{S}$ of observable states is the least set which contains the initial state s_0 and which is closed with respect to the operations insert and delete and creation of* `new Elem`() *objects.*

Each state consists of a set of objects and the edges connecting them. The examples of states are presented in Figure 2. The class `PQS` may be viewed as a definition of the relational structure PQS. The universe U of the structure consists of the objects of the inner classes of the class `PQS`. The attributes of objects of U define functions between objects. The set of objects of type `Node` will be denoted NODE. analogously for the set of objects of type `Elem` and of type `PQ`:

$$\begin{aligned} \text{NODE} &= \{n:\ n \text{ instanceof } \texttt{Node}\},\\ \text{ELEM} &= \{e:\ e \text{ instanceof } \texttt{Elem}\},\\ \text{PQ} &= \{q:\ q \text{ instanceof } \texttt{PQ}\}. \end{aligned}$$

Definicja 20.4. *The class* `PQS` *determines an algebraic structure*

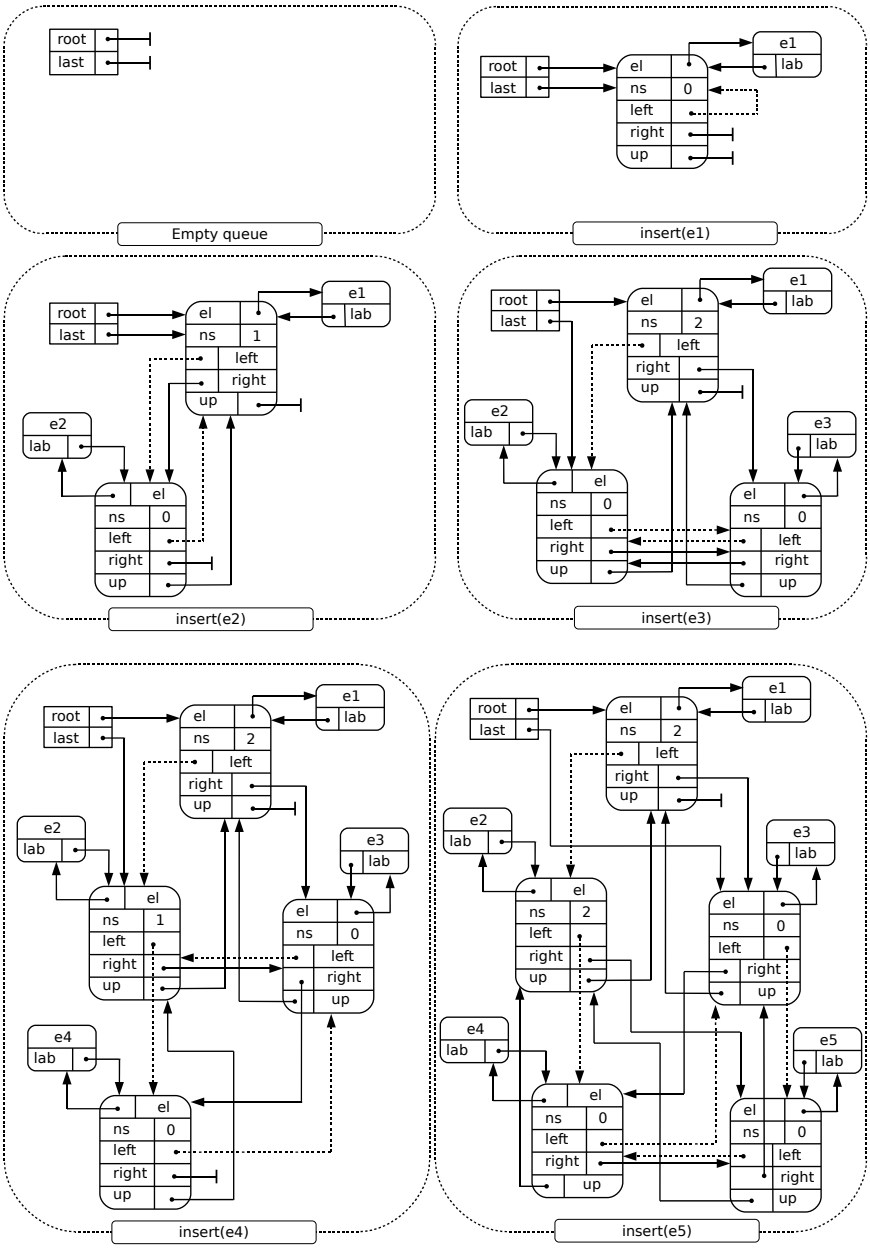
$$\text{PQS} = \langle U, .up, .left, .right, .el, .lab.\rangle,$$

where U is a subset of the union NODE$\cup$ELEM$\cup$PQ *which satisfies the condition*

$$(\forall_{n\in\text{NODE}})\ (\forall_{e\in\text{ELEM}})\ n.el = e \Leftrightarrow e.lab = n.$$

The functions of the structure PQS *are defined as follows*

$$\begin{aligned} .up &\stackrel{df}{=} \{\langle n, n'\rangle :\ n, n' \in \text{NODE},\ n' = n.up\},\\ .left &\stackrel{df}{=} \{\langle n, n'\rangle :\ n, n' \in \text{NODE},\ n' = n.left\},\\ .right &\stackrel{df}{=} \{\langle n, n'\rangle :\ n, n' \in \text{NODE},\ n' = n.right\},\\ .el &\stackrel{df}{=} \{\langle n, e\rangle :\ n \in \text{NODE}, e \in \text{ELEM},\ e = n.el\},\\ .lab &\stackrel{df}{=} \{\langle e, n\rangle :\ e \in \text{ELEM}, n \in \text{NODE},\ n = e.lab\}. \end{aligned}$$



Rysunek 20.2: Wstawianie elementów e1 - e5

Our first observation is that if we abstract from (i.e. we forget about) the arrows *.left* and *.right* then for each state s its graph shows a tree.

Definicja 20.5. *Let s be an observable state, consider the objects of type* `Node` *in this state. Let T_s be the set of these object of type* `Node` *that access the object .root object by a path composed from .up arrows only.*
$T_s = \{o \in \text{NODE} \cap s$: *there is a path composed from .up arrows only, leading from object o to object .root*$\}$

Lemat 20.8. *In any observable state s the pair $\langle T_s, .up\rangle$ is a tree.*

Definicja 20.6. *(of a son) We say that a node n is a son of a node f in the tree T_s if and only if $n.up = f$. A node n is said to be a leaf of the tree T_s if and only if it has no sons.*

Our next observation is

Lemat 20.9. *For every $o \in T_s$, $o.ns$ = number of sons of o.*

Definicja 20.7. *We say that an arrow .left from the node n is solid iff $n.ns > 0$. We say an arrow .right from the node n is solid iff $n.ns = 2$. Otherwise the arrows are said weak, or dotted.*

Next, we observe that solid arrows lead against *.up* arrows.

Lemat 20.10. *For every state s, the tree T_s with solid arrows only, forms a binary tree.*

Proof: For every two nodes n and f of the tree T_s the following properties hold:

- if a solid .left arrow leads from node f to node n then $n.up = f$
  $(f.left = n \wedge f.ns > 0\ \ ) \Rightarrow n.up = f$,
- if a solid .right arrow leads from node f to node n then $n.up = f$
  $(f.right = n \wedge f.ns = 2\ \ \wedge) \Rightarrow n.up = f$,
- $n.up = f \Leftrightarrow (f.left = n \vee f.right = n)$.

Hence T_s is a binary tree. Our next observation can be stated as follows:

Lemat 20.11. *There exists at most one node n in T_s such that $n.ns = 1$. If it is the case then $last = n$.*

Now, we observe that the leaves of tree T_s are on two levels only.

Lemat 20.12. *For every state s, there exists a natural number $k(T_s)$ such that every leaf of the tree T_s is on the level $k(T_s)$ or $k(T_s) - 1$.*

The number $k(T_s)$ is equal the length of the path composed from *.up* arrows leading from the object *last.left* to the *root* object. It is equal 0 if *root = none*.

Now we try to guess the rôle of non-solid arrows *.left* and *.right*. The following property holds:

Lemat 20.13. *The object referenced by the variable last in the tree T_s is the leftmost node on the level $k(T_s) - 1$ which has less than two sons.*

Let us return to the Figure 1 and observe the following facts:

Uwaga 20.14. *A) If a node n has two sons then its left brother has also two sons.*
B) If a node n has one son then it is its left son.
C) If a node n has one son then its brother from the left has two sons and its brother from the right is a leaf.

Lemat 20.15. *The value of the variable last is a head of a list of leaves linked together via .right (weak) arrows.*

Lemat 20.16. *The value of the variable last is a head of a cyclic list of leaves linked by (weak) .left arrow.*

We see that all leaves on the level $k(T_s)$ are grouped to the left.

Lemat 20.17. *Tree T_s is a perfect binary tree i.e. all the levels are completely filled with an eventual exception on the deepest level, in this case all the leaves are grouped to the left.*

The following four lemmas have similar form $(\alpha \Rightarrow I\beta)$, where I is either instruction $insert(e)$ or instruction $delete(e)$, α is a precondition and β is a postcondition of the instruction I.

Lemat 20.18. *Let $I : insert(e)$ and*
$\alpha_1 : \{last = o \wedge o.left = n1 \wedge o.right = k \wedge o.ns = 0 \wedge e.lab = n \wedge n.el = e\}$,
$\beta_1 : \{last = o \wedge o.ns = 1 \wedge o.left = n \wedge o.right = k \wedge n.up = o \wedge n.left = n1\}$.
The instruction I is correct with respect to the precondition α_1 and postcondition β_1 in the structure PQS

$$\mathrm{PQS} \models (\alpha_1 \Rightarrow I\beta_1).$$

Dowód. How to read this lemma? It states that in any state s, if the precondition α_1 is satisfied by s, then the execution of instruction $insert(e)$ will succesfully lead to certain state s' and the postcondition β_1 will be satisfied by s'. What is the meaning of the precondition α_1 of the instruction *insert*? We can draw it, see Fig. 3.

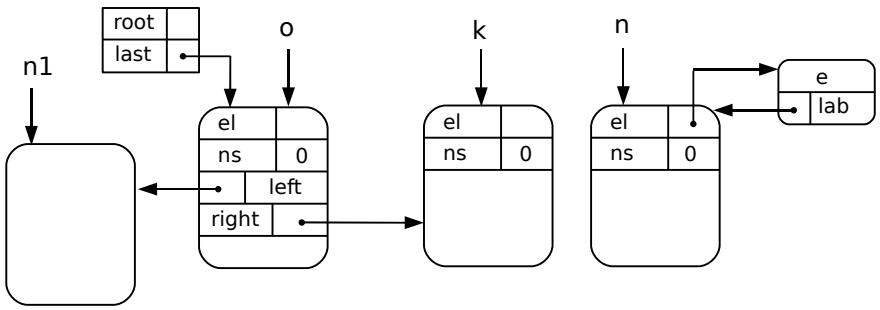


Fig. 3 Warunek wstępny α_1.

We check that the configuration of objects drawn on Fig. 3 satisfies the precondition α_1, The equality $last = o$ is satisfied since both variables point to the same object. The variable $last.left$ points to the object pointed by $n1$. $last.right$ point to the object pointed by k. The objects e and n are linked together, $e.lab = n$ and $n.el = e$.
Next, we can follow step by step the execution of the command $q.insert(e)$ with the text of method *insert* in hand. We start observing that the precondition α implies $last.ns = 0$ hence the instructions executed by *insert* are:
x:=e.lab; last.ns := 1; z := last.left; last.left := x; x.up := last;
x.left := z; z.right := x;last:=z;
Now we can draw modifications to the picture following the instructions.

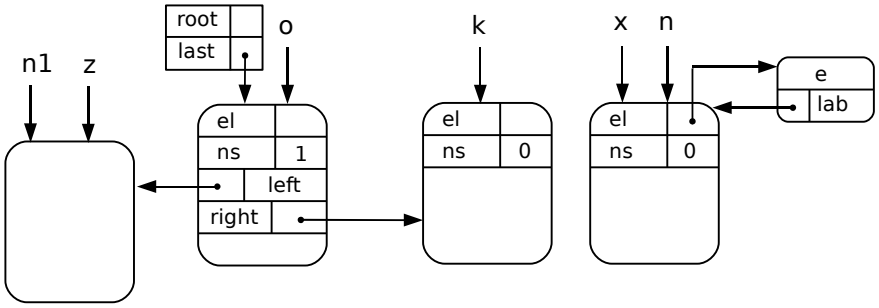


Fig. 4 Migawka po wykonaniu 3 instrukcji z treści instrukcji insert

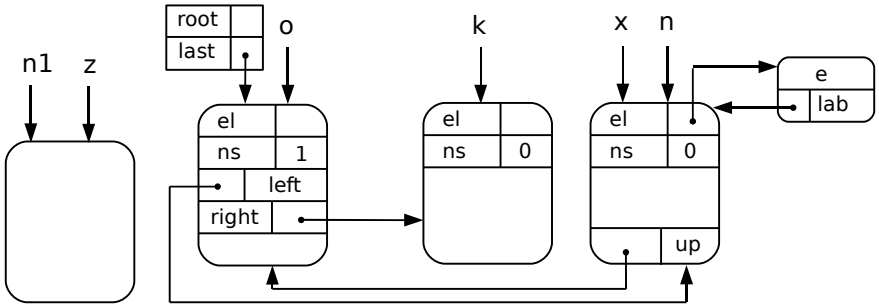


Fig. 5 Migawka po wykonaniu 5 instrukcji z treści instrukcji insert

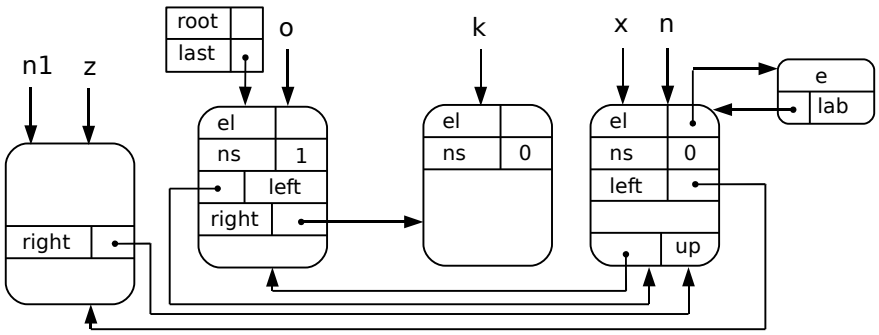


Fig. 6 Migawka po zakończeniu instrukcji insert .

The state shown on Fig. 6 can be described by the following formula $\gamma : \{last = o \wedge o.ns = 1 \wedge o.left = n = n1.right = e.lab \wedge o.right = k \wedge n.up = o \wedge n.left = n1 \wedge n.el = e\}$. It is easy to observe that the formula γ implies the formula β_1. Note that if the initial state s satisfies the precondition α_1 then at the end of the execution of the instruction $insert(e)$ in PQS we obtain the state s' satisfying the postcondition β_1.

Lemat 20.19. *Let $I : insert(e)$ and*
α_2:{last=o∧o.left=n1∧o.right=n2∧o.ns=1∧e.lab=n},
β_2: {last=n2∧o.ns=2∧o.left=n1∧o.right=n∧n.up=o∧n.left=n1 ∧n1.right=n}.
The instruction $I : insert(e)$ is correct with respect to the precondition α_2 and the postcondition β_2.

$$\mathrm{PQS} \models (\alpha_2 \Rightarrow I\beta_2).$$

Dowód. One can easily verify that if $last.ns = 1$ then the actions executed by the method *insert* are those presented in Table 2 below.

Table 2. Proof of lemma 3.

Precondition: α_2:(last=o∧o.left=n1∧o.right=n2∧o.ns=1 ∧e.lab = n)	
Instruction	*Effect*
x := e.lab	x=n, *since precondition says* e.lab=n
last.ns:=2	o.ns=2, *since* last=o
z := last.right	z=n2, *because* last.right=n2
last.right:=x	o.right=n, *because* last=o and x=n
x.right := z	n.right=n2, *because* x=n
x.up:= last	n.up=o, *because* last=o and x=n
z.left:=x	n2.left=n, *because* z=n2
last.left.right:=x	n1.right=n, *because* last.left=n1
x.left:=last.left	n.left =n1, *because* last.left=n1 and x=n
last :=z	last=n2, *because* z=n2
Postcondition: β':(o.ns=2∧o.right=n∧n.right=n2∧n.up=o∧ n2.left=n∧n1.right=n∧ n.left=n1∧last=n2∧o.left=n1∧e.lab=n)	

The postcondition β' collects the facts enlisted in the column Effect extended by the formulas e.lab=n and o.left=n1 for they remain satisfied after execution of *insert*. In this way we obtained a postcondition which is even stronger than the formula β_2.

The proofs of lemmas 20.18 and 20.19 exemplify two different ways of argumenting that a semantical property is valid. The first one, informal, consists in drawing the pictures. It may be related to drawing Venn's diagrams in the algebra of sets. Like diagrams of Venn it does not replace the proving but it is helpful. The second one is nearer to the goal of mechanization of proving.
In the following two lemmas we analyze properties of algorithm *delete*.

Lemat 20.20. *Let the formulas α_3, β_3, and the instruction D be defined as follows:*
$\alpha_3 : \{last = o \wedge o.left = n1 \wedge o.right = n2 \wedge o.ns = 0 \wedge e.lab = n \wedge n1.left = n3\}$,

$\beta_3 : \{last = n1.up \wedge last.right = o \wedge last.ns = 1 \wedge o.left = last \wedge o.right = n2 \wedge$
$n1.ns = 0 \wedge n1.up = n1.left = n1.right = none \wedge n3.right = none \wedge n1.el = e\}$,
$D : delete(e)$.
The instruction $delete(e)$ *is correct with respect to conditions* α_3 *and* β_3, *i.e.*

$$\mathrm{PQS} \models (\alpha_3 \Rightarrow D\beta_3).$$

Now we consider another case of applying the instruction *delete*.

Lemat 20.21. *Let the formulas* α_4, β_4 *be defined as follows:*
$\alpha_4 : \{last = o \wedge o.left = n1 \wedge o.right = n2 \wedge o.ns = 1 \wedge e.lab = n \wedge n1.left = n3\}$,
$\beta_4 : \{last = o \wedge o.left = n3 \wedge o.ns = 0 \wedge o.right = n2 \wedge n1.el = e \wedge n1.up =$
$n1.left = n1.right = none \wedge n3.right = none\}$.
The instruction $delete(e)$ *is correct with respect to conditions* α_4 *and* β_4, *i.e.*

$$\mathrm{PQS} \models (\alpha_4 \Rightarrow D\beta_4).$$

The instructions call correctUp() and call correctDown(), that end the execution of procedures insert and delete, serve to guarantee that for each path in the tree T of nodes the elements associated to the nodes of the path form a decreasing sequence with the minimum in the root.

Lemat 20.22. *(on procedure correctUp)*
Procedure instruction call correctUp(r) is correct w.r.t. the precondition γ_1 *and the postcondition* γ_2 *given below*
$\gamma_1 : r$ **in** $elem \wedge r.less(r.lab.up.el) \wedge last.left.el = r$
(The first condition r **in** $elem$ *is checked by compiler.) The second condition says the newly added element is less or equal than the element associated with the father of* r. *The third condition says: the companion node* n *of the element* r *is pointed by the pointer* $last.left$.
γ_2 :*for every node* n *on the path beginning at the element* r, *the following condition holds* $n.up.less(n) \vee n.up = none$.

Lemat 20.23. *(on procedure correctDown)*
Procedure correctDown is correct w.r.t. the precondition γ_3 *and the postcondition* γ_4 *given below*
$\gamma_3 : r$ **in** $elem \wedge \neg r.less(r.lab.up.el$
γ_4 :*for every node* n *on the path beginning at the element* r, *the following condition holds* $n.up.less(n) \vee n.up = none$.

Lemat 20.24. *For every two nodes* x, y *in the tree* T_s, $x.up = y \Rightarrow y.less(x)$.

Dowód. This property is invariant with respect to the operations insert and delete. At the very beginning the tree T is empty and the property holds. Assume that the property holds for a certain tree T. Consider another tree T' which is the result of operation insert or delete. After the insertion of an element the procedure correctUp is called and the tree is going to be repaired to keep the property. The same remark may be repeated in the case when the tree T' is the result of the operation delete on tree T. □

This sequence of observations leads to the following:

Lemat 20.25. *In each observable state* s *the tree* T_s *is a heap.*

Before proving the corectness of the implementation we should extend the class `PQ` adding two methods *empty* and *member*.

Boolean empty() { return root=none}
Boolean member(Elem e, PQ q) { the body of this method is given on the righthand side of the equivalence a8 of Table 1}.

Now we are ready to verify the implementation `PQS` of priority queues against the specification $ATPQ$ given in Table 1.
Let us start with the structure consisting of elements and heaps $\{T_s : s \in \mathcal{S}\}$ and operations insert and delete, min, and the relations empty and member. Consider the quotient structure $\mathcal{PQS}$ in which we identify the heaps that have the same sets of elements. We claim that it is a priority queues structure, a standard model of the theory of $ATPQ$. It suffices to verify that the axioms of Table 1 are formulas valid in the structure $\mathcal{PQS}$.
a2) The program **while** not q.empty() **do** q.delete(q.min()) **done** always terminates since each operation delete removes one element from a heap.
a3) This follows from the lemmas 20.18 and 20.19.
a4) This follows from the lemmas 20.20 and 20.21.
The verification of the remaining axioms of $ATPQ$ is left to the reader. We can conclude:

Twierdzenie 20.26. *The structure $\mathcal{PQS}$ of elements and* `PQ` *objects implemented by the class* `PQS` *is a priority queue.*

Remarks on cost:
The pessimistic cost of an operation *insert* or *delete* is $O(logn)$, where n is the number of elements in the priority queue. This property is very important in the application of priority queue as plan of experiment in the class Simulation that inherits (extends) the class PriorityQueue. Imagine, in an simulation experiment of a pandemia of influenza where the objects of SimProcess class count in hundreds of thousands and the number of EventNotice objects goes in milions, any implementation with the cost worse than $O(logn)$ would be impractical.

20.3.6 Final remarks

We have demonstrated the work on verification of a given class K against a specification S. Aswering the question *is the class K a correct implementation of the specification S* is a task completely different than proving correctness of an algorithm with respect to a given pre- and post-conditions. This paper shows that the formal counterpart of the task is asking whether a given class implements a set of axioms. In this context it is natural to conceive the class K as an algorithmic definition of some algebraic structure $\mathbb{A}$ and to study the question is the structure $\mathbb{A}$ a model of the specification S. We are stressing that high integrity programming requires many skills and a lot of invention. The analysis of this case study shows the wide repertoire of questions that may appear during the work on a software project. The incomplete list contains the following kinds of subgoals:

- specification of algorithms,
- specification of classes (this work is strongly related to the goal of specification of a data structure),

- construction of a method (i.e. procedure or function),
- construction of a class,
- verification of a conjecture given class K correctly implements some specification S.

In a future paper we shall demonstrate that, in a favorable circumstances, one can construct a class together with a proof of its correctness.

From previous sections we conclude that EOP (experiment, observe, prove) method can be successfully applied to software analysis. We do not claim it is a trivial task but it is at least realizable. As a matter of fact most programmers would agree that formal methods are generally better than sophisticated testing and simultaneously most of them are not using such methods at all. It seems that essential to the problem is lack of proper tools and experience i.e., tools which can integrate specification, implementation and verification tasks into single, consistent process. Experience can be gained only through everyday practice. Our goal is to develop integrated programming environment supporting every phase of software construction using formal methods. EOP idea presented in this paper is a part of bigger scheme called temporarily SpecVer programming. We assume that whole process of software production needs preparation of:

- specification documents: formal texts along with some math analysis (is it astonishing that some specifications are incorrect, or mor precisely they may be inconsistent or incomplete?),
- implementation code,
- verification reports: again formal texts with some proofs about implementation's quality.

As you can remark, EOP method could be used both for specification and verification tasks as soon as we can perform observations analogous to these made about Fig. 2. This again direct us towards programming tools. We need some object debugger showing program memory from object perspective, some formal support tools helping in logical theorems construction and perhaps checking if formulas are properly written and much, much more ... Some work has already been started, but a lot of it still should be done. For now we can present initial implementation of Eclipse plugin [31] supporting creation of specification documents.

Appendix - the class PQS

The full text of the class PQS as written by W.M. Bartol and D. Szczepańska. It is a part of bigger program of simulation of bank department [27] [28].

Część III

Programowanie z agentami lub programowanie rozproszone

W tej części przedstawimy współprogramy i procesy oraz obiekty współprogramów i obiekty procesów.
Wyjątki - programowanie z sygnałami wyjątków.?? Może to przenieść wyżej?

Rozdział 21

Współprogramy $\mathcal{L}_9$

21.1 Współprogramy

W tym rozdziale omówimy moduły współprogramów i współpracę z obiektami współprogramów.
Współprogram (*ang.* coroutine) jest modułem programu podobnym do klasy. Różnice można sprowadzić do krótkiego zdania: słowo class zastąp słowem coroutine i zezwól programiście stosować polecenie attach(c), Tu c jest zmienną wskazująca na obiekt współprogramu. Ważne jest by pamiętać, że instrukcja attach(c) może pojawiać się nie tylko w treści współprogramu, ale i w modułach funkcji i procedur zadeklarowanych we współprogramie. Wynika z tego następująca zasada: instrukcje występujące w treści współprogramu dzielą się na dwie części – te wykonywane aż do polecenia return włącznie z nim, to tzw. konstruktor, pozostałe instrukcje do wykonywanie których można powrócić gdy inny obiekt współprogram wyda polecenie attach(c) to są instrukcje wątku współprogramu, niektórzy mówią instrukcje włókna współprogramu.

Składnia

Współprogram jest to moduł o następującej strukturze

```
unit M: coroutine(args);
  𝔻
begin
  𝕀1 (* instrukcje konstruktora *)
  return;
  𝕀2 (* włókno lub wątek współprogramu *)
end M
```

gdzie M jest nazwą współprogramu,

args – listą parametrów formalnych,

$\mathbb{D}$ – listą deklaracji lokalnych,

$\mathbb{I}_1$, $\mathbb{I}_1$ – są ciągami instrukcji.

Instrukcja *wznawiania współprogramu* ma postać następującą

attach(x) .

Zmienne typu współprogram deklarujemy w ten sam sposób co zmienne typu klasa.
Np. **var** x,y: MyCoroutine, u,v: MyClass;
Jeśli chcesz to możesz tworzyć tablice współprogramów, zob. **??**
Wyrażeniami obiektowymi typu współprogram są :

- odpowiednio zadeklarowana zmienna,
- wyrażenie **new**(), generujące obiekt współprogramu,
- nazewnik funkcyjny zwracający obiekt współprogramu.

Semantyka

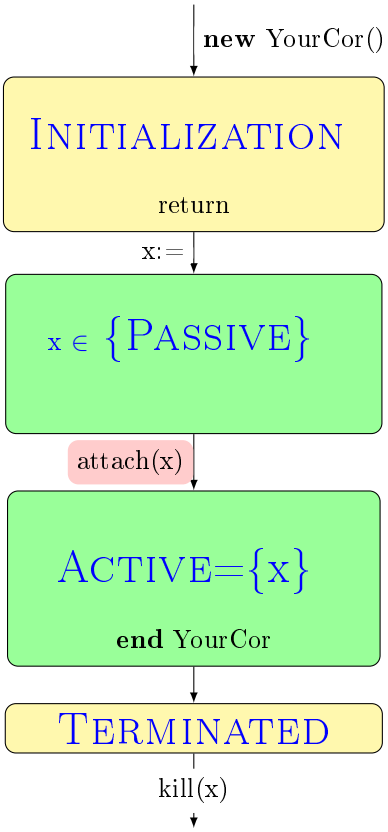
Semantyka obiektów współprogramów ma dwie strony: *bierną* – gdy obiekt współprogramu jest przedmiotem działań oraz *czynną* gdy obiekt współprogramu wykonuje swoje kolejne instrukcje (gdy sterowanie zostało przekazane do takiego obiektu). Zechciej porównać – sterowanie nie może zostać przekazane do obiektu klasy. Z drugiej strony instrukcja procedury wytwarza *anonimowy* rekord aktywacji procedury i *przekazuje* sterowanie do tego rekordu. Obiekt współprogramu może posiadać nazwę, tak jak obiekt klasy, ponadto inny współprogram może mu przekazać sterowanie. Do tego służy instrukcja attach(c). Program główny jest współprogramem. Program główny może wytworzyć obiekty współprogramów i może przekazać sterowanie do jednego z nich. Zobacz podrozdział Producent i konsument **??** poniżej.

Wartością wyrażenia obiektowego, np. **new** M(1,2), jest nowy obiekt współprogramu. Tworzy się go w ten sam sposób jak obiekt klasy. Każda operacja wykonywana na obiekcie klasy może też być wykonana na obiekcie współprogramu.

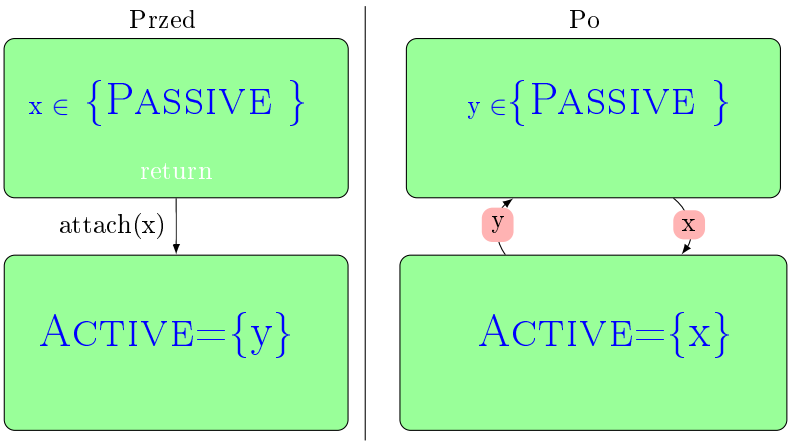
Ponadto PASYWNY obiekt c współprogramu może przejść do stanu AKTYWNY i wznowić wykonywanie swoich instrukcji.
Niech y będzie aktywnym obiektem, wykonanie instrukcji attach(c) polega na zapamiętaniu gdzie należy wznowić działanie gdy (inny) obiekt współprogramu wykona polecenie attach(y) (kiedyś, w przyszłości) i na wznowieniu wykonywania instrukcji współprogramu c, w miejscu w którym zawiesił on wykonywanie instrukcji. Za pierwszy razem polecenie attach(c) wznowi instrukcję następująca po poleceniu return. Każde kolejne polecenie attach(c) wznowi wykonywanie instrukcji od instrukcji następującej po ostatnio wykonanym poleceniu attach(x) w treści obiektu c. Obrazowo, wznawiamy działanie w miejscu w którym przerwaliśmy.

Poniżej przedstawiamy diagram stanów obiektu współprogramu Z rysunku 21.1 widać, że po wyczerpaniu wszystkich instrukcji do wykonania w obiekcie współprogramu, przechodzi on w stan TERMINATED. Polecenie attach(c) jest w takiej sytuacji błędem i zostanie to zgłoszone w trakcie wykonywania programu wraz z diagnostyka błędu.

Do wyjaśnienia: gdzie zostaną wznowione działania gdy obiekt współprogramu zakończył działanie?



Rysunek 21.1: Diagram stanów obiektu współprogramu



Rysunek 21.2: Zmiana stanów systemu współprogramów

Na rysunku 21.2 widać, stany systemu współprogramów przed i po wykonaniu instrukcji attach(x). Zbiory stanów PASYWNY i AKTYWNY zamieniły się obiektami. Obiekt aktywny y przeszedł w stan PASYWNY a obiekt x jest teraz AKTYWNY. Instrukcja ta jest wykonywana przez obiekt aktywny. Zwracamy uwagę na to, że instrukcja attach(x) powodując uaktywnienie obiektu x równocześnie przerywa wykonywanie ciągu instrukcji w aktywnym dotąd obiekcie y. To czego na rysunku nie widać, to fakt, że obiekt x wznawia obliczenia w tym miejscu w którym zostały one przerwane. Zobacz Przykład ...

Porównaj z diagramem stanów obiektu klasy. Łatwo zauważyć, że obiekt współprogramu może znaleźć się w nowym stanie AKTYWNY.

Pojęcie współprogramu (*ang.* coroutine) pojawiło się w latach 60 XX wieku. Wielu do dziś przyjmuje jako definicję zdanie sformułowane przez D. Knutha w [18], *subroutines are special case of coroutines.*

Trzeba jednak pamiętać o kontekście w jakim zdanie to zostało umieszczone. Dla Donalda Knutha środowiskiem w którym należy pisać programy jest assembler (wtedy MIX, dziś MMIX). W assemblerze pojęcie podprogramu ma inny sens niż pojęcie procedury w językach pochodnych od Algolu60. Podprogram jest fragmentem programu do którego możemy wykonywać skoki ze śladem. Podprogram ma jedno wejście tj. początek i w zasadzie jedno wyjście. Po ponownym wejściu do podprogramu nie wiemy nic o wartościach zmiennych lokalnych, jakie zostały obliczone poprzednio. Podprogram nie ma stanu jaki by mógł przetrwać od jednego do ponownego uruchomienia tego podprogramu. Korutyna (coroutine) pozwala wznowić obliczenia w miejscu z którego ją opuszczono. W języku programowania obiektowego współprogram jest specjalnym rodzajem klasy. Umożliwia to tworzenie wielu obiektów danego współprogramu C tak jak się tworzy obiekty klas. Po utworzeniu przez polecenie **new** obiekt taki pozostaje pasywny i można z nim postępować jak z obiektem klasy.
Możemy taki obiekt pasywny uaktywnić.

Współprogramy (*ang.* coroutines) - Pojęcie współprogramu ma dwie odmienne definicje(!).
Obie definicje zgodnie stwierdzają, że współprogram cechuje się posiadaniem ciągu instrukcji do wykonania i ponadto możliwością zawieszania wykonywania jednego współprogramu A i przenoszenia wykonywania do innego współprogramu B. W szczególności można wznowić pracę zawieszonego współprogramu A, a wykonywanie będzie podjęte w miejscu, w którym zostało zawieszone. Tym co różni obie definicje jest zdolność współpracy z rekurencyjnymi procedurami. (Nb. W językach programowania funkcyjnego koncepcja współprogramu istnieje pod postacią *kontynuacji* - pojęcia wprowadzonego niemal równocześnie z współprogramami.)

Obiekt współprogramu jest quasi-wątkiem. Tak jak wątek, obiekt współprogramu ma ciąg instrukcji do wykonania. W odróżnieniu od wątków obiekty współprogramów nie działają równolegle. Jest niezmiennikiem systemu współprogramów to, że w każdej chwili, dokładnie jeden obiekt współprogramu wykonuje swoje instrukcje:

$$card\{coroutine : \ coroutine \ is \ Active \ \} = 1.$$

W literaturze znaleźć można termin *włókno* (*ang.* fiber) dla odróżnienia od *wątku* (*ang.* thread).

Współprogramy jako specjalny rodzaj klas	**Współprogramy jako bogatszy rodzaj podprogramów**
W językach programowania: Simula67 , Loglan 82, BETA można tworzyć moduły **coroutine** tj. współprogramów. Składnia współprogramu różni się od składni klasy tym, że zamiast słowa **class** piszemy **coroutine**, i co ważniejsze, wewnątrz takiego modułu wolno używać instrukcji atomowych *attach* oraz *detach*. Instrukcje takie mogą się też pojawiać wewnątrz metod zadeklarowanych w współprogramie. Stwarza to nowe i interesujące możliwości współpracy współprogramów i procedur(funkcji) rekurencyjnych.	Śubroutines are special cases of coroutines."$_{D.Knuth}$. W assemblerze od dawna występuje pojęcie podprogramu - nie należy go mylić z pojęciem procedury. Podprogram istnieje w kodzie programu i ma co najwyżej jedną instancję. Nie jest możliwe rekurencyjne wykonywanie podprogramów.
Przykład: Łączenie drzew BST **var** T : *arrayof Traverser*; **unit** *Traverser* : **coroutine**(n : *node*); **var** *kolejny* : *integer*; **unit** *traverse* : **procedure**(m : *node*); **begin** **if** $m \neq$ *none* **then** **call** *traverse*(*m.left*); *kolejny* := *m.val*; **detach**; (∗ instrukcja **detach** wznawia współprogram, który ostatnio uaktywnil ten (this) obiekt Traverser wykonujac **attach**(.) ∗) **call** *traverse*(*m.right*); **fi** **end** *traverse*; **begin** **return**; **call** *traverse*(n) **end** *Traverser*; (*Stworz drzewa BST, w liczbie np. k drzew.* *Dla kazdego drzewa d* *stworz* $T[i]$:= **new** *Traverser*(d) (*Kolejne uaktywnienie tego wspolprogramu* **attach**($T[i]$) *spowodujewykrycie kolejnego co do wielkosci elementu z drzewa d*)	Przykład **var** q := **new** *kolejka* **coroutine** *produkuj* **loop** **while** *q nie jest pelna* *stworz troche nowych przedmiotow* *wstaw przedmioty do q* **yield to** *konsumuj* **coroutine** *konsumuj* **loop** **while** *q jest niepusta* *pobierz troche przedmiotow z q* *uyj te przedmioty* **yield to** *produkuj* W
Uwagi. 1Mamy tu do czynienia z dynamicznym systemem współprogramów. Wyrażenia generujące postaci "'new"' Traverser(...) umożliwiają stworzenie wielu obiektów typu współprogram Traverser. 2. Obiekt współprogramu Traverser wywołuje metodę traverse(). Łańcuch dynamiczny tego obiektu wydłuża się o rekord aktywacji metody traverse. Łancuch taki może być tak długi jak gałąź odwiedzanego drzewa BST. 3. Instrukcje "'attach"'() oraz "'detach"' mogą wystąpić nie tylko w ciągu instrukcji współprogramu, lecz także w metodach współprogramu. 4. Przełączenie wykonywania odbywa się pomiędzy łańcuchami dynamicznymi współprogramów.	**Uwagi**. 1. Ten system współprogramów jest statyczny: zawiera dwa współprogramy. Deklaracja **coroutine** automatycznie tworzy obiekty typu produkuj i konsumuj. 2. W tym systemie są dwa punkty wejścia. Otrzymany graf składa się z dwu współprogramów i dwu przełączeń **yield**.

latach 60 XX wieku współprogram był fragmentem kodu napisanego w assemblerze. o następujących właśnosciach: * dokładnie "jeden" współprogram wykonuje swoje instrukcje, tzn. jest aktywny, * współprogram aktywny może przejść w stan pasywny wskazując przy tym na inny wątek, który ma być uaktywniony, * współprogram x uaktywniony w efekcie wykonania instrukcji $attach(x)$ (w dotychczas aktywnym współprogramie y) kontynuuje wykonywanie instrukcji od odpowiedniego punktu wejścia, dokładniej: pierwsze uruchomienie instrukcji wątku współprogramu spowoduje wykonanie pierwszej instrukcji wątku, każda następna instrukcja $attach(x)$ wznawiająca wykonywanie wątku współprogramu x rozpoczyna wykonywanie instrukcji od punktu wejścia wyznaczonego przez ostatnio wykonaną w nim instrukcję $attach(...)$.

== Zasada działania współprogramów ==

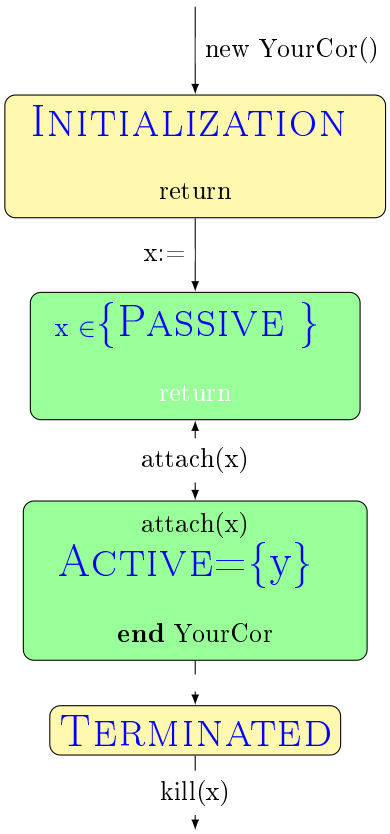
System współprogramów można nazwać systemem "quasi-współbieżnym". Nazwa ta jest uzasadniona dwojako: liczne przykłady programów współbieżnych np. producent-konsument, czytelnicy-pisarze itd. zapisane przy pomocy współprogramów okazuja się wystarczająco adekwatne do zastosowań. Inny argument wspierający użycie tej nazwy to fakt, że od bardzo dawna stosuje się współprogramy do symulacji systemów, np. w Simuli67, Loglanie'82 i in. Odpowiednia klasa Simulation dostarcza klasę wewnętrzną simproces - obiekty klas pochodnych od klasy simproces symulują rzeczywiste procesy np. pacjentów w systemie symulacji epidemii choroby, pojazdy w systemie symulacji ruchu w mieście itp.

Wielu autorów uważa, iż "współprogramy to podprogramy wykonywane w taki sposób, że sterowanie może zostać przekazywane pomiędzy nimi wielokrotnie, przy czym wywołanie danego współprogramu powoduje wykonywanie instrukcji od miejsca ostatniego przerwania wykonania (ostatniego punktu wyjścia), a nie od początku". Nie jest to całkiem ścisłe. Podprogramy (funkcja, metoda) tworzą rekordy aktywacji. Po opuszczeniu takiego rekordu jest on automatycznie usuwany i nie ma możliwości wznowienia go. Współprogramy wymagają więc wątków i są realizowane jako obiekty odpowiednich klas, a nie jako podprogramy czy procedury. Cytowany wyżej pogląd mocno zawęża koncepcję współprogramów. Co więcej, nie można zapominać, że instrukcjami wątku współprogramu mogą być instrukcje wywołania jego prywatnych metod(procedur). Metody te mogą zawierać instrukcje $attach(...)$ przekazujące sterowanie z jednego do innego współprogramu. Dokładniej, instrukcja attach przekazując [[sterowanie]] z jednego do drugiego współprogramu przenosi je z łańcucha dynamicznego jednego współprogramu do łańcucha dynamicznego innego współprogramu.

Łańcuch dynamiczny współprogramu zawiera obiekt i wątek współprogramu i ponadto, jeśli wykonano instrukcję procedury, to do łańcucha dynamicznego dołączony jest rekord aktywacji procedury. Zakończenie wykonywania instrukcji procedury(metody) powoduje skrócenie łańcucha dynamicznego. Instrukcja $attach(x)$ wykonana w rekordzie aktywacji procedury powoduje przejście do punktu wejścia w łańcuchu dynamicznym współprogramu x. Punktem wejścia (powrotu) dla dotychczas aktywnego współprogramu jest instrukcja w rekordzie aktywacji procedury występująca bezpośrednio za instrukcją $attach(x)$. Widać stąd, że liczba punktów wejścia (powrotu) danego współprogramu może być zmienna w czasie i może nie być niczym ograniczona!

Podsumowując, współprogramy to więcej niż obiekty zwyczajnych klas, a mniej niż obiekty aktywne wątków (ang. threads).

== Schemat zmian stanów obiektu współprogramu ==



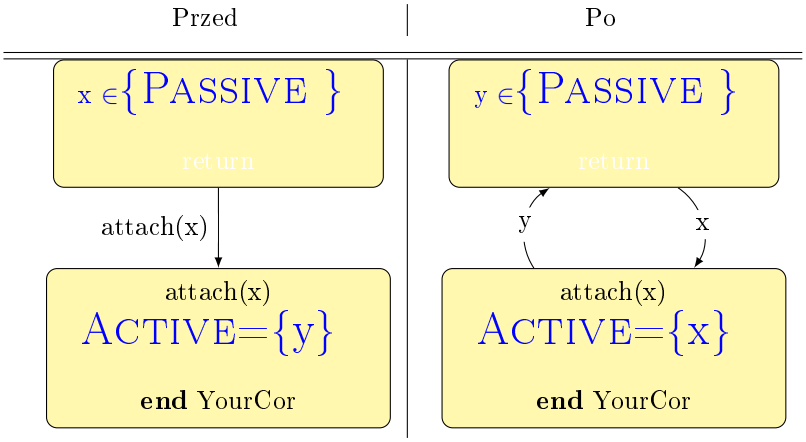
Rysunek 21.3: Diagram stanów obiektu współprogramu

Poniżej przedstawiamy diagram stanów obiektu współprogramu W poniższym zestawieniu widać stany systemu współprogramów przed i po wykonaniu instrukcji attach(x). Instrukcja ta jest wykonywana przez obiekt aktywny. Zwracamy uwagę na to, że instrukcja attach(x) powodując uaktywnienie obiektu x równocześnie przerywa wykonywanie ciągu instrukcji w aktywnym dotąd obiekcie y. To czego na rysunku nie widać, to fakt, że obiekt x wznawia obliczenia w tym miejscu w którym zostały one przerwane. Zobacz Przykład ...

== Współprogramy w języku Loglan '82 ==

* instrukcją przenoszenia sterowania z aktywnego współprogramu do drugiego współprogramu x jest $attach(x)$, * moduł współprogramu jest specyficzną klasą (stosuje się słowo 'coroutine' zamiast 'class'), * instrukcja $attach(x)$ może występować nie tylko w wątku współprogramu, lecz także w prywatnych metodach współprogramu(!), * "punktami wejścia" do współprogramu są: pierwsza instrukcja wątku współprogramu oraz każda instrukcja następna po instrukcji $attach(x)$, * można też używać bezparametrowej instrukcji $detach$, odpowiada instrukcji 'attach(ten współprogram, który ostatnio mnie wezwał)'.

== Instrukcje przenoszenia sterowania między współprogramami w różnych



Rysunek 21.4: Zmiana stanów systemu współprogramów

[[Język programowania|językach programowania]] ==

Instrukcje przenoszące sterowanie z jednego do drugiego współprogramu to * attach(x) oraz detach – w językach [[Simula 67]] i [[Loglan 82]], * yield(x) – w nowszych językach, np. w [[Python]]ie, * cede - w [[Perl]]u, w bibliotece Coro * instrukcja skoku - w [[assembler]]ze oraz w [[Fortran]]ie, gdzie podprogram nie jest procedurą, * trzeba odnotować też [[Implementacja (informatyka)|implementację]] współprogramów w [[Java|Javie]], jako wyspecjalizowanych wątków Javy.

Argument x wskazuje na współprogram pasywny w danej chwili.

== Przykładowe zastosowania współprogramów ==

* historycznie pierwsze współprogramy to skaner i [[Analizator składniowy]] [[kompilator]]a[odn|ref=nie|Conway|1963], * podobny schemat występuje w wielu sytuacjach np. jeden współprogram zbiera wyniki pomiarów i zapisuje je w bazie danych, a drugi współprogram opracowuje zebrane wyniki, ogólny schemat to producent-konsument, * jeżeli jakaś metoda (funkcja lub procedura) jest wykonywana wielokrotnie z tymi samymi parametrami aktualnymi, to warto utworzyć odpowiednie obiekty współprogramu (tablicę współprogramów) dla każdego zestawu parametrów aktualnych. Następnie każdą instrukcję wywołania procedury zastępujemy odpowiednią instrukcją *attach*(..). Zysk może okazać się znaczny, ponieważ wykonanie instrukcji *attach* jest znacznie prostsze od tworzenia rekordu aktywacji procedury. * Jeśli współprogramy są klasami wyposażonymi w instrukcję *attach*(...) to można tworzyć hierarchie współprogramów wykorzystując dziedziczenie. * główne zastosowanie współprogramów to narzędzia symulacji, takie jak klasy Simulation w Simuli 67 i w Loglanie'82.

Zadanie 21.1. *Czy dwa obiekty współprogramów mogą być równocześnie aktywne?*

Przypisy <references/>

== Bibliografia == cytuj pismo |odn=tak|nazwisko=Conway| imię=M.E.| tytuł= Design of a separable transition-diagram compiler| czasopismo= Communications of the ACM| wydanie=6|miesiąc= July|rok= 1963 cytuj książkę|odn=tak|imię=Ole-Johann |nazwisko=Dahl|imię2= Bjarne|nazwisko2=Myhrhaug|imię3= Kristen| nazwisko3= Nygaard| tytuł=Common Base Language (Simula67) |miejsce=Oslo

|rok=1970 |wydawca=NCC cytuj pismo |odn=tak|nazwisko = Dahl|imię = O.-J. |nazwisko2 = Wang|imię2 = A.| tytuł = Coroutine sequencing in a block structured environment |czasopismo = BIT| strony = 425-449|rok = 1971 cytuj książkę|odn=tak|imię=W. M. |nazwisko=Bartol| nazwisko2=i in.| tytuł=Report on the Loglan'82 Programming Language |miejsce=Warszawa Łódź |rok=1984 |wydawca=PWN |url=http://lem12.uksw.edu.pl/images/c/c2/Report82.pdf cytuj książkę|odn=tak|imię=Andrzej|nazwisko=Szałas| nazwisko2=Warpechowska|imię2=Jolanta| tytuł= Loglan'82 |miejsce=Warszawa |rok=1991 |wydawca=WNT |url=http://lem12.uksw.edu.pl/imag

21.2 Producent i konsument

Popatrzmy jak wygląda jeden z najczęściej omawianych przykładów. Program producent konsument ma następującą strukturę.

```
program PRODCONS;
  var prod:producer,cons:consumer,n:integer,mag:real,last:bool;

  unit producer: coroutine; ...
  end producer;

  unit consumer: coroutine(n:integer); ...
  end consumer;

begin (* MAIN program *)
  prod:=new producer;
  read(n);
  cons:=new consumer(n);
  attach(prod); (* R *)
  writeln;
end PRODCONS;
```

Obliczenie programu rozpoczyna się od utworzenia obiektu *prod* współprogramu PRODUCER. Następnie zostanie odczytana wartość n – będzie to rozmiar bufora. Z kolei program główny tworzy obiekt *cons* współprogramu CONSUMER. Instrukcja *attach*(*prod*) przenosi procesor do obiektu *prod*. Po powrocie obliczeń do wykonywania programu głównego wykonamy instrukcję writeln, Miejsce powrotu dla programu głównego jest zaznaczone jako (* R *).
Dalsza analiza programu wymaga przeczytania treści współprogramu PRODUCER,

```
unit PRODUCER: coroutine;
begin
  return; (* 1 *)
  do
    I_P : { read(mag); (* markujemy algorytm produkcji – obliczenia mag *)
            (* mag is nonlocal variable, common store*)
            if mag=0
            then (* end of data *)
                 last:=true;
                 exit
            fi;
          }
    attach(cons); (* A *)
  od;
  attach(cons) (* B *)
end PRODUCER;
```

a także treści współprogramu CONSUMER.

```
unit CONSUMER: coroutine(n:integer);
  var Buf:arrayof real; var i,j:integer;
begin
  newarray Buf dim(1:n);
  return; (* 2 *)
  do
    I_C : { for i:=1 to n
            do
              Buf(i):=mag;
              attach(prod); (* C *)
              if last then exit exit fi;
            od;
            for i:=1 to n
            do (* print Buf *)
              write(' ',Buf(i):10:2)
            od;
            writeln;
          }
  od;
  (* print the rest of Buf *)
  for j:=1 to i do write(' ',Buf(j):10:2) od;
  writeln;
  attach(main); (* D *)
end CONSUMER;
```

Podczas wykonywania tego programu powstaną trzy obiekty współprogramów. Na poniższym rysunku 21.5 są one przedstawione w kolorze zielonym. Strzałki koloru czarnego wskazują, że wartością zmiennej *prod* jest obiekt typu PRODUCER do którego prowadzi ta strzałka, a wartością zmiennej *cons* jest obiekt typu CONSUMER wskazany przez strzałkę zaczynająca się od cons. Natomiast czerwone strzałki pokazują efekt wykonania operacji `attach( )`. Wykonywanie programu rozpoczyna się w rekordzie aktywacji programu głownego. Pierwsza instrukcja spowoduje utworzenie obiektu współprogramu PRODUCER i przypisanie tego obiektu jako wartości zmiennej *prod*. Druga instrukcja powoduje wczytanie (z klawiatury) liczby i przypisanie jej do zmiennej n. Trzecia instrukcja spowoduje utworzenie obiektu współprogramu CONSUMER i przypisanie go do zmiennej *cons*. W efekcie inicjalizacji (konstrukcji) obiektu *cons* powstaje tablica Buf o elementach numerowanych od 1 do n.

Czwarta instrukcja programu głównego przenosi procesor do obiektu *prod*, w miejsce w którym instrukcja return zakończyła inicjalizację tego obiektu. Proces produkcji przedmiotu i wstawienia do magazynu ilustrujemy przy pomocy instrukcji read(mag). W praktycznych zastosowaniach może to być np. ciąg instrukcji dokonujących pomiarów. Przyjęliśmy umowę, że wczytanie wartości mag=0 kończy pracę programu. Jeśli wczytana wartość mag $\neq$0 to wykonamy instrukcję attach(cons). Za pierwszym razem wznowimy działanie obiektu *cons* za instrukcja return. Proces konsumpcjijest tu zamarkowany przez wstawienie wczytanej wartości do bufora Buf (i wydrukowanie bufora co jakiś czas). poczym konsument *cons* przekazuje sterowanie do producenta *prod*, wykonując polecenie attach(prod). Producent powtara etap produkcji i ponownie przekazuje sterowanie konsumentowi. Zakładamy, że w pewnym momencie zostanie wczytana wartość mag=0. Wtedy producent ustawia wartosc zmiennej boolowskiej *last* true i przekazuje sterownie do konsumenta. Konsument w każdym cyklu sprawdzał czy wartość last jest true. (Z definicji początkowa wartosc last to false). Gdy jednak last jest true konsument drukuje częściowo wypełnioną tablicę Buf i przekazuje sterowanie do programu głównego wykonując polecenie attach(main). Przypominamy w tym miejscu, że program główny jest obiektem współprogramu.
Od tej pory obiekty *prod* i *cons* kooperują ze sobą, przekazując sobie sterowanie (lub jeśli wolisz przenosząc processor) za pomocą instrukcji `attach`. **Uwaga**. Zakończenie wykonywania programu nastapi gdy obiekt *prod* ustawi wartość zmiennej *last* true i przekaże sterowanie obiektowi *cons*. Z kolei obiekt *cons* wydrukuje tele liczb, ile znajduje sie w buforze i oddaje sterowanie programowi głównemu, dla zakończenia obliczeń.[1] Analiza tego programu prowadzi do następujących wniosków:

- Podczas wykonywania programu istnieja cztery jednostki dynamiczne: MAIN – rekord aktywacji programu głównego, obiekty współprogramów *prod* $\in$ PRODUCER, *cons* $\in$ CONSUMER i tablica buf.

- Wprowadzenie czerwonych strzałek stworzyło wrażenie chaosu, a tak nie jest. Lepszy obraz współdziałania tych trzech obiektów współprogramów uzyskamy na rysunku 21.6 poniżej .

Pewnym przybliżeniem, sekwencyjnego przecież, programu PRODCONS może być wyrażenie regularne

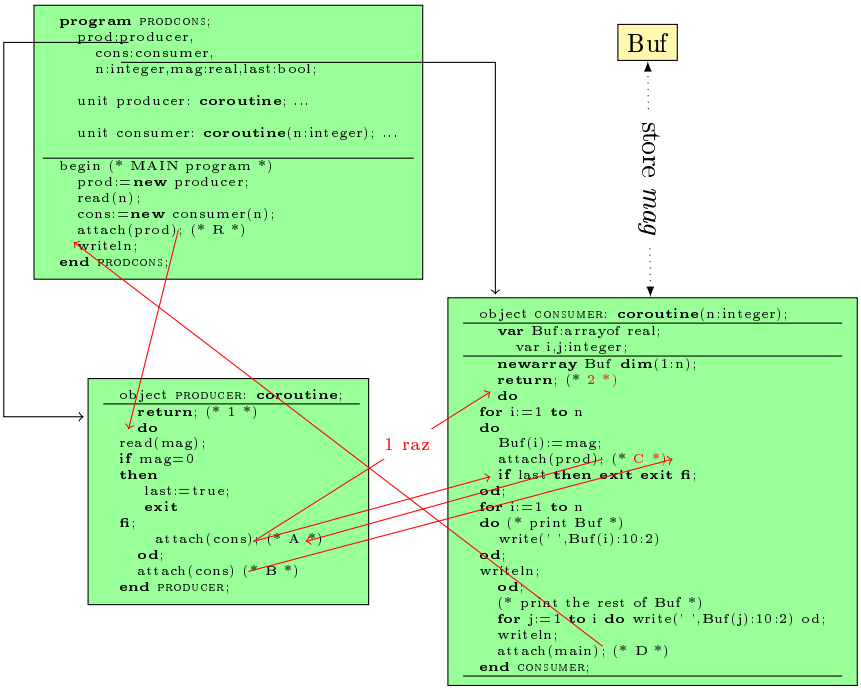
$$I_{main}; (I_P; I_C)^+; R_C; R_{main}$$

w którym I_P oznacza ciąg instrukcji oznaczony, a_p instrukcję attach(prod), ...

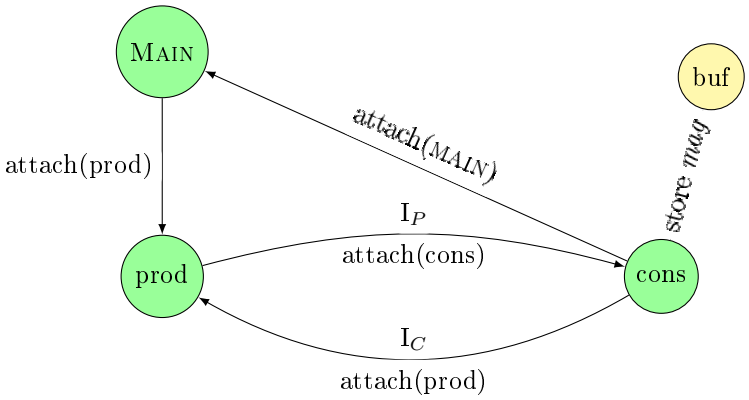
Wniosek 21.1. *Z powyższych obserwacji wynika, że program* PRODCONS *można zastąpić równoważnym programem iteracyjnym, wolnym od modułów współprogramów* PRODUCER *i* CONSUMER.

Jaki byłby koszt tego przekształcenia? Zapewne spory. Wydaje się, że taniej jest napisać program z wykorzystaniem współprogramów. Bierzemy tu również pod uwagę, że zastosowanie współprogramów zmniejsza ryzyko popełnia błędu w organizacji współpracy.

[1] Zwracamy uwagę na fakt, że ani obiekt *prod*, ani obiekt *cons* nie wyczerpuja list swoich instrukcji do końca. Tzn. nie dochodzi do zmiany stanu obiektu *prod* na TERMINATED.



Rysunek 21.5: Trzy obiekty współprogramów: prod, cons i MAIN



Rysunek 21.6: Diagram współpracy agentów prod, cons i MAIN

21.3 Instrukcja detach

W poprzednim przykładzie użyliśmy instrukcji attach do przełączania obliczeń pomiędzy obiektami współprogramów. W tej sekcji poznamy instrukcję detach.

Przykład.
Mamy do dyspozycji dwie drukarki, jedna o szerokości m1 i druga o szerokości m2. Program wczytuje ciąg bloków liczb. Każdy blok rozpoczyna się od nagłówka będącego liczbą m1 lub m2. Każdy blok kończy się liczbą 0. Koniec pliku wejściowego jest zgłaszany jako blok o nagłówku 0. Nasz program może wyglądać tak:

```
program READER_PRINTERS;
  const m1=10,m2=20;
  var reader:reading,printer_1,printer_2:writing;
  var n:integer,new_sequence:boolean,mag:real;
    unit writing:coroutine(n:integer); ...
    end writing;
    unit reading:coroutine ; ...
    end reading;
begin
  reader:=new reading;
  printer_1:=new writing(m1); printer_2:=new writing(m2);
  do
    read(n);
    case n
    when 0: exit
    when m1: attach(printer_1)
    when m2: attach(printer_2)
    otherwise write("wrong data"); exit
    esac
  od
end;
```

Współprogram READING ma następująca treść

```
unit READING: coroutine;
begin
  return;
  do
    read(mag);
    if mag=0 then new_sequence:=true; fi;
    detach;
    (* detach returns control to printer_1 or printer_2
     depending which one reactivated reader *)
  od
end READING;
```

W jaki sposób zapewnić, że ...?
Czy we współprogramach ...?
Współprogram writing wygląda tak:

```
unit WRITING: coroutine(n:integer);
  var Buf: arrayof real, i,j:integer;
begin
  array Buf dim (1:n); (* array generation *)
  return;
  do
    attach(reader); (* reactivates coroutine reader *)
    if new_sequence
    then
    (* a new sequence causes buffer Buf to be printed out *)
      for j:=1 to i do write(' ',Buf(j):10:2) od; writeln;
      i:=0; new_sequence:=false;
      attach(main)
    else
      i:=i+1; Buf(i):=mag;
      if i=n
      then
        for j:=1 to n do write(' ',Buf(j):10:2) od; writeln; i:=0;
      fi
    fi
  od
end WRITING;
```

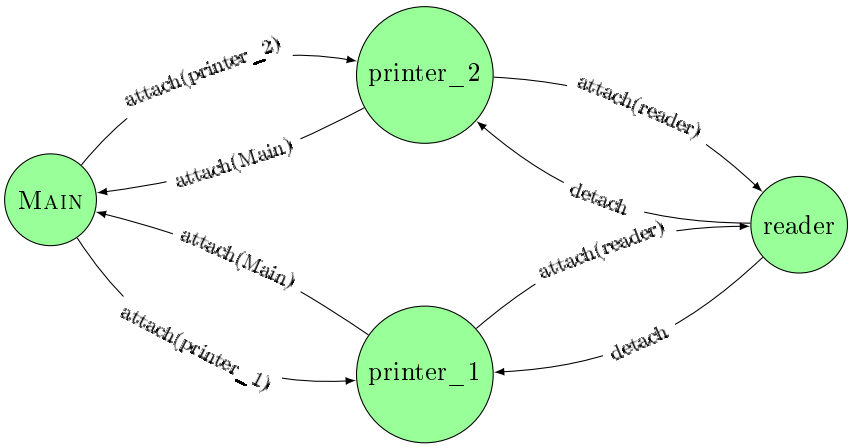
Dlaczego w współprogramie READING występuje polecenie detach? Co by się działo gdybyśmy użyli polecenia attach?
W tym przykłdzie zamiast instrukcji detach mozna uzyć
if szer=m1 then attach(printer_1) else attach(printer_2) fi
No i trzeba jeszcze zadbac by wartośc szer była odpowiednio aktualizowana. Napisz odpowiednie polecenie(a).
A co zrobić gdy obiektów uaktywniających współprogram reader jest dużo? gdy liczba ta nie jest znana przed wykonaniem programu?
Widac, że polecenie detach jest przydatne.

Poniższy rysunek 21.7 ilustruje związki pomiędzy czterema współprogramami: Main, printer_1, printer_2 i reader.

A może narysować podobny, inny diagram? Tak by było wyraźnie widać, że uaktywnieniu obiektu *reader* współprogramu READING z obiektu *printer_i* odpowiada powrót do aktywności w tym samym obiekcie *printer_i*. W ten sposób dochodzimy do zrozumienia uwagi Donalda Knutha sprzed 40 lat: *"podprogramy to szczególny przypadek współprogramów"*. Rzeczywiście, obiekt *reader* zachowuje się jak podprogram.
Niektórzy (Dahl, Kreczmar) mówia w takim przypadku o semi-coroutinach.

21.4 LICZNIK - dynamiczna tablica obiektów współprogramu.

W tym paragrafie zorganizujemy współpracę systemu obracających się kółek, który w efekcie zachowuje się jak wiele liczników:np. licznik zuzytej energii elektrycznej, licznik przejechanych kilometrów, etc. Program licznik pokazuje jak wykorzystać współprogramy w większym programie: moduł A może zaj-



Rysunek 21.7: Diagram współpracy agentów printer_1, printer_2, reader i Main

mować się obserwacją "impulsów", zdarzeń powodujących, że
moduł B wytwarza kolejną kombinację – tu przedmiotów 0,1,2, ... ,9 i przekazuje sterowanie do kolejnego
modułu C, który w jakiś sposób pożytkuje tę informację (np. o stanie licznika).

zmodyfikuj program PawelG, zastąp instrukcję drukowania instrukcją detach i ...

Częściej potrzebna nam będzie kolejna permutacja. Odpowiednio zmodyfikowany program PawelG zajmie się wytworzeniem wszystkich permutacji.

21.5 Scalanie drzew BST – łańcuch dynamiczny

Ten paragraf poświęcony jest pojęciu łańcucha dynamicznego.
Zaczniemy od przykładu. Przykład zaczerpnięty z pracy [?], por. [?]. **Zadanie**.
Dane są drzewa BST w liczbie k. Należy utworzyć ciąg niemalejący elementów zapisanych w węzłach tych drzew.
Zadanie to można rozwiązać na wiele sposobów. Wykorzystanie współprogramów okaże się bardzo naturalnym podejściem do tego problemu.
Dane to k elementowa tablica D drzew BST (a dokładniej, tablica węzłów-korzeni tych drzew). Z każdym drzewem $D[i]$ związemy jeden współprogram $T[i]$.

Analiza zadania.
Drzewo BST może być opisane np. tak

```
unit Tree: class;
  var root: node;
  unit node: class(val: integer);
    left, right: node;
  node;
  traverse: procedure(v:node);
  begin
    if v=none then return
    else call traverse(v.left); write(v.val); call traverse(v.right);
    fi
  end traverse;
  unit insert: procedure(e:integer); ... end insert;
end Tree;
```

Zaobserwujmy

Lemat 21.2. *Niech k będzie obiektem klasy* TREE. *Instrukcja call traverse(k) spowoduje wydrukowanie wszystkich liczb zapisanych w drzewie o korzeniu k.*

A teraz zmienimy nasz program. W procedurze traverse instrukcję write zastąpimy poleceniem detach. Rozpatrzmy następujący program *test*.

```
program test;
  unit node: class(val:integer);
    var left,right: node;
  end node;
  var A: CA, B: CB, kolejny, integer, n: node, czyWszystkie:boolean;
```

```
unit CA: coroutine(n: node);
    unit T: procedure(y: node);
    begin
      if y<>none then
        call T(y.left);
        kolejny := y.val; detach;
        call T(y.right)
      fi
    end T;
begin
    czyWszystkie:=false;
    return;
    call T(n);
    czyWszystkie:=true;
end CA;
```

```
unit CB: coroutine;
begin
    return;
    while not czyWszystkie do
      attach(A)
      write(kolejny);
    od
end CB
```

```
begin
  n:= new node;   (* tu instrukcje wypełniające drzewo n *)   A:= new CA(n);
  B:= new CB;
  attach(B)
end test
```

Lemat 21.3. *Wykonanie powyższego programu spowoduje wydrukowanie wszystkich liczb zapisanych w drzewie w porządku rosnącym.*

Dowód. Dowód wynika z poprzedniego lematu i z własności operacji detach oraz attach. Dowód przez indukcję ze względu na wysokość drzewa n. □

Jaki sens ma zastępowanie prostszego programu przez program bardziej skomplikowany?

Wyobraźmy sobie teraz, że mamy tyle współprogramów ile jest drzew. Program główny może "*poprosić*" dowolny z współprogramów by podał kolejną co do wielkośći wartośc przechowywana w drzewie . Oto szkic algorytmu scalania drzew BST.

---

1. *niech każdy współprogram poda najmniejszy element w drzewie D[i].*
2. *powtarzaj ...*
3. *wybierz najmniejszy element e, zapamiętaj nr drewa w j,*
4. *dopóki choćiaż jedno drzewo ma jeszcze element do pokazania*

Definicja tego pojęcia
Łańcuchem dynamicznym obiektu c pewnego współprogramu nazywamy ciąg jednostek dynamicznych taki, że (Baza) do łańcucha należy obiekt c, (krok indukcyjny) a) jeśli wykonywane jest polecenie tworzące nową jednostkę dynamiczną bloku, procedury, funkcji lub obiektu to wskaźnik systemowy DL nowej jednostki wskazuje na poprzednio aktywną jednostke dynamiczną. Jednostka ta jest w ten sposób dołączana do łańcucha wydłużając go,
b) jeśli w ostatnim elemencie łańcucha zakończono wykonywanie ostatniej instrukcji (bloku, procedury, funkcji lub inicjalizacji klasy) to jednostka taka jest odłączana od łańcucha dynamicznego skracając go.

21.6 attach zastępuje call

W tym miejscu (po raz drugi) nawiązujemy do zdania wypowiedzianego przez Donalda Knutha. W strukturze drzew binarnych poszukiwań implementujemy trzy operacje:

m) sprawdzania czy element e nalezy do drzewa,

i) wstawianie elementu e do drzewa,

d) usuwanie eleentu e z drzewa.

Rozważmy program, który wielokrotnie wykonuje każdą z tych operacji. Powiedzmy, że operacja sprawdzania jest wykonywana k razy, operacja wstawianie l razy, operacja usuwania jest wykonywana m razy. W związku z tym program utworzy $k + l + m$ razy rekord aktywacji odpowiedniej procedury. Taki rekord istnieje tylko przez chwilę, potem, po zakończeniu procedury jest automatycznie usuwany. Kolejne wywołanie (instrukcja **call**) powoduje powstanie niemal identycznego rekordu aktywacji i ... jego usunięcie i tak wiele razy. Czy można tego uniknąć?
Spróbujemy zastąpić instrukcje **call** member(e), **call** insert(e) i **call** delete(e) przez instrukcje attach aktywujące odpowiedni z trzech obiektów współprogramów. Deklarację klasy BST zawierającą funkcję member i procedury insert oraz delete zastąpimy deklaracją podobnej klasy BSTC w której zamiast procedur pojawią się współprogramy.

```
unit BSTC: class (type t; function less(x, y:t):boolean);
  var root: node, member: e, insert: i, delete: d;
  unit node: class (value: t);
    var l, r: node;
  end node;

  unit e: coroutine; ...

  unit help: e coroutine; ...

  unit i: help coroutine; ...

  unit d: help coroutine; ...

begin
  member:=new e; insert:=new i; delete:=new d;
  inner;
  kill(member); kill(insert); kill(delete)
end BSTC;
```

Zauważ, że współprogramy zadeklarowane w tej klasie tworzą hierarchię.

W programie głównym możemy zechcieć zadeklarować klasę MT i funkcję boolowska *mniejsze* porównującą obiekty klasy MT. Wtedy następująca instrukcja bloku dziedzicząca z klasy *BSTC* zastąpi instrukcję bloku dziedziczącą z klasy *BST*.

```
pref BSTC(MT, mniejsze) block
var y:MT;
...
begin

... (* call insert(y) *)
insert.x:=y;
attach(insert);

... (* elem:= contains(y) *)
member.x:=y;
attach(member);
if member.elem then ... fi;

... (* call delete(y) *)
delete.x:=y;
attach(delete);
...
end;
```

Obejrzyjmy deklaracje współprogramów. Współprogram e ma następującą treść

```
unit e: coroutine;
(*elem- output attribute*)
  var trick, elem: boolean, q, v: node, x:t;
begin
  return;
  do trick, elem:=false; (* loop for member *)
    q:=root;
    v:=none;
    while q=/=none
    do
      if less(x, q.value)
      then v:=q; q:=q.l
      else
        if less(q.value, x)
        then v:=q; q:=q.r
        else elem:=true; exit
        fi
      fi
    od;
    (* elem=true iff x belongs to S *)
    inner;
    detach;
  od
end e;
```

Pomocniczy współprogram *help* dziedziczy z *e*.

```
unit help: e coroutine;
begin
  inner; (* trick=true iff x does not belong to S *)
  if not trick then exit fi;
  if v=none
  then root:=q
  else
    if less(x, v.value)
    then v.l:=q
    else v.r:=q
    fi (* after insert or delete *)
  fi (* attach new node q to its father v *)
end help;
```

Współprogram *help* jest rozszerzany przez współprogramy *insert* i *delete*.

```
unit i: help coroutine;
begin
  trick:=true;
  if elem then exit fi;
  q:=new node(x) (* insert is a dummy if x belongs to S *)
end i;
```

Współprogram *d* jest trochę dłuższy.

```
unit d: help coroutine;
  private w, u, s;
  var w, u, s: node;
begin (* delete is a dummy if x does belong to S *)
  if not elem then exit fi;
  w:=q;
  if q.r=none
  then q:=q.l
  else
  if q.l=none
  then q:=q.r
  else u:=q.r;
  if u.l=none
  then u.l:=q.l; q:=u
  else
  do s:=u.l;
  if s.l=none then exit fi;
  u:=s
  od;
  s.l.:=w.l; u.l:=s.r;
  s.r:=w.r; q:=s
  fi
  fi
  fi;
  kill(w)
end d;
```

Załóżmy, że MT jest klasą i że funkcja *mniejsze* spełnia warunki wyliczone w definicji relacji porządkującej, tzn. jest zwrotna, przechodnia i antysymetryczna. Program (blok) P2 powstaje z programu P1 przez zastapinie każdej instrukcji call przez odpowiednią parę instrukcji wg następującej tabeli.

call insert(a);	≡	insert.x:=a; attach(insert);
bl:=member(b);	≡	member.x:=b; attach(member); bl:=member.elem;
call delete(c);	≡	delete.x:=c; attach(delete);

Twierdzenie 21.4. *Przy spełnieniu powyższych założeń programy P1 i P2 są równoważne.*

Dowód. Dowód przebiega przez indukcję ze względu na liczbę instrukcji call w programie P1. W dowodzie wykorzystamy następujące lematy.

□

21.7 Wieże Hanoi

Przyjrzyjmy się ponownie zadaniu przenoszenia krążków w buddyjskiej świątyni w Hanoi. W rozwiązaniu rekurencyjnym dostrzeżemy szansę na skrócenie czasu obliczeń. Podstawową operacją jest wywołanie procedury *move* czyli przenieś n krążków z wieży o numerze f na wieżę o numerze t. Wiemy, że takie operacje trzeba wykonać 2^n razy. Tymczasem liczba różnych zestawów argumentów wynosi $n \times 3 \times 3$ i jest znacznie mniejsza od 2^n. Każde wykonanie instrukcji call

move(a,b,c) wymaga utworzenia rekordu aktywacji dla procedury *move* i jego zamknięcia.
Proponujemy by

1. zastąpić procedurę *move*, współprogramem o nazwie *wz*. Zobacz tabelę poniżej.
2. utworzyć $9n$ (a nawet tylko $6n$) obiektów tego współprogramu z wszystkimi dopuszczalnymi wartościami argumentów i zapisać je w trójwymiarowej tablicy P,

   var P: arrayof arrayof arrayof wz;

3. zastąpić każdą instrukcję call move(a, b, c) przez instrukcję attach(P(a,b,c))

Ad 1. Porównajmy procedurę *move* i współprogram *wz*.

```
unit move:procedure(n,f,t:integer);
(* n rings from stick f to stick t *)
     var k:integer;
begin



    k:=6-(f+t);
    if n>1 then call move(n-1,f,k); fi;
    call modyf(f,t);
      (* move only one ring *)
    if n>1 then move(n-1,k,t); fi;
    return

end move;
```

```
unit wz:coroutine(n,f,t:integer);
(* n rings from stick f to stick t *)
     var k:integer;
begin
    return;
    do
      k:=6-(f+t);
      if n>1 then attach (p(n-1,f,k)); fi;
      call modyf(f,t);
      (* move only one ring *)
      if n>1 then attach (p(n-1,k,t)); fi;
       detach;
    od;
end wz;
```

Ad 3. Porównanie instrukcji call move(4, 1, 2) i instrukcji attach(P(4,1,2)) prowadzi do następującego wniosku.

każde wywołanie	var x: wz;
call move(a,b,c);	...
spowoduje	x:=new wz(a,b,c);
– utworzenie rekordu aktywacji move	...
– przesłanie argumentów a, b, c	każde polecenie
– wykonanie treści procedury	attach(x)
– usunięcie rekordu po zakończeniu	jest równoważne poleceniu
	call move(a,b,c).

Wniosek 21.5. *Przyjmujemy, że dla każdego* $1 \leq i \leq n$, *dla każdego* $1 \leq j \leq 3$ *i dla każdego* $1 \leq k \leq 3$ *zachodzi* $P[i,j,k] =$ ***new*** $wz(i,j,k)$.
Wtedy wykonanie instrukcji call move(i,j,k) *daje dokładnie ten sam efekt, co wykonanie instrukcji* attach(P[i,j,k]).
Natomiast instrukcja attach pozwala zaoszczędzić czas potrzebny na utworzenie rekordu aktywacji i na przekazanie parametrów.

Na żółto pomalowany jest rekord aktywacji procedury *move*. Rekord taki tworzony jest za każdym wywołaniem call move(n, f, t). Przenoszone sa do niego wartości parametrów n, f, t. Wykonują się instrukcje z treści procedury *move*. Potem ten rekord aktywacji jest usuwany.
Na zielono pomalowany jest obiekt współprogramu *wz*. Obiekt taki jest tworzony jeden raz dla każdej kombinacji argumentów n, f, t przez polecenie P[n,f,t]:=

```
SL=MAIN   |   DL=
n: integer =
f: integer =
t: integer =
k: integer
begin


  k:=6-(f+t);
  if n>1 then call move(n-1,f,k); fi;
  call modyf(f,t);
     (* move only one ring *)
  if n>1 then move(n-1,k,t); fi;
  return
end move;
```

```
SL=MAIN   |   DL=
n: integer =
f: integer =
t: integer =
k: integer
begin
  return;
   do
     k:=6-(f+t);
     if n>1 then call move(n-1,f,k); fi;
     call modyf(f,t);
       (* move only one ring *)
     if n>1 then move(n-1,k,t); fi;
     detach
   od
end wz;
```

new wz(n,f,t). Każde polecenie call move(n, f, t) może być zastąpione przez polecenie attach(P[n,f,t]) z tym samym efektem na zmiennych programu głównego. Nie trzeba tworzyć rekordu aktywacji – obiekt jest gotowy. Nie trzeba przesyłac parametrów – to zostało zrobione podczas tworzenia obiektu współprogramu. Nie trzeba usuwać obiektu – pozostaje on w gotowości do następnego wykorzystania.

A tutaj mamy cały program

```
program HanoiTowers;
(* towers of hanoi *)
(* there are three towers built of decreasing rings stringed onto sticks *)
(* at the initial state all rings are stringed onto stick no. 1. our job is *)
(* to move all rings from the stick 1 to the stick 3. the difficulty is *)
(* that we mustn't violate the following conditions *)
(*  1. we can move only one ring at one step *)
(*  2. each ring may be placed only onto a greater one *)
(* to manage with this difficult problem we have an auxilliary stick 2 *)
unit wz:coroutine(n,f,t:integer);
(* move n rings from stick f to stick t *)
var k:integer;
begin
return;
do
k:=6-(f+t);
if n>1 then attach (p(n-1,f,k)); fi;
call modyf(f,t); (* move only one ring *)
if n>1 then attach (p(n-1,k,t)); fi;
detach;
od;
end wz;

unit modyf:procedure(f,t:integer);
```

```
(* move the topmost ring from stick f to stick t *)
begin
top(t):=top(t)+1;
w(t,top(t)):=w(f,top(f));
w(f,top(f)):=0;
top(f):=top(f)-1;
call displ;
end modyf;

unit displ:procedure;
(* printing *)
var t,i,j,k,m,n:integer;
begin
t:=1;
for i:=2 to 3 do
if top(i)>top(t) then t:=i fi od;
t:=top(t);
for i:=t downto 1 do
m:=15;
for j:=1 to 3 do
for k:=1 to m do write(); od;
if w(j,i)=/=0 then for k:=1 to w(j,i) do write("*") od;
fi;
m:=15-w(j,i);
od;
writeln;
od;
for i:=1 to 15 do write(); od;
for i:=1 to 45 do write(-"); od;
writeln;
end displ;

var w:arrayof arrayof integer, (* how many rings are stringed *)
(* on each stick *)
top:arrayof integer, (* the topmost ring size on each stick *)
nb,i,j,k,timeb:integer,
p:arrayof arrayof arrayof wz; (* coroutine pointers *)

begin
array w dim(1:3);
array top dim(1:3);
writeln("program towers of hanoi");
writeln("version with coroutines");
do writeln("give the number of rings");
read(nb);
writeln(nb);
if nb>0 then exit else writeln(ńumber of rings must be greater than 0")
fi od;
timeb:=time;
top(1):=nb;
```

```
array w(1) dim(1:nb);
array w(2) dim(1:nb);
array w(3) dim(1:nb);
k:=nb;
for i:=1 to nb do w(1,i):=k;
k:=k-1;
od;
(* stick 1 is full *)
writeln("the algorithm acts as follows");
call displ;
    array P dim (1:nb);
    for i:=1 to nb
    do
      array P(i) dim(1:3);
      for j:=1 to 3
      do
        array P(i,j) dim(1:3);
        for k:=1 to 3
        do if j=/=k then P(i,j,k):=new wz(i,j,k) fi      od
        od
      od;
    (* ∀1≤i≤n∀1≤j≤3∀1≤k≤3P[i,j,k] = new wz(i,j,k) *)
    attach (P(nb,1,3));
    writeln(" execution time for",nb:4," rings =",time-timeb," sec");
end
```

21.8 Treegen

W tym paragrafie przedstawiamy hierarchię typów współprogramów. W dowodzie twierdzenia o poprawności programu korzystającego z tej hierarchii stosujemy indukcję ze względu na tę hierarchię, a własciwie jest to indukcja ze względu na długość wyrażenia.

Definiowanie skończonych zbiorów

Zadaniem programu treegen.log jest wydrukowanie wszystkich słów z języka opisanego przez wyrażenie regularne, niezawierające gwiazdek. Zwróć uwagę, na ogół zbiory skończone przechowywane sa w jakiejś strukturze danych. Ale nie musi tak być zawsze. Jeśli potrafisz zapisać zbiór odpowiednią formułą to możesz zyskać w wielu aspektach. Przykładem jest formuła $23 \leq i \leq 2306$. Dzięki niej możemy precyzyjnie i krótko opisać zbiór $A = \{i \in mathbbN : 23 \leq i \leq 2306\}$.

Struktura programu

Program główny dzieli się swoimi dwoma zasobami z obiektami współprogramów, jakie są tworzone w dalszym ciągu. Są to tablica znaków WORD i zmienna całkowito-liczbowa i.

```
var WORD: arrayof char, i: integer, ...
```

Zacznijmy od współprogramu *konsumenta*

```
(* otoczenie wspólne dla wszystkich współprogramów *)
  var WORD: arrayof char,
      i: integer;
unit konsumuj: coroutine;
begin
  return;
  do
    attach(o);
    (* print the WORD *)
    for J:=1 to I
    do
      write(WORD(J))
    od;
    writeln;
    if o.B then exit fi
  od
end konsumuj;
```

Program – konsument – instrukcja attach(o) przekazuje sterowanie do obiektu

Hierarchia współprogramów

Tworzymy kilka modułów współprogramów.

RYsunek – drzewo hierarchii REGEXP

```
unit REGEXP:coroutine;
  var B:BOOL; (* B ≡ all the words of the language were shown *)
begin
  return
  inner;
  B := true
end REGEXP;
```

```
unit ATOM: REGEXP coroutine(c:CHAR);
begin
  do
    i:=i+1; (* update the position *)
    WORD(i):=c;
    B:=TRUE;
    detach
  od
end ATOM;
```

```
unit UNiON: REGEXP coroutine(l,r:REGEXP);
  var m: INTEGER;
begin
  do
    m:=i;
    do
      attach(l);
      if l.B then exit fi;
      detach;
      i:=m
    od;
    l.B:=FALSE;
    do
      detach;
      i:=m;
      attach(r);
      if r.B then exit fi;
    od;
    r.B:=FALSE;
    B:=TRUE;
    detach;
  od;
end UNION;
```

```
unit CONCATENATION: REGEXP class(l,r:REGEXP);
  var N,m:INTEGER;
begin
  do
    m:=i;
    do
      attach(l);
      N:=i;
      do
        attach(r);
        if r.B then if l.B then exit exit else exit fi fi;
        detach; i:=N
      od;
      r.B:=FALSE;
      detach;
      i:=m
    od;
    r.B, l.B:=FALSE;
    B:=TRUE;
    detach
  od;
end CONCATENATION;
```

Twierdzenie 21.6. *Niech o będzie obiektem podklasy klasy Regexp, o in Regexp. Poniższy program Pr wydrukuje wszystkie słowa należące do języka regularnego |o| opisanego przez wyrażenie regularne o i zatrzyma się.*

```
Pr: I:=0;
do
attach(o);
if o.B then exit
else
(* print the WORD *)
for J:=1 to I
do
write(WORD(J))
od;
writeln;
fi
od
```

Dowód. Dowód jest rozproszony pomiędzy podklasy klasy Regexp. Ktoś mógłby powiedzieć, że mamy do czynienia z dowodem ze względu na hierarchię podklas klasy regexp. W rzeczywistości nasz dowód przebiega przez indukcję ze względu na długość wyrażenia regularnego ω, jakie zostało zakodowane w obiekcie *o*. Baza) Jeśli obiekt spełnia relację o is Regexp to wykonanie instrukcji attach(o) powoduje natychmiastowy powrót z wartościa o.B true. Takie wyrażenie opisuje pusty język. Teza twierdzenia jest w tym przypadku spełniona.
Udowodnimy nieco mocniejszy lemat:

Lemat 21.7. *Przypuśćmy, że wcześniejsze aktywacje obiektu o współprogramu Regexp wypełniły tablicę WORD na miejscach Let i0 be the value of the variable I. Suppose that the some words of the language L(o) were generated by the earlier activations of the coroutine o.*
An execution of command attach(o) has the following effect: the subsequent word of the language L(o) is concatenated to the content of the WORD(1), ... , WORD(I); i.e. the new word is placed beginning of the position WORD(I+1). The value of B attribute becomes true iff all the words of the language L(o) were shown.

Jeśli obiekt spełnia relację o is Atom spełniony jest następujący lemat
to uzyskujemy jedno słowo o długości jeden.
W obu tych przypadkach teza jest □

program Treegen; (* Generates the language defined by a regular expression*)
(* Program written by A. Kreczmar 1982
proof written by A. Salwicki 1990 *)

unit REGEXP:coroutine;
(* an object in the hierarchy of subtypes of type REGEXP represents a regular expression *)
(*
Theorem
For every object o in the hierarchy of classes that inherit from Regexp class the

program Pr (see below), when executed will print all the words of the regular language represented by the object o and then it will stop.

```
Pr: I:=0;
do
attach(o);
(* print the WORD *)
for J:=1 to I
do
write(WORD(J))
od;
writeln;
if W.B then exit fi
od
```

Lemma
Let i0 be the value of the variable I. Suppose that the some words of the language L(o) were generated by the earlier activations of the coroutine o.
An execution of command attach(o) has the following effect: the subsequent word of the language L(o) is concatenated to the content of the WORD(1), ... , WORD(I); i.e. the new word is placed beginning of the position WORD(I+1). The value of B attribute becomes true iff all the words of the language L(o) were shown.

PROOF of the lemma is distributed in the subclasses of the class regexp, i.e. the proof goes by induction with respect to the length of a regular expression *)

```
var B:BOOL; (* B º all the words of the language were shown *)
begin
return
inner;
B := true
end REGEXP;
```

```
unit ATOM: REGEXP class(C:CHAR);
(* an atomic regular expression consists of a letter
Proposition. An execution of attach statement applied to this object will place
the letter C on I+1-th place
in the table WORD and the value of B will be assigned to true. In this way the
whole regular language is displayed at once.
in this way we proved the base of the induction proof of the Lemma. *)
begin
do
I:=I+1; (* update the position *)
WORD(I):=C;
B:=TRUE;
detach
od
end ATOM;
```

```
    unit UNION: REGEXP class(L,R:REGEXP);
(* represents the expression (L È R) i.e. the union *)
(* Proposition. Assume that objects L and R enjoy the property expressed by
the Lemma
then any time this coroutine will be attached we obtain a subsequent word of
the union of the languages L and R.
```

Consider, a regular expresion of the length k. By our definition it is either a union object or a concatenation object.
Let o be a union object i.e. o is UNION. The structure of its commands assures the following

```
while not exhausted(L)
do
attach(L) – by induction hypothesis this command returns a word of L language
od
(* L.B = true *) – the exhaustion mark for L
while not exhausted(R)
do
attach(R) – by induction hypothesis this command returns a word of R language
od
(* R.B = true
B = true *)
```

It is evident that in this way by repeated execution of attach(o) one obtains a sequence of words composed from the all words of L language followed by the sequence of all words from the R language. *)

```
    var m: INTEGER;
begin
do (* repeat : store I; generate one word (first from L next from R; detach;
restore I until exhausted *)
m:=I;
(* I is the position of the lastly generated letter. *)
(* M+1 is the position where the current UNION object *)
(* will place the letters of the currently generated word. *)
do
attach(L); (* by the inductive assumption this statement causes that one word
will be generated of the language L and it will be concatenated to the content
of WORD(1) , ... , WORD(I) *)
if L.B then exit fi;
detach;
I:=m (* reestablish the position in the table WORD for the next word *)
od;
L.B:=FALSE; (* restart language L *)
do
detach;
I:=m; (* reestablish the position in the table WORD for the next word *)
attach(R); (* by the inductive assumption this statement causes that one word
will be generated of the language R and it will be concatenated to the content
of WORD(1) , ... , WORD(I) *)
```

```
if R.B then exit fi;
od;
R.B:=FALSE; (* restart language R *)
B:=TRUE;
detach;
od;
end UNION;

    unit CONCATENATION: REGEXP class(L,R:REGEXP);
(* represents the concatenation (L·R) of the languages represented by the regular expressions L and R *)
(* Suppose the object o is of the class CONCATENATION.

    Now the loop of commands of object o assures basically the following
while not exhausted
do
store (I);
attach(L); – a word from L
attach(R); – followed by a word from R
detach; – hence a word of (L R) is given
restore(I)
od

    with the necessary reactions to a case when one language (L or R) ends.
It is clear that if the object L and R enjoy the property mentioned in the Lemma
then the object o enjoys it too*)

    var N,m:INTEGER;
begin
do
m:=I; (*M stores the begin position of first language word position *)
do
attach(L);
N:=I; (* N stores the begin position of the second language word position *)
do
attach(R);
if R.B then if L.B then exit exit else exit fi fi;
detach; I:=N (* restart language R word generation position *)
od;
R.B:=FALSE; (* restart language R *)
detach; I:=m (* restart language L word generation position *)
od;
R.B,L.B:=FALSE; B:=TRUE; detach
od;
end CONCATENATION;



const N=50; (* DIMENSION FOR ARRAY WORD *)
```

```
    var A,B,C,D,E,W,V,L,O,G,II,NN:REGEXP,
I,J,N,M:INTEGER;
(* I = GLOBAL POSITION POINTER FOR ARRAY WORD *)
var WORD: arrayof CHAR; (* BUFFER FOR WORDS GENERATION *)

begin
writeln(ŁANGUAGE GENERATOR USING COROUTINES");
writeln(ŁANGUAGE IS REPRESENTED AS A TREE WITH OPERATIONS
IN NODES");
writeln(ÓUR OPERATIONS ARE SET THEORETICAL JOIN AND CON-
CATENATION OF");
writeln(ŁANGUAGES");writeln;
A:=new ATOM('A'); B:=new ATOM('B'); C:=new ATOM('C');
D:=new ATOM('D'); E:=new ATOM('E');
L:=new ATOM('L'); G:=new ATOM('G');
II:=new ATOM('I'); NN:=new ATOM('N');
O:=new ATOM('O');
W:=new UNION(A,L);
W:=new CONCATENATION(W,new UNION(D,O));
V:=new CONCATENATION(II,C);
V:=new UNION(V,new CONCATENATION(L,new CONCATENATION(A,NN)));
V:=new CONCATENATION(G,V);
V:=new UNION(A,V);
W:=new CONCATENATION(W,V);
writeln("WE HAVE LANGUAGE DEFINED BY THE FOLLOWING EXPRES-
SION");
writeln;
writeln("(AÈL)·(DÈO)·(AÈG·(I·CÈL·A·N))");
writeln; writeln;
array WORD dim(1:N);
do
attach(W);
write();
for J:=1 to I
do
write(WORD(J))
od;
writeln;
if W.B then exit fi
od
end

    (*
Theorem
For every object o in the hierarchy of classes that inherit from Regexp class the
program Pr (see below), when executed will print all the words of the regular
language represented by the object o and then it will stop.
Pr: I:=0;
do
attach(o);
```

for J:=1 to I
do
write(WORD(J))
od;
writeln;
if W.B then exit fi
od

Lemma
Let i0 be the value of the variable I. Suppose that the some words of the language L(o) were generated by the earlier activations of the coroutine o.
An execution of command attach(o) has the following effect: the subsequent word of the language L(o) is concatenated to the content of the WORD(1), ... , WORD(I); i.e. the new word is placed beginning of the position WORD(I+1). The value of B attribute becomes true iff all the words of the language L(o) were shown.

Proof.
Induction with respect to the length of the expression represented by the object o.

Base. Suppose the actual type of o is ATOM. Then the thesis of the lemma is satisfied.

Induction step. Suppose the lemma holds for every regular expression shorter than an integer k. Consider, a regular expresion of the length k. By our definition it is either a union object or a concatenation object.
case A. Let o be a union object i.e. o is UNION. The structure of its commands assures the following
while not exhausted(L)
do
attach(L) – by induction hypothesis this command returns a word of L language
od
L.B := true – set the exhaustion mark for L
while not
do
attach(R) – by induction hypothesis this command returns a word of R language
od
L.R := true
B := true

It is evident that in this way by repeated execution of attach(o) one obtains a sequence of words composed from the all words of L language followed by the sequence of all words from the R language.

case B Suppose the object o is of the class CONCATENATION.

Now the loop of commands of object o assures basically the following
while not exhausted
do

store (I);
attach(L); – a word from L
attach(R); – precedes a word from R
detach; – hence a word of (L R) is given
restore(I)
od

with the necessary reactions to a case when one language (L or R) ends.
It is clear that if the object L and R enjoy the property mentioned in the Lemma then the object o enjoys it too.
This ends the proof of the Lemma. ¨

21.9 Przeszukiwanie z nawrotami

Dość często stosuje się zasadę poszukiwania rozwiazań, która sprowadza sie do przeglądania wszystkich możliwości. Najczęściej spotykamy sie z takim podejściem w zadaniach sztucznej inteligencji oraz w grach. Wystarczy przypomnieć znane zadanie o przewiezieniu trzech kanibali i trzech misjonarzy przez rzekę gdy do dyspozycji jest łódka pozwalająca przewieźć naraz tylko dwie osoby.

Poniżej podamy

21.10 Symulacja.

Wspólprogramy mogą być z sukcesem użyte do symulacji procesów dyskretnych. Np do symulacji epidemii grypy w pewnej populacji, do symulacji ruchu pojazdów w mieście, etc. Czyli, tam gdzie trudno zastosować wzory analityczne, natomiast znane są pewne statystyczne prawidłowości dotyczące procesów.
TODO
opisz przykład,
wytłumacz pojęcia: simproces, oś zdarzeń,
operacje: hold, schedule, passivate, run
Schemat programu stosującego klasę Simulation

```
l program backtracking;
unit backtrack: class;
hidden se;
var root:node,search:se,found:node,
number_of_leaves,number_of_answers:integer;

unit node: class(father:node);
var nsons,level: integer , deadend:boolean;
unit virtual leaf: function :boolean;
end leaf;
unit virtual answer :function :boolean;
end answer;
unit virtual lastson: function : boolean;
end lastson;
unit virtual nextson: function : node;
end nextson;
unit virtual equal : function (w:node):boolean;
end equal;
unit virtual cost: function :real;
end cost;
begin
if father =/= none
then
level:=father.level+1
else
level:=0
fi;
end node;

unit ok: function (v:node):boolean;
var w:node;
begin
if v=none then result:=false; return fi;
result:=true; w:=v.father;
while w =/= none
do
if v.equal(w) then result:=false; return fi;
w:=w.father
od
end ok;

unit purge: procedure (v:node);
var w: node;
begin
if v=none then return fi;
do
w:=v.father; kill(v);
if w=none then return fi;
w.nsons:=w.nsons-1;
if w.nsons =/= 0 then return fi;
v:=w
od;
end purge;

unit virtual insert: procedure(v:node);
end insert;

unit virtual delete : function :node;
end delete;
```

```
program Bank;
    unit FIFOqueues: class
        ...
    end FIFOqueues;
    unit PriorityQueues: class
        ...
    end PriorityQueues;
    unit Simulation: PriorityQueues class
        ...
    end Simulation;
begin
    ...
    pref Simulation block
    (* w tym bloku można wykorzystywać pojęcia i operacje
       opisane w klasie Simulation *)
    begin
        pref FIFOqueues block
        (* procesy symulowane mogą stawać w kolejkach *)
        begin

        end
    end (* Simulation *)
end
```

21.11 Poprawność klasy Simulation

W tym rozdziale pokażemy jak mozna zrealizować klasę na podstawie jej specyfikacji, konstruując przy tym dowód poprawności. Rozważać będziemy dwie specyfikacje: specyfikację Symulacja docelowej klasy Simulation, oraz specyfikację ATPQ kolejek priorytetowych – tj. bazę na której zbudowana będzie klasa Simulation.

21.11.1 Introduction

In earlier articles we discussed the problems related to the specifications. In [22] we demonstrated the risks of creating inconsistent or incomplete specifications. We made an example of complete specification. The algorithmic formulas used there allow to show that any two implementations of the specifcation are isomorphic. We also remarked that the algorithmic logic makes a formal base for proofs of correctness of algorithms. In other article [23] we demonstrate ... In these articles specifications are viewed as somewhat theoretical, abstract beings. In fact, we talk of formalized algorithmic axiomatizations. Yet, the practice of programming brings the concept similar to specifications, it is interface module.

This article is third in series. In [22] we discussed the problems related to the creation of a specification of a class. In [23] we presented a proof of correctness of a class w.r.t. a specification.

Now, we illustrate that in some circumstances one may build a class using only knowledge of two specifications: specification $\mathcal{S}_A$ of the base class A and a

specification $\mathcal{S}_B$ of a target class B that inherits the class A. No knowledge on the body of the inherited class is needed.

The meaning of this result is the following: the created class B will be correct with any class A provided it correctly implements the specification $\mathcal{S}_A$. One possible application of this result is a quick construction of software. A prototype class A may be used in order to quickly construct the system consisting of classes A and B. Later, one may replace the class A by a more efficient implementation.

A comment is in place here: our specifications can be compared with the interface modules of Java. It turns out that:

1° Interfaces limit themselves to the signature part of an specification. The specification files .spec of the SpecVer system contain signatures as well as properties (or invariants, or axioms) of specified classes.

2° Interfaces can not prevent misinterpretation of the signature. Imagine, an interface

```
interface Stacks  {
    Stacks push(Element e, Stacks s) {  }
    Stacks pop(Stacks s) {  }
    Element top(Stacks s) {  }
}
```

What will happen if a class Stacks implemnts this specification in the manner of FIFO instead of LIFO? This and similar examples demonstrate that specifications of Java do not garantee anything but that the class implementing an interface has declared some methods of given names and parameters.

3° Interfaces are not complete. Obviously, one interface I may be implemented in many ways. There is no way to express that any implementation of I must ...

In section 21.12 we give the specification for the class Simulation. Section 21.12.1 presents step by step work on implementation of the class Simulation. Appendix A contains the specification of the base class PQS. Appendix B contains an informal, yet sufficiemtly complete, specification of coroutines.

21.12 Specification of Simulation class

Now we are going to describe and to axiomatize a system of discrete event simulation. There are numerous situations in which we have to deal with processes to be simulated, for example:

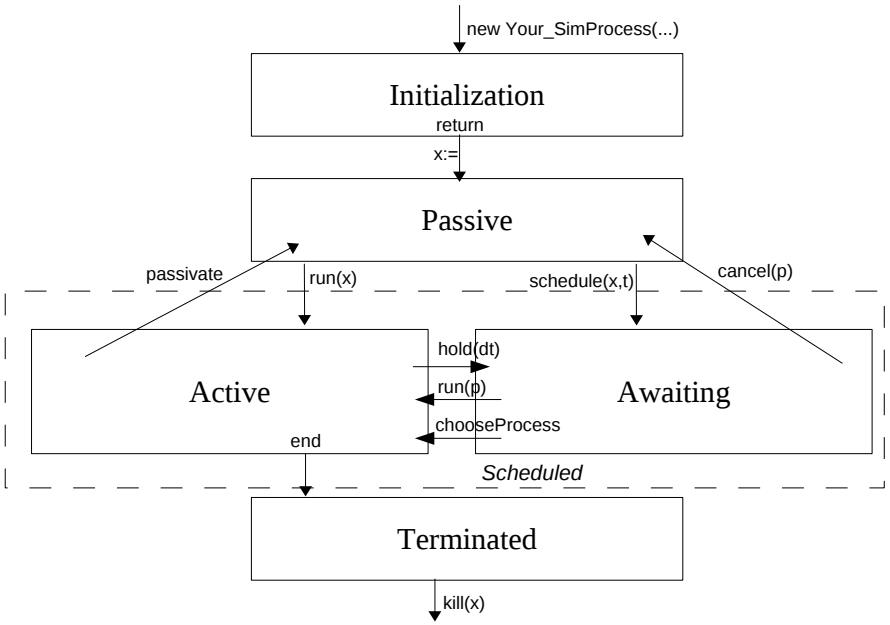
- system of patients and medecine doctors,
- system of vehicles and street lamps,
- system of ...kupno-sprzedaż

Class Simulation is meant as a base for further extensions. It offers a prototype of simulated processes and defines a set of basic operations on processes. The universe of the system *Simulation* of simulated processes and discrete events consists of four disjoint subsets: $SimProcess, EventNotice, Time$ and $SimulationPlans$. Objects of type $SimProcess$ are quasi-threads, more precisely they are coroutines. Each object of type $SimProcess$ has a thread, however only one coroutine is executed in a moment. The control passes from one coroutine to another grace the direct command attach(x).

The type $Time$ may be a predefined class.

The type $EventNotice$ has two fields: process and time.

The type $SimulationPlan$ is the data structure of priority queues of $EventNocices$



21.12.1 Constructing Simulation class

From specification's signature to the skeleton of the class Simulation

The first step is a simple,almost automatic, translation of the specification's signature onto the skeleton of the class Simulation.

Specification's signature	**Class' skeleton**

Types:
simprocess
$plan\_of\_simulation \subset PQ$
time
eventnotice
 	
Operations:
current - this process is currently active,
$current : PQ \to SP$
time - the value of currently simulated time,
$time : PQ \to T$
schedule - enables planning of events,
$schedule : (SP \times T) \times PQ \to PQ$
hold - suspend a current process for a while,
$hold : T \times PQ \to PQ$
run - immediately execute the process,
$run : SP \times PQ \to PQ$
passivate - suspends the current process,
$passivate : PQ \to PQ$
cancel - removes a process from plan,
$cancel : SP \times PQ \to PQ$
idle?
$idle : SP \to Boolean$
terminated?
$terminated : SP \to Boolean.$
	unit Simulation: PriorityQueues class
	
	unit Simprocess: elemFIFO coroutine;
	unit isIdle: function: Boolean;
	unit isTerminated: function: Boolean;
	end Simprocess;
	
	unit EventNotice: elemPQ class; ...
	end EventNotice;
	
	unit PlanSymulacji: QueueHead class;
	unit schedule : proc(p: SimProcess, t: time):
	unit hold: procedure(dt: time);
	unit run: procedure(p: SimProcess);
	unit passivate: procedure;
	unit cancel: procedure;
	unit chooseProcess: procedure;
	unit currentProcess: function: SimProcess;
	unit currentTime: function: time;
	var currProcess: SimProcess, currTime: Time;
	end PlanSymulacji;
	
	unit Time: class ... end Time;
	
	var SQS: PlanSymulacji;
	end Simulation;

Below we gathered the postulates (invariants, axioms) the class Simulation should obey.

$SQS\ is\ a\ finite\ set.$ (S1)

$EventNotice = SimProcess \times Time$ (S2)

$SQS.currentProcess = (SQS.min\ qua\ EventNotice).p$ (S3)

$SQS.currentTime = (SQS.min\ qua\ EventNotice).t$ (S4)

$\neg\, idle(p, pq) \Rightarrow (\exists t) member((p, t), pq)$ (S5)

$\forall_{pq \in SimulationPlan} \forall_{p \in Simprocess} member((p, t_1), pq) \wedge member((p, t_2), pq) \Rightarrow t_1 = t_2$ (S6)

$pq = c \wedge idle(p, pq) \wedge \neg terminated(p, pq) \Rightarrow$ (S7)
$[callschedule((p, t), pq)] \neg idle(p, pq) \wedge pq = insert((p, t), c)$

$(pq = o \wedge \neg idle(p, pq) \wedge \neg terminated(p, pq)) \Rightarrow$ (S8)
$[callschedule((p, t), pq)](idle(p, pq) \wedge pq = insert((p, t), delete((p, time), o)),$

$terminated(p, pq) \Rightarrow [callschedule((p, t), pq)]\{ERROR\},$ (S9)

$[\text{call } hold(t, pq)]\ \alpha \equiv [\text{call } schedule((current(pq), time + t), pq)]\ \alpha,$ (S10)

$(current(pq) = p' \wedge \neg terminated(p, pq)) \Rightarrow$ (S11)
$\{[callrun(p, pq)]\alpha \equiv [callschedule(p, time, pq)]\alpha\}$

$(p = current(pq) \wedge pq = o) \Rightarrow$ (S12)
$[\text{call } passivate(pq)](idle(p, pq) \wedge pq = delete((p, time), o))$

$pq = o \Rightarrow [\text{call } cancel(p, pq)](idle(p, pq) \wedge pq = delete((p, t), o)).$ (S13)

We made the following decisions:

- We introduce the class PlanSymulacji derived upon the class Queuehead which implements priority queue [23].
- Instead of functions $schedule, run, hold, passivate, etc.$ of type PQ with one argument of type PQ we declare methods $schedule, run, hold, passivate, etc.$ within class PlanSymulacji. In this way we spare transferring argument and receiving the result. This simplifies the body of class Simulation and the usage of it.

The full text of the first version is here.
Now we ought to fill the bodies of methods schedule, run, hold, etc in such a way that the properties S1 – S13 are valid.

Własność S1

$$SQS \text{ is a } finite \text{ set.} \tag{S1}$$

Remark that the property S1 is garanteed. For the variable SQS points to a priority queue object.

Własność S2: EventNotice = Simprocess × Time

$$EventNotice = SimProcess \times Time \tag{S2}$$

Własność S2 says: objects of type $EventNotice$ are pairs $\langle s, t\rangle$ where $s \in SimProcess$ and $t \in Time$. $EventNotice$ objects are inserted into priority queue. Hence they need an ordering relation. We use the following definition:
$e1 \leq e2 \stackrel{df}{\equiv} e1.t \leq e2.t$
The class $EventNotice$ takes the following form

```
unit EventNotice: elemPQ class(p: SimProcess, t: Time);
    unit less: virtual function(e: EventNotice): Boolean;
    begin
        result:= t ≤ e.t
    end less;
end EventNotice;
```

The full text of the second version is here.

Własności S3 i S4

We shall prove that in each state of SimulationPlan SQS and hence in each moment of a simulation experiment the following two properties hold.

$$SQS.currentProcess = (SQS.min \ qua \ EventNotice).p \tag{S3}$$

and

$$SQS.currentTime = (SQS.min \ qua \ EventNotice).t \tag{S4}$$

To assure this, we put the instruction
call *chooseProcess*;
as the last instruction in the operations: $hold, run, passivate$.
For example,

```
unit hold: procedure(dt: time);
begin
    ...
    call chooseProcess;
end hold;
```

The instruction *chooseProcess* has to select from the priority queue SQS the eventnotice of the minimal time and to activate the process named in this eventnotice. Moreover information on the chosen process and on the time of chosen eventnotice are to be accessible as the values of function designators: *cuurrentProcess* and *currentTime*. We achieve this by declaring private variables: *currProcess* and *currTime* and makking their values available through the methods *currentProcess* and *currentTime*.

```
unit chooseProcess: procedure;
    var e: EventNotice;
begin (* value of SQS.min is the least element of priority queue *)
    e:=SQS.min qua EventNotice; (* projection qua EventNotice is needed here *)
    (* variables currTime i currProcess are private variables of the object SQS *)
    currProcess:= e.p;
    currTime := e.t;
    attach(e.p);
end chooseProcess;
```

We can not not forget to declare method *currentProcess*.

```
unit currentProcess: function: SimProcess;
begin
    result := currProcess;
end currentProcess;
```

In a similar way we declare method *currentTime*. The reader sees that properties S3 and S4 are valid. The full text of the third version is here.

Własność S5

$$\neg idle(p, pq) \Rightarrow (\exists t) member((p, t), pq) \tag{S5}$$

In words, *every not suspended process is planned for certain time t*. This property requires that information whether an object s of type $SimProcess$ has an eventnotice in the $SimulationPlan$ or not were known to the object itself. The simplest way to achieve this is to put the eventnotice $\langle s, t\rangle$ in the object s. Now, the function *idle* answers correctly by checking the value of the variable *event*.

```
unit SimProcess: elemFIFO coroutine;
    var event: EventNotice; (* make sure that, event.p = this SimProcess *)
    unit isIdle: function: Boolean;
    begin
```

```
        result := (event=none); (* not scheduled iff event = none*)
    end isIdle;
end SimProcess;
```

The full text of the fourth version is here.

Własność S6

$$\forall_{pq \in SimulationPlan} \forall_{p \in Simprocess} member((p,t_1),pq) \wedge member((p,t_2),pq) \Rightarrow t_1 = t_2 \tag{S6}$$

In words, *in every simulation plan every process can be planned at most once.* Własność S6 will follow from the analysis of methods $schedule, hold, run$, for they are inserting an eventnotice to the plan of simulation.

Własność S7

Własność S7 reads:

$$pq = c \wedge idle(p,pq) \wedge \neg terminated(p,pq) \Rightarrow [callschedule((p,t),pq)] \neg idle(p,pq) \wedge pq = insert((p,t),c) \tag{S7}$$

In words, *a suspended process can be scheduled to be reactivated at time t. Note it is the time of simulation system.*

$$terminated(p) \equiv objectpexecutedallitsinstructions$$

We shall write a constructor (empty) and a thread of the coroutine $SimProcess$.

```
begin
    return; (* end of constructor, constructor is empty *)
    inner; (* it is a place for the thread of derived class *)
    finished :=true; (* thread is terminated *)
    call passivate;
    raise Error; (* at the attempt to activate a terminated simprocess object *)
end SimProcess;
```

A provisory variant of procedure *schedule* may look as follow:

```
unit schedule: procedure(p:SimProcess, t: time);
    var ev: EventNotice;
begin
    ev := new EventNotice(p, t);
    if p.idle and not p.terminated then call SQS.insert(ev); p.event:=ev; endif;
end schedule;
```

The full text of the fifth version is here.

Własności S8 i S9

$$(pq = o \wedge \neg idle(p,pq) \wedge \neg terminated(p,pq)) \Rightarrow [callschedule((p,t),pq)](idle(p,pq) \wedge pq = insert((p,t), delete((p,time \tag{S8}$$

A scheduled already process p can be scheduled again for another time, this will delete an earlier eventnotice for p.

$$terminated(p,pq) \Rightarrow [callschedule((p,t),pq)]\{ERROR\}, \tag{S9}$$

An attempt to schedule a terminated process results in an error. Properties S8 and S9 require the correct, full version of operation *schedule*.

```
unit schedule: procedure(p:SimProcess, t: time);
    var ev: EventNotice;
begin
    if p. terminated
    then
        raise ErrorDo_not_ScheduleTerminatedProcess;
    else
        ev := new EventNotice(p, t);
        if not p.idle
        then
            call SQS.delete(p.event);
        endif;
        call SQS.insert(ev);
        p.event:=ev;
    endif;
end schedule;
```

As it is easy to observe the operation *schedule* defined in this way satisfies properties S7 and S8. We should react when the parameter t has the value less than *currentTime*.

The full text of the sixth version is here.

Własność S10

$$[\text{call } hold(t,pq)]\ \alpha \equiv [\text{call } schedule((current(pq), time + t), pq)]\ \alpha, \tag{S10}$$

for arbitrary formula α.

A hold operation suspends the current process and schedules its activation after t units of time. The property leads directly to the following body of the procedure hold.

```
unit hold: procedure(dt: time);
begin
    call SQS.schedule(currentProcess, currentTime+dt);
    call SQS.chooseProcess;
end hold;
```

The first instruction causes suspension of the current simprocess' thread for dt units of time. The second instruction will choose a new active simprocess object. The full text of the seventh version is here.

Property S11

$$(current(pq) = p' \wedge \neg terminated(p, pq)) \Rightarrow \{[callrun(p, pq)]\alpha \equiv [callschedule(p, time, pq)]\alpha\} \quad \text{(S11)}$$

for arbitrary formula α.

Procedure run immediately activates the indicated simprocess p.

```
unit run: procedure(p: SimProcess);
begin
    if p.terminated
    then
        raise ErrorDo_not_ScheduleTerminatedProcess;
    endif;
    call hold(0.1); (* wstrzymaj na chwile bieżacy proces *)
    call schedule(p, currentTime);
end run;
```

The instruction call run (x) results in immediate activation of the simprocess object x.

The full text of the eigtht version is here.

Własność S12

$$(p = current(pq) \wedge pq = o) \Rightarrow [\text{call } passivate(pq)](idle(p, pq) \wedge pq = delete((p, time), o)) \quad \text{(S12)}$$

Instruction passivate removes the current simprocess from the plan of simulation.

```
unit passivate: procedure;
    var s: Simprocess;
begin
    s := currentProcess;
    call SQS.delete(s.event);
    s.event :- none;
    call SQS.chooseProcess;
end passivate;
```

The full text of the ninth version is here.

Własność S13

$$pq = o \Rightarrow [\text{call } cancel(p, pq)](idle(p, pq) \wedge pq = delete((p, t), o)). \quad \text{(S13)}$$

Instruction cancel applies to a scheduled, non-active simprocess.

```
unit cancel: procedure(p: SimProcess);
begin
    call SQS.delete(p.event);
    p.event :- none;
    call SQS.chooseProcess;
end cancel;
```

The full text of the tenth version is here.

Własność S6 ponownie

$$\forall_{pq \in SimulationPlan} \forall_{p \in Simprocess} member((p, t_1), pq) \wedge member((p, t_2), pq) \Rightarrow t_1 = t_2 \quad \text{(S6)}$$

Własność ta powiada: w każdym momencie obliczeń i dla każdego procesu s, w planie symulacji nie ma dwu zdarzeń e1 i e2 planujacych wznowienie procesu s w dwu różnych chwilach. Można sprawdzić, że procedura schedule zapewnia te własność, a w konsekwencji także pozostałe procedury planowania, które sie na tej procedurze opieraja, zapewniaja te własność

Uwagi

skad sie wział prior?
Unit Mainpr: SimProcess class

21.12.2 Wnioski

Jako jeden z natychmiastowych wniosków otrzymujemy:

Twierdzenie 21.8 (*o relatywnej poprawności*)**.** *Jeśli klasa bazowa jest modelem specyfikacji Kolejek Priorytettowych to klasa Simulation jest modelem (tj. poprawnie implementuje) specyfikację Symulacja.*

Dowód. Wynika z konstrukcji klasy Simulation. □

Kolejne pytanie jakie się nasuwa brzmi: czy istnieje więcej implementacji będących modelem specyfikacji Symulacja? a może wszystkie one są izomorficzne?

Twierdzenie 21.9. *Niech klasa C_{PQ} będzie modelem specyfikacji ATPQ kolejek priorytettowych. Każde dwie poprawne implementacje specyfikacji Symulacja (tj. modele) które bazują na klasie C_{PQ} są izomorficzne.*

Dla użytkownika duże znaczenie ma następujące

Twierdzenie 21.10. *W każdym kroku obliczenia ... współprogramem aktywnym jest current process.*

Dowód. Wynika z własności S1 – S13. □

A więc ...

21.12.3 Appendix A: Specification of Priority Queues

The specification of priority queues class was published in [24], we recall it here for the convenience of the reader.

Table 1. Specification $ATPQ$ of priority queues.

Signature	Comments
Sorts	$Universe = E \cup PQ$
E	set of elements
PQ	set of priority queues
Operations	let $e \in E$ and $q \in PQ$
$insert : E \times PQ \longrightarrow PQ$	put e into q
$delete : E \times PQ \longrightarrow PQ$	delete e from q
$min : PQ \longrightarrow E$	find the minimum element
$empty : PQ \longrightarrow \{true, false\}$	is a priority queue q empty?
$member : E \times PQ \longrightarrow \{true, false\}$	does $e \in q$?
$\leq : E \times E \longrightarrow \{true, false\}$	the ordering relation

Axioms

(a1) *The set E ofelements is linearly ordered by the relation* $\leq$.
(a2) [**while** *not* $empty(q)$ **do** $q := delete(min(q), q)$ **done**] $true$
This axiom says for all q program halts, i.e. the priority queue q is finite
(a3) $[q1 := insert(e, q)]\{member(e, q1) \wedge (\forall_{e1 \neq e}\ member(e1, q1) \Leftrightarrow member(e1, q))\}$
(a4) $[q1 := delete(e, q)]\{\neg member(e, q1) \wedge (\forall_{e1 \neq e}\ member(e1, q1) \Leftrightarrow member(e1, q))\}$
(a5) $empty(q) \Rightarrow (\forall_{e \in E}\ \neg member(e, q))$
(a6) $\neg empty(q) \Rightarrow (\forall_{e \in E}\ member(e, q) \Rightarrow min(q) \leq e))$
The operation min finds the least element of the set q.
(a7) $[e := min(q)]true \Leftrightarrow \neg empty(q)$
Axiom (a7) says the result of expression $min(q)$ is defined iff $\neg empty(q)$
(a8) $member(e, q) \Leftrightarrow$ **begin**
s1 := q; result := false;
while not empty(s1) and not result **do**
if e =min(s1) **then** result:=true **fi**;
s1 := delete(min(s1),s1)
done
end result

Rozdział 22

Procesy $\mathcal{L}_{10}$

22.1 Programowanie współbieżne i rozproszone $\mathcal{L}_{10}$

W programowaniu niesekwencyjnym występują dwa podstawowe pojęcia: proces i procesor.
Program sekwencyjny jest procesem. Program niesekwencyjny może utworzyc i uruchomić więcej procesów. Każdy proces jest sekwencyjny w tym sensie, że jego instrukcje wykonywane są po kolei, przez odpowiedni procesor. Procesor – fizyczny lub wirtualny zapewnia postęp w obliczeniach wykonując instrukcje procesu i dokonując zmian w pamięci.
Z tymi, nie całkiem precyzyjnymi pojęciami, możemy zróżnicować obliczenia nie sekwencyjne.

współbieżne Z obliczeniami współbieżnymi wielu procesów mamy do czynienia wtedy, gdy wykonywane są na jednym wspólnym procesorze,

rozproszone Obliczenia rozproszone wykonywane są na procesorach połączonych siecią, Z konieczności każdy procesor działa w osobnej pamięci.

równoległe Obliczenia równoległe wykonywane są na komputerze wyposażonym w wiele procesorów i wspólną pamięć. Dzisiaj, każdy procesor wyposażony jest w swoją podręczną pamięć o całkiem sporym romiarze.

Definicje przyjęte w Loglanie

W dalszych rozważaniach przyjmujemy zawężające pojęcie procesora.

Definicja 22.1. *Moduł programu podobny do deklaracji klasy, odróżniający się od tej ostatniej tym, że w miejsce słowa kluczowego class występuje słowo* ***process*** *nazywamy modułem procesu.*

Definicja 22.2. *Obiekt utworzony na podstawie modułu procesu nazywamy obiektem procesu.*

Definicja 22.3. *Procesorem jest system wykonawczy Loglanu, w skrócie VLP lub ułamek jego czasu.*
Jeśli dwa (lub więcej) procesy wykonywane są przez jeden procesor VLP, to

każdemu z nich, na zmianę, przydziela się parę milisekund czasu procesora fizycznego. Kazdy z nich otrzyma w przybliżeniu 1/2 czasu procesora.

Loglan oferuje jednolity model programowania obliczeń współbieznych i rozproszonych. Model ten obejmuje też rozmaite konfiguracje mieszane. Przez procesor będziemy rozumieć loglanowską maszynę wirtualna VLP. Na każdej takiej maszynie można wykonywac wiele programów naraz, a także można wykonywać zadania wielu procesów jednego programu. Procesory VLP mogą być w łatwy sposób łączone siecią. Zobacz... Tworzy się w ten sposób klaster (*ang.*cluster, czyli grono) maszyn wirtualnych. Program P może tworzyć obiekty procesów i alokować je na różnych procesorach. Może też je wznawiać.

Przykład

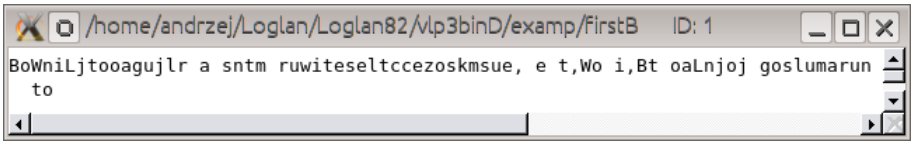
Zacznijmy od krótkiego przykładu

```
program first;
  #include "classes/gui.inc";
   (* deklaracja procesu *)
    unit writer: process(node: integer, s: string);
      var i: integer, A: arrayof char;
     begin
      A := unpack(s);
      return;
      for i := lower(A) to upper(A)
      do
        write(A(i));
      od
     end writer;
  var w1, w2, w3: writer;
begin
   (* tworzenie obiektów procesu writer *)
     w1 := new writer(0, "Bonjour tristesse Bonjour tristesse");
     w2 := new writer(0, "Welcome to London");
     w3 := new writer(0, "Willkomen in Berlin");
   (* uruchamianie obiektów w1, w2, w3 *)
     resume(w1);
     resume(w2);
     resume(w3);
   (* oczekiwanie na klawisz 'f'  *)
     pref GUI block
     begin
      while true do
        if GUI_KeyPressed=ord('f') then exit fi
      od
     end (* block *);
  call endrun
end
```

Co się składa na ten program? Elementy programu istotne dla obliczeń współbieżnych zaznaczono kolorem czerwonym.
Deklaracje w tym programie sprowadzają się do: 1°deklaracji procesu writer, i

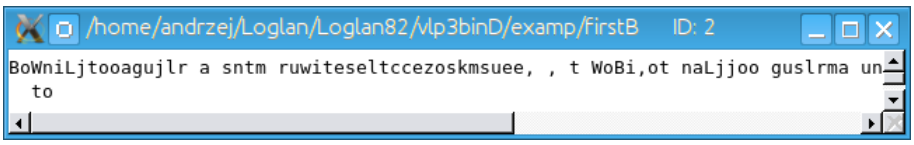
2°deklaracji trzech zmiennych w1, w2, w3 typu writer.
Działanie programu first polega na: 1°stworzeniu trzech obiektów procesu writer i przypisaniu ich do zmiennych w1, w2, w3. Każdy z tych obiektów jest alokowany na tym samym procesorze co program. (Decyduje o tym wartość pierwszego parametru równa 0.) Utworzone obiekty pozostają w stanie PASYWNY. 2°Następnie program uruchamia każdy obiekt wykonując polecenia resume. Każdy z obiektów w1, w2, w3 zaczyna wykonywać swoje zadania niezależnie od innych, drukując na ekranie odpowienią literę z zadanego tekstu. Łącznie zprogramem głównym wykonywane są cztery wątki obliczeń sekwencyjnych. 3°Program główny oczekuje, że zostanie naciśnięty klawisz 'f'. Potem program kończy działanie – call endrun.

Obserwując zachowanie tego prostego programu możemy dokonać kilku spostrzeżeń. Po pierwsze, na ekranie pojawi się mieszanina liter.



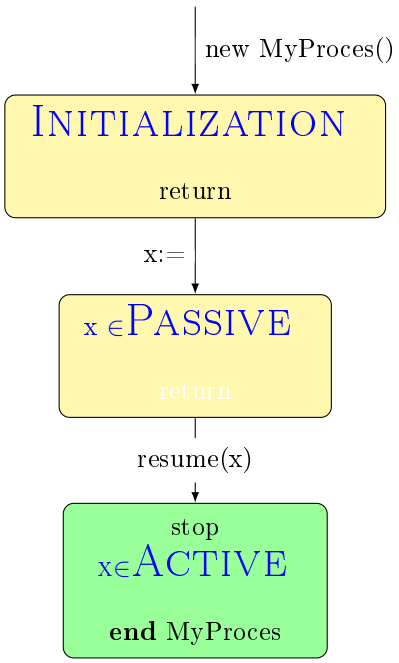
Wynik programu first (pierwsza próba)

Dlaczego tak się dzieje? To proste, wszystkie trzy obiekty procesów drukuję na tym samym ekranie.
Po drugie, jeśli powtórzysz wykonywanie tego programu, kilka lub kilkanaście razy, to zaobserwujesz, że wzór przemieszania liter może się zmienić.



Wynik programu first (druga próba)

W tym przypadku już w drugiej próbie uzyskaliśmy inny obraz. Ale to ten sam program i nie jest zależny od danych. Dlaczego więc są różnice w wydruku? Aha, obliczenia współbieżne nie są deterministyczne! Ta obserwacja ma doniosłe konsekwencje. W programowaniu sekwencyjnym analizę własności programów opieramy na zasadzie determinizmu: *wielokrotne wykonanie programu z tymi samymi danymi początkowymi da zawsze ten sam wynik*. Ale jak widać w programowaniu współbieżnym zasada determinizmu nie obowiązuje. Najważniejszą konsekwencją tego co zaobserwowaliśmy, jest całkowita nieprzydatność tzw. testowania programu. Cóż nam bowiem po tym, że na naszych komputerach program współbiezny lub rozproszony nie dał powodu do niepokoju. W nieznacznie zmienionej konfiguracji, np. przy innej temperaturze, lub gdy komputery działają z trochę inną prędkością nieczekiwany błąd może sie pojawić jak diabeł z pudełka.
Oznacza to m. in., że własność stopu ma dwa oblicza: możemy pytać czy każde obliczenie programu P jest skończone? Ale możemy też zastanawiać się czy istnieje chociaż jedno obliczenie skończone tego programu? Zamiast zadawać sobie pytanie czy po zakończeniu (deterministycznego) programu P zachodzi warunek



Rysunek 22.1: Diagram stanów obiektu procesu

β, trzeba teraz pytania formułować inaczej: czy koniecznie po zakończeniu programu M zachodzi warunek β i/lub czy możliwe jest że po zakończeniu programu M zachodzi warunek β. Język opisu zjawisk semantycznych występujących w programowaniu współbieżnym wymaga modalności: *konieczne* i *możliwe*. Ujmijmy to jeszcze inaczej: w obliczeniach determinstycznych jeśli jest możliwe, że po wykonaniu programu P zachodzi warunek α, to jest konieczne, ze po wykonaniu tego programu zachodzi warunek α.
Do zjawiska niedeterminizmu będziemy jeszcze powracać.
DO ZROBIENIA
-
-

obliczenie rozproszone odmiana programu first

semafor - program second.log

22.2 Składnia i semantyka

Poniżej przedstawiamy składnię i semantyke poleceń dotyczących procesów. Zechciej zauważyc jak niewiele potrzeba, by rozszerzyć język obliczeń sekwencyjnych tak by można było programować obliczenia współbieżne i rozproszone. Narzędzia służące programowaniu obliczeń współbieżnych i rozproszonych są naturalne, w tym sensie, że są zgodne z koncepcją programowania obiektowego. Język $\mathcal{L}_{10}$ jest rozszerzeniem poprzedniego języka $\mathcal{L}_9$.

$$\mathcal{L}_9 \subsetneq \mathcal{L}_{10}$$

Deklaracje

W tym rozszerzeniu języka znajdujemy deklaracje procesów – modułów i deklaracje zmiennych typu proces. Deklaracja modułu process jest podobna do deklaracji klasy czy współprogramu, z tym, że słowo **class** należy zastąpić przez **process**. W treści takiego modułu mogą wystąpić instrukcje zarządzające obiektami procesów (są one wyliczone poniżej).

Definicja 22.4. *Niech P będzie identyfikatorem, params – listą parametrów formalnych. Deklaracja modułu procesu ma następującą postać*

unit P: **process**(params); $\mathbb{D}$ **begin** $\mathbb{I}$ **end**

Moduł procesu może rozszerzać (tj. dziedziczyć z) moduł klasy lub współprogramu. Niech K będzie nazwą zadeklarowanej klasy.

unit K: **class**(params_K); $\mathbb{D}_K$ **begin** $\mathbb{I}_K$ **end** K;
unit P: K **process**(params_P); $\mathbb{D}_P$ **begin** $\mathbb{I}_P$ **end** P

Możliwa jest także sekwencja odwrotna w której moduł klasy dziedziczy z modułu procesu

unit P: **process**(params_P); $\mathbb{D}_P$ **begin** $\mathbb{I}_P$ **end** P;
unit K: P **class**(params_K); $\mathbb{D}_K$ **begin** $\mathbb{I}_K$ **end** K

wynik operacji rozszerzania jest modułem procesu.

W Loglanie'82 przyjęto następujące dodatkowe ograniczenia na postać deklaracji modułu procesu:

- Deklaracja procesu musi być zawarta bezpośrednio wśród deklaracji programu głównego (tj. musi być deklaracją *top-level*).
- Pierwszy parametr formalny na liście params parametrów formalnych modułu procesu musi być typu integer. Wartość pierwszego parametru aktualnego decyduje na którym procesorze będzie alokowany obiekt procesu.
- Parametry formalne procesu mogą być typu pierwotnego lub typu proces. Niedopuszczalne jest by parametr aktualny był typu tablicowego lub obiektowego (za wyjatkiem obiektów procesów).
- W module procesu nie występują zmienne nielokalne. To ograniczenie wynika w naturalny sposób, gdy zrozumiemy, że obiekty procesu mogą być alokowane na oddzielnych komputerach (połączonych siecią).

Definicja 22.5. *Niech P będzie nazwą modułu procesu. Napis postaci*

var a: P;

jest deklaracją zmiennej a typu proces P.

Deklaracje zmiennych tego samego typu P można łączyć, np. **var** a,b: P;
jest deklaracją dwu zmiennych a i b typu proces P.

Instrukcje

W języku $\mathcal{L}_{10}$ pojawia sie kilka nowych instrukcji atomowych. Natomiast zachowane zostają konstrukcje programotwórcze: **if**, **for**, **while** oraz średnik.

Definicja 22.6. *Następujące napisy są instrukcjami atomowymi w języku* $\mathcal{L}_{10}$

- *instrukcje atomowe języka* $\mathcal{L}_9$ *oraz*
- *instrukcja utworzenia obiektu procesu i przypisania go do zmiennej, np.*
  x := **new** Myproces((*node=*) 8, ...) ;
- polecenie resume, np.
  resume(x)
- polecenie call np.
  call x.meth(...)
- polecenie accept, np.
  accept meth1, meth3
- polecenie enable, np.
  enable *meth1, meth2* ;
- polecenie disable, np.
  disable *meth3* ;
- polecenie stop
  stop

Klaster procesorów VLP

zrzut ek.

Procesor VLP jest wirtualną maszyna loglanowska. Możesz mu zlecić różne zadania. Procesor VLP może łączyć się z innymi procesorami. W ten sposób możesz utworzyć klaster (*ang.* cluster).

Deklaracje procesów

Deklaracje są niezbędne by można było utworzyć obiekty procesów.
Deklaracja modułu procesu tylko nieznacznie różni się od deklaracji klasy:

i) zamiast słowa class należy napisać process,

ii) treba pamiętać, że wszystkie lokalne zmienne procesu są prywatne,

iii) pierwszy parametr formalny (tj. pierwszy argument) deklaracji jest przeznaczony na przekazanie numeru wirtualnego loglanowskiego procesora w wirtualnym klastrze loglanowskich procesorów (zob. poniżej),

Deklaracje zmiennych typu process niczym nie różnią się od deklaracji zmiennych obiektowych. Wartościami takich zmiennych są obiekty procesów.

Polecenia specyficzne procesów

Parę słów o poleceniach.
Obiekt procesu *tworzymy* wykonując polecenie x:=new Myproces(p, ...). Zwracamy uwagę na to, że pierwszy argument p wyznacza procesor na jakim będzie alokowany obiekt procesu. Typ tego argumentu musi być liczbą całkowitą.
Jeśli wartość p jest równa zero to nowotworzony obiekt procesu jest alokowany na tym procesorze, który wykonuje instrukcję. Wartość argumentu p różna od zera wskazuje na numer procesora VLP w klastrze wirtualnych procesorów loglanowskich. Nowoutworzony obiekt procesu jest PASYWNY. Tzn. jest gotów do rozpoczęcia obliczeń według swojego programu.

Zmiana stanu obiektu procesu na AKTYWNY tj. uaktywnienie lub wznowienie obiektu procesu x, ma miejsce gdy wykonano polecenie resume(x). Zwróć uwagę na to, że w odróżnieniu od obiektów współprogramów – wiele obiektów procesów może być naraz w stanie AKTYWNY. Jeśli polecenie wykonywane jest na procesorze o numerze 7, a obiekt jest alokowany na procesorze 2 to polecenie aktywacji jest przekazywane temu drugiemu procesorowi – w efekcie obiekt jest wykonywany na tym procesorze na którym był alokowany.

Obiekt aktywny może wykonywać wiele poleceń, które poznaliśmy wcześniej: instrukcje przypisania, warunkowe, iteracji etc. Instrukcje te powodują zmianę stanu obiektu procesu. Ważnym elementem stanu obiektu procesu jest dostępność (public) lub niedostępność (private) metod obiektu. Na początku wszystkie metody obiektu procesu są ukryte (private). Polecenie enable met1, met4 udostępnia wyliczone metody innym obiektom. A dokładniej dodaje te metody do zbioru już wcześniej udostępnionych (enabled) metod. Można sobie wyobrazic, że elementem stanu obiektu procesu jest zbiór metod udostepnionych, oznaczamy go MASK. Polecenie disable met2, met3 usuwa nazwy tych metod ze zbioru MASK. Możemy uważać, że odtąd metody te są prywatne i niedostępne z zewnątrz.

Obce wołanie metody

Obiekty procesów mogą współdziałać. Podstawą współdziałania jest oryginalny protokół obcego wołania metod (*ang.* alien call) wymyślony przez Bolka Ciesielskiego w r. 1988 [8, ?]. Nie spotkaliśmy podobnej koncepcji w żadnym innym języku programowania.
Składnia obcego wołania metody ma dwa składniki, jeden w obiekcie y procesu wywołującym metodę methd w innym obiekcie x procesu: call x.meth. Drugi składnik występuje w wywoływanym obiekcie x i jest to albo polecenie accept ..., methd, ...; albo polecenie enable methd;. W pierwszym przypadku mówimy o *synchronicznym* wykonaniu poleceń

y:: call x.methd(args) || x:: accept methd

W drugim przypadku mamy do czynienia z *asynchronicznym*, także wspólnym, wykonaniem polecenia call x. methd(.) przez oba obiekty procesów.

$$\text{y:: call x.methd(args)} \parallel \text{x::} \left\{ \begin{array}{l} \text{enable methd;} \\ \ldots; \\ \ldots; \\ \ldots; \end{array} \right\}$$

Jak wytłumaczyć nazwy synchroniczna lub asynchroniczna odmiana protokołu alien call?

s) zsynchronizowane wywołanie metody meth zadeklarowanej w obiekcie x pewnego procesu, ma miejsce gdy nastąpi *spotkanie* instrukcji wywołania metody z obcego obiektu procesu z instrukcją accept w tym obcym obiekcie.

a) asynchroniczne wywołanie metody meth zadeklarowenej w obiekcie x pewnego procesu, ma miejsce gdy obiekt ten umożliwił *przerwanie* wykonywania swego ciągu instrukcji wykonując polecenie enable methd. Przerwanie takie może nastapić w bliżej niekreślonym momencie, zawsze jednak po zakończeniu wykonywania instrukcji (tzn. przy średniku).

Realizacja protokołu alien call

Obiekt y pewnego procesu wykonuje polecenie call x.methd(args):

1. Sprawdzane są następujące warunki: czy obiekt x istnieje? czy jest AKTYWNY? czy metoda methd jest udostępniona (enabled)? Gdy te warunki są spełnione przechodzi się do kolejnego punktu. Jeśli nie to obiekt y oczekuje na ich spełnienie.
2. Obiekt y przekazuje argumenty obiektowi x i oczekuje na sygnał zakończenia protokołu i na wyniki od obiektu x.
3. Obiekt x
   - zapamiętuje zmienną systemowa MASK,
   - MASK :=$\emptyset$ tj. wszystkie metody obiektu są zablokowane,
   - tworzony jest rekord aktywacji metody methd,
   - wykonywana jest treść tej metody,
   - po zakończeniu odsyła wyniki – argumenty **out**,
   - odtwarzana jest wartość zmiennej MASK,
   - wykonywane jest polecenie **return ... enable ... disable ...;**
   - obiekt wykonuje swoją kolejną instrukcję i
4. obiekt y kontynuuje swoje obliczenie

Komentarza wymaga polecenie **return enable ... disable ...;** Polecenia tego nie można podzielić, gdyż w takim przypadku część **enable ... disable ...** nie zostanie wykonana lub jej efekt zostanie zmarnowany. Zastanów sie dlaczego tak by było?
Na czym polega wykonanie polecenia accept met3, met3?

1. do zbioru udostępnionych metod dołączane sa nazwy metod met2 i met3,

2. jesli jakiś inny proces wywołał metodę znajdująca sie w zbiorze metod udostępnionych (enabled) to rozpoczyna się wykonywanie tej metody, patrz wyżej. (Zauważ, proces x jest aktywny i spełnione są warunki ..)

3. w przeciwnym przypadku proces oczekuje na wywołanie którejś z metod udostępnionych.

Można to wysłowić inaczej proces x nie rozpocznie wykonywania instrukcji po instrukcji accept dopóki nie zostanie zrealizowane jedno z poleceń call ...

22.3 Program Rozmowa

Przedstawiamy poniżej program umożliwiający rozmowę dwu osób poprzez sieć. Dla działania programu niezbędne jest by w klastrze połączonych wirtualnych procesorów loglanowskich znajdowały sie conajmniej dwa procesory VLP. Dla potrzeb tego programu przyjmijmy, że procesory VLP działają na węzłach o numerach 2 i 12. Program główny wykonywany jest przez procesor na węźle nr 2. Struktura tego programu jest bardzo prosta:

- program główny tworzy dwa obiekty rozmówca, na węzłach 0 i 12. Obiekt procesu alokowany na węźle 0 jest faktycznie alokowany i wykonywany na węźle nr 2. Tworzone są też dwa obiekty procesu ekran na tych samych węzłach.

- obiekt procesu rozmówca odbiera znaki naciśnięte na klawiaturze i wysyła je do obu ekranów: lokalnego i zdalnego,

- obiekt typu ekran jest serwerem mastępujących usług:

  - wydrukuj przysłany znak w części lokalnej ekranu,
  - wydrukuj przysłany znak w części zarządzanej przez drugiego rozmówcę.
  - skasuj ostatnio wydrukowany znak (Backspace) (lokalnie i zdalnie),
  - zakończ pracę gdy naciśnięto klawisz Esc (escape) i prześlij odpowiedni sygnał do pozostałych procesów.
    To jest sposób na realizację rozproszonego zakończenia całego programu – wszystkich jego procesów.

Proces ekran

Najpierw tworzone są dwa obiekty procesu ekran. Argumentem ekranu jest rozmówca, który nim zarządza. Ale w trakcie tworzenia obiektu ekran nie znany jest obiekt rozmowca. (Nie istnieje jeszcze żaden taki obiekt.) Aby temu zaradzić zarzadziliśmy co następuje:

- obiekt ekran posiada metodę zarejestruj, która przyjmuje argument typu rozmowca i przypisuje go do lokalnej zmiennej rozm obiektu. Pierwszą instrukcją obiektu ekran jest accept zarejestruj. W ten sposób obiekt oczekuje na przysłanie mu nazwy właściciela.

- Program główny dopilnuje by po utworzeniu obiektów procesu rozmówca przekazać ich nazwy do obiektów procesu ekran. Zob. poniżej.

Deklaracja procesu:

```
unit ekran : ANSI process(node:integer;rozm:rozmowca);

    unit rejestruj : procedure(r:rozmowca);
    begin
      rozm := r;
    end rejestruj;
    unit odbior_zdalny : procedure(s:char);
    begin
      call GotoXy(hy, hisline);
      call Bold;
      writeln(s);
      hy := hy+1;
      if s=chr(13) then hisline := hisline + 1;hy:=1; fi;
      if hy = 80 then hy := 1; hisline:=hisline+1 fi;
      call Normal;
      if hisline = 12 then hisline := 1;
      fi;
    end odbior_zdalny;
    unit odbior_lokalny : procedure(s:char);
    begin
      call GotoXY(my, myline);
      write(s);
      my := my+1;
      if s=chr(13) then myline := myline + 1;my:=1; fi;
      if my = 80 then my := 1; myline := myline+1 fi;
      if myline = 25 then myline := 13;
      fi;
    end odbior_lokalny;
    unit kasuj_lokalny : procedure;
    begin       my := my - 1;
      if my<0 then my := 0 fi;
      call GotoXY(my, myline);
      write();
    end kasuj_lokalny;
    unit kasuj_zdalny : procedure;
    begin
      hy := hy - 1;
      if hy<0 then hy := 0 fi;
      call GotoXY(hy, hisline);
      write();
    end kasuj_zdalny;
    unit koniec : procedure;
    begin
      knc := true;
    end koniec;
    var knc:boolean, myline,hisline,my,hy:integer, s:char, name:string;
  begin
    knc := false;
    myline := 13;
    hisline := 1;
    my := 1;
    hy := 1;
    call Clear;
    call GotoXY(1, 12);
    call Bold;
    write("————————————-ESC - koniec ————————————");
    call Normal;
    return;

    accept rejestruj;
    while not knc do
      accept odbior_lokalny, odbior_zdalny, kasuj_lokalny, kasuj_zdalny, koniec;
    od ;
    call rozm.koniec;
  end ekran;
```

Poniżej cytujemy treść procesu rozmowca.

```
unit rozmowca : IIUWgraph process(node: integer;el,ez: ekran);
  (* z klasy IIUWgraph wykorzystamy funkcję  inkey *)
  var knc:boolean, s:char, i:integer;
  unit koniec :  procedure;
  begin
  knc := true;
  end koniec;
begin
  knc := false;
  return;
  enable koniec;
  while not knc
  do
    i := inkey;
    if i<>0 then
    s := chr(i);
    if i=8 then
     call el.kasuj_lokalny;
     call ez.kasuj_zdalny;
    else
     call el.odbior_lokalny(s);
     call ez.odbior_zdalny(s);
    fi;
    if i = 27 then
     call ez.koniec;
     call el.koniec;
    fi;
  fi;
  od;
end rozmowca;
```

A tu jet program główny

```
program talk;
  #include "classes/ansi.inc" (* dołącz tekst klasy ANSI *)
  unit ekran:  process ...
  unit rozmowca:  process ...
  var p1,p2:rozmowca, e1,e2:ekran;
begin
  e1 := new ekran(0,none);
  resume(e1);
  e2 := new ekran(12,none);
  resume(e2);
  p1 := new rozmowca(0,e1,e2);
  p2 := new rozmowca(12,e2,e1);
  call e1.rejestruj(p1);
  call e2.rejestruj(p2);
  resume(p1);
  resume(p2);
end
```

W klasie ANSI znajdują się m.in. następujące procedury:

```
unit ANSI: class;
    unit GotoXY: procedure(x,y:integer);...
    unit SetColor: procedure(col:integer); ...
    unit SetBackground: procedure(col:integer); ...
    unit Bold: procedure; ...
    unit Normal: procedure; ...
end ANSI;
```

Znaczenie operacji wykonywanych przez te procedury jest łatwe do odgadnięcia.

22.4 Producenci i konsumenci

Przedstawiamy analizę kilku odmian zadania producent – konsument. Rozwiązania wykorzystują mechanizm obcego wołania procedury (*ang.* alien call). Analiza poprawności rozwiazań staje się łatwiejsza dzięki alien call.

22.4.1 Zadanie - wielu producentów, jeden konsument

Jest to jeden z najczęściej omawianych problemów programowania współbieżnego i rozproszonego. Zakładamy, że pewna liczba aktywnych obiektów procesu Producent wytwarza dane – obiekty i składa je w kontenerze k. Obiekt aktywny procesu Konsument pobiera po kolei dane z kontenera i je przetwarza.
Kontener – kolejka – jest własnością konsumenta
Należy zapewnić by żaden obiekt nie był przetwarzany dwukrotnie, by żaden obiekt nie przepadł i by informacja składana w kontenerze nie uległa deformacji np. podczas równoczesnego odczytywania i zapisu danych.

22.4.2 Rozwiązanie z wykorzystaniem alien call

Proces Producent:
while needed do
produkuj ;
wyślij do Konsumenta
od

proces Konsument
ma kontener k - kolejka, metodę odbierz i działa w pętli

```
unit queue:class(type element; size:integer);
   (* Ta pomocnicza klas implementuje kolejkę FIFO.
   Element jest nazwą typu elementów kolejki *)
   unit insert:procedure(e:element); ...
   (* wstaw element do kolejki *)
   unit delete:function:element; ...
   (* podaj pierwszy elemnt i usuń go z kolejki *)
   unit isempty:function:boolean; ...
   (* sprawdź czy kolejka jest pusta *)
   unit isfull:function:boolean; ...
   (* sprawdź czy kolejka jest pełna *)
end queue;
```

Poniżej przedstawiamy moduł procesu Konsument.

```
unit Konsument:process(node:integer);
   var Q: queue, f: product;
   unit deposit: procedure(f:product);
   begin
      call Q.insert(f);
      if Q.isfull
      then
         return disable deposit
      fi;
   end deposit;

begin
   Q := new queue(product, 50);
   return;
   do
      (* withdraw an element if any *)
      (* await for an element if empty *)
      disable deposit;
      if Q.isempty
      then
         accept deposit
      fi;
      f := Q.delete;
      enable deposit;
      (* consumption of the product f *)
      ...
   od
end Konsument;
```

Deklaracja modułu procesu Producent wygląda tak:

```
unit Producent: process(node: integer, k: Konsument);
  var p: product;
begin
  do
    (* produce product f *)
    ...
    call k.deposit(p)
    (* producer awaits for the completion of call k.take(.) *)
    (* it may be longer if the queue k is full *)
  od
end Producent;
```

A oto program główny, zawiera on moduły procesów Konsument i Producent. Zakładamy, że typ product należy zastąpić przez jedno ze słów kluczowych: integer, real, string lub char lub Boolean. Czyli jeden z typów pierwotnych języka. Utwórz obiekt k procesu Konsument i pewną liczbę obiektów procesu Producent. Wykonaj operację resume() na każdym z tych obiektów. Każdy z nich staje sie AKTYWNY Wszystkie obiekty procesów działają niezaleznie.

```
program Producenci_i_Konsument;
  unit Konsument: process ...
  unit Producent: process ...
  unit Queue: class ...
  var k: Konsument, p:Producer;
begin
  k := new Consument(0); resume(k);
  for i := 1 to noProducers do
    p := new Producer(nr, k);
    resume(p)
  od;
  (* zakoncz *)
end program
```

22.4.3 Analiza

Następujące pytania nasuwają się podczas czytania tego programu:

1. Przypuśćmy, że kilka obiektów procesu Producent równoczesnie przesyła swe produkty do obiektu k procesu Konsument wykonując polecenia dwu producentów p i q, np.

   call k.deposit(g) || **call** k.deposit(f)

   Czy możemy zapewnić, że żadne z tych żądań nie będzie zgubione lub, że dojdzie do równoczesnego wykonywania operacji insert na kolejce i , że w ten sposób w kolejce pozostanie tylko jeden z dwu wstawianych produktów?

2. Przypuśćmy, że obiekt procesu Konsument pobiera produkt z kolejki i że równocześnie jeden (lub więcej) producent wstawia produkt do kolejki. Mogłoby to spowodować błąd (błędy).

k:: f := Q.delete || p:: **call** k.deposit(g)

Odpowiedzi na te pytania przynosi następujący

Lemat 22.1. *Żadne żądanie nie zostanie pominięte. Nie dojdzie do równoczesnego wykonania dwu instancji procedury* deposit *(na rzecz dwu różnych producentów). Nie dojdzie do równoczesnego pobierania elementu z kolejki przez konsumenta i wstawiania innego elementu do kolejki przez producenta.*

Dowód. Należy rozpatrzyć cztery przypadki.

A) Nie dojdzie do równoczesnego wykonywania dwu instancji procedury deposit ponieważ protokół obcego wołania metody zaczyna się od zapamiętania stanu metod udostępnionych (enabled) i od ukrycia (disable) wszystkich jego metod. Tzn. jeśli konsument wykonuje operację deposit na zlecenie jednego z producentów, to wykonywanie tej procedury nie będzie przerwane przez wykonywanie takiej procedury na rzecz innego producenta.

B) Nie dojdzie do równoczesnego pobierania elementu z kolejki przez konsumenta i wstawiania nowego elementu do kolejki przez jakiegoś producenta, ponieważ instrukcje pobierania sa poprzedzone instrukcją disable deposit.

C) W sytuacji przepełnienia kolejki metoda deposit kończąc wykonuje polecenie return disable deposit. Oznacza to, że kolejne żądania call k.deposit(p) wstawienia elementu do kolejki zostaną wstrzymane do czasu gdy w kolejce zwolni się miejsce.

D) W sytuacji pustej kolejki obiekt k procesu Konsument rozpocznie wykonywanie instrukcji accept deposit, która zakończy się, gdy kolejka będzie nie pusta.

Nie może dojść do zagłodzenia jakiegoś producenta - zapewnia to maszyna wirtualna VLP. □

22.4.4 Comments

Ponieważ w obecnej wersji Loglanu nie można przekazać kopii obiektu z jednego obiektu aktywnego do drugiego, to musimy posłużyć się następującą protezą:
proces Producent wywołuje u Konsumenta metodę "przekazuję" z wszystkimi parametrami potrzebnymi do stworzenia kopii obiektu o przez Konsumenta.
call k.przekazuję(<params>)
a metoda ta wykonuje polecenie
aux := new Obiekt(<params>) i składuje aux w kontenerze.
Jest to proteza, TAK. Ale o ile prostsza od serializacji i "marshalling" w Javie RMI.

22.4.5 Jeden producent, wielu konsumentów

ćwiczenie

W tym przypadku kontener – to jest znowu kolejka – jest własnością producenta. To konsumenci adresują swoje potrzeby do producenta. Rozwiązanie jest symetryczne w stosunku do poprzedniego. Napisz program i uruchom go. Przeprowadx analize poprawności Twego rozwiązania.

22.4.6 Wielu producentów, wielu konsumentów

I jeszcze jeden obiekt aktywny kolejka.
W przypadku gdy jest wielu producentów i wielu konsumentów, warto utworzyć aktywny obiekt procesu Bufor, zarządzający kolejką, z metodami *deposit* – dla producentów i *withdraw* – dla konsumentów. Poniżej przytaczamy deklarację procesu Bufor:

```
unit Buffer: process(node:integer);
  var q:kolejka;
  unit deposit: procedure();
  end deposit;

  unit withdraw: function():product;
  end withdraw
begin
  q := new kolejka();
  return;
  do
    if q.isempty() then accept deposit
    else
    if q.isfull() then accept withdraw
    else
      accept deposit, withdraw
    fi
  fi
  od;
end Buffer;
```

Program główny wygląda teraz tak

```
program Konsumenci_i_Producenci;
  unit Konsument: process ...
  unit Producent: process ...
  unit Bufor: process ...
  var k: Konsument, p: Producent, b: Buffer, i: integer;
  const noProducers=29, noConsumers=15;
begin
  (* create one Buffer and some number of Producers and Consumers *)
  (* put them into action *)
  b := new Buffer(0); resume(b);
  for i := 1 to noProducers do
  p := new Producent(nr, b);
  resume(p)
  od;
  for i := 1 to noConsumers do
  k := new Konsument(nr, b);
  resume(k)
  od;
  (* zakończ *)
end program
```

Analiza

W tym programie obiekt b zachowuje się jak monitor. Pełni rolę dostawcy usług deposit i withdraw. Ponadto dopilnuje by w sytuacji pustego bufora konsumenci poczekali aż któryś producent wstawi produkt do bufora. I w analogicznej sytuacji gdy bufor jest pełny, nasz monitor dopilnuje by producenci poczekali na to by któryś z konsumentów pobrał produkt z kolejki tworząc miejsce na następny produkt.
Z własności obcego wołania metod wynika, że nie dojdzie do równoczesnego wykonywania metod deposit i withdraw. Podsumowując stwierdzamy

Lemat 22.2. *W systemie producentów i konsumentów*

(i) każde żądanie zostanie obsłużone w odpowiednim czasie,

(ii) nie dojdzie do pomieszania komunikatów.

Gdy producenci tworzą duże obiekty

np. z dużymi tablicami.
Proponuję by wtedy nie przekazywać kopii takich obiektów, lecz by były to obiekty aktywne (dokładniej – obiekty procesów, niekoniecznie z własnym wątkiem). Mogą to być serwery usług i danych. Wtedy przekazywanie od producenta do konsumenta – poprzez bufor – jest łatwe: przekazujemy wskaźnik do takiego obiektu aktywnego. Bufor może stworzyć strukture danych jaką programista mu wybierze – tablice lub drzewo. To w niczym specjalnie nie przeszkadza. Można mieć
var A: **arrayof** Produkt;
gdzie produkt jest **process**em np.
unit Produkt: **process**(...); ... **end** Produkt;

A co z obliczeniami równoległymi?

Nie mamy implementacji dla obliczeń równoległych gdy obiekty procesów są alokowane na procesorach/rdzeniach i wykonują się we wspólnej pamięci.

Podejrzewam, że stosuje się wtedy rozwiązania właściwe dla obliczeń współbieżnych. Może to i dobre podejście.

Czy rozproszenie może imitować równoległość?

Gdy proces jest i konsumentem i producentem

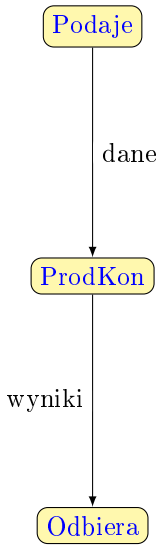
Rozważmy sytuację, w której jeden proces PASS wykonuje się na szybkim procesorze i pobiera (powoli) dane z procesu INP. Ten sam proces PASS wysyła dane (powoli) do procesu OUTP.

obrazek1

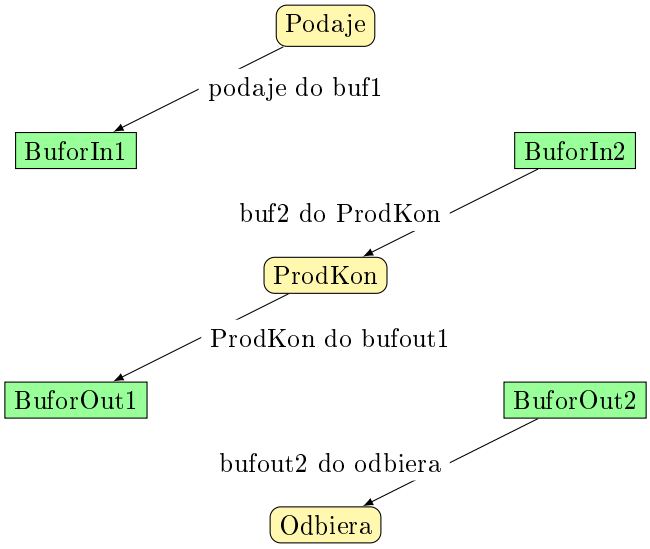
Ustanowienie pojedynczego bufora pomiędzy PASS a INP nie zapewni istotnego przyśpieszenia. Jeśli jednak powołamy do życia dwa obiekty procesu BUFORIN to być może zyskamy na czasie.

obrazek2

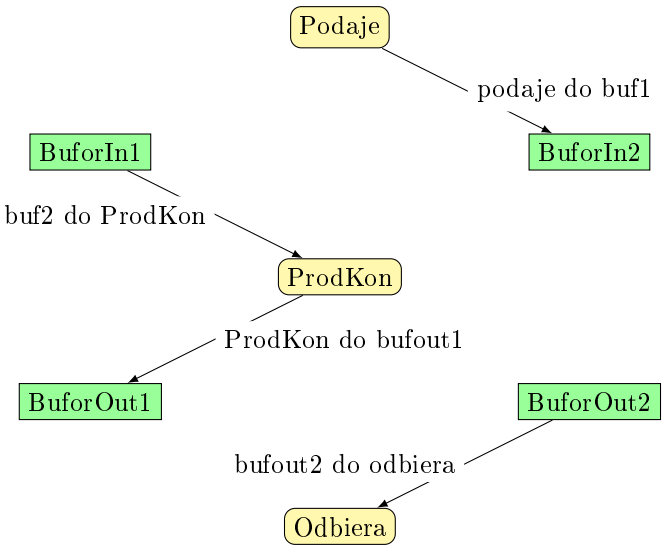
Teraz obiekt PASS



Rysunek 22.2: Proces ProdKon i dwaj partnerzy



Rysunek 22.3: Proces ProdKon i dwie pary buforów



Rysunek 22.4: Inna chwila

22.5 Czytelnicy i pisarze

Omawiamy znany problem czytelników i pisarzy, jak zapewnić bezpieczny dostęp do wspólnego, globalnego zasobu unikając przy tym zagłodzenia pisarzy i czytelników.Materiał tu przedstawiany pochodzi z pracy Bolka Ciesielskiego [8]. Zechciej porównać z rozwiązaniem stosującym semafory [7].

22.5.1 Rozwiązanie z możliwością zagłodzenia

OPISAĆ – naszkicować straightforward solution –
strukturą może być tablica lub drzewo np. drzewo BST.
operacje czytania to szukaj lub modyfikuj rekord, bez zmiany klucza
operacje pisania to insert i delete - zmieniają strukturę

```
unit STR: process();
   unit czytaj: procedure ...
   begin
      enable czytaj
      ...
   end czytaj;
   unit pisz: procedure ...
begin
   return
   do
      accept czytaj, pisz
   od
end STR;
```

W systemie może istnieć wiele aktywnych obiektów typu Klient. Np.

```
unit Klient: process(node: integer, srw: STR);
...
end Klient
```

Każdy aktywny obiekt typu Klient może wykonać polecenie bądź call srw.czytaj() bądź call srw.Pisz(). Posługując się wlasnościami protokołu *alien call* można wykazać, że nie dojdzie do konfliktu czytelników z pisarzami. Umieszczona na początku procedury Czytaj instrukcja enable Czytaj "*otwiera drogę*" następnemu czytelnikowi.
Rozwiązanie takie nie wyklucza jednak zagłodzenia pisarza(y).

22.5.2 Rozwiązanie bez zagłodzenia pisarzy

W tym rozwiązaniu wprowadzony jest obiekt procesu Supervisor, który kontroluje dostep do zasobu. W celu uniknięcia zagłodzenia pisarzy do czytania wpuszcza się czytelników grupami. Gdy wszyscy czytelnicy z grupy zakończą czytanie sprawdza się czy jakiś pisarz zgłosił chęć pisania.
Poniżej przedstawiamy schemat procesu Reader.

```
unit Reader:process(node: integer, sv:supervisor);
begin
  return;
  do
    call sv.stamp; (* zarejestruj się *)
    call sv.startread; (* żądanie dostępu do zasobu *)
    (* Czytanie *)
    call sv.endread; (* czytanie zakończone *)
    (* Inne czynności *)
  od
end Reader;
```

Proces Writer ma podobną strukturę.

```
unit Writer:process(node:integer, sv:supervisor);
begin
  return;
  do
    call sv.startwrite; (* żądanie dostępu do zasobu *)
    ...
    (* Pisanie *)
    ...
    call sv.endwrite; (* zakończenie pisania *)
    ...
  od
end Writer;
```

Utrzymaniem porządku zajmuje się obiekt procesu Supervisor.

```
unit Supervisor:process(node:integer);
  var waiting_readers:integer, (* liczba zarejestrowanych czytelników *)
    writing:boolean, (* TRUE if a writer is writing *)
  (* the following variables concern the group of readers serviced
  in a single supervisor cycle *)
  cr:integer, (* total number of readers *)
  br:integer, (* number of readers which started reading *)
  er:integer; (* number of readers which finished reading *)
  unit stamp:procedure;
  begin
    waiting_readers := waiting_readers+1;
  end stamp;
  unit startread:procedure;
  begin
    br := br+1; (* next reader started the reading *)
    writing := false; (* no writer is writing *)
  end startread;
  unit endread: procedure;
  begin
    er := er+1; (* a reader finishes the reading *)
  end endread;
  unit startwrite:procedure;
  begin
    writing := true; (* a writer is writing *)
  end startwrite;
  unit endwrite:procedure;
  end endwrite;
begin
  return;
  do
    br := 0; er := 0;
    accept startread, startwrite;
    if writing (* if a writer was first *)
    then (* then we wait until it finishes the writing *)
      accept endwrite; (* and finish the service cycle *)
    else
      disable stamp;
      cr := waiting_readers+1; (* the number of waiting readers*)
      waiting_readers := 0; (* additional readers will be *)
      (* serviced in the next readers group *)
      enable stamp;
      while br < cr
      do
        accept startread, endread;
      od;
      while er < cr
      do
        accept endread
      od;
    fi;
  od
end supervisor;
```

Struktura programu głównego wygląda tak:

```
program Czytelnicy_i_Pisarze;
  unit Reader: process ...
  unit Writer: process ...
  unit Supervisor: process ...
begin (* the main program *)
  c := new Supervisor(0);
  resume(c);
  for i := 1 to num_readers
  do
    resume(new Reader(nr(i),c));
  od;
  for i := 1 to num_writers
  do
    resume(new Writer(nr(i),c));
  od;
  (* zakończ *)
end
```

Analiza

Rozwiązanie przedstawione przez Bolka (powyżej) zakłada współbieżny (i konfliktowy) dostęp do zasobu.
Można udowodnic, że przedstawione tu rozwiązanie jest wolne od konfliktówych dostępów do zasobu znajdującego się w gestii supervisora, można też udowodnić, że nie wystąpi zagłodzenie klientów pisarzy.
Ale trudno sobie wyobrazić równoczesne odczytywanie (przeszukiwanie) danych. Każdy proces jest (lub może być) alokowany na innym komputerze w sieci. Każdy proces jest sekwencyjny. Metody procesu są wykonywane na zasadzie protokołu alien call czyli po kolei. Oznacza to, że proces serwer może zgodzić się na wykonywanie kilku na raz instancji procedury Czytanie (która musiałaby byc metodą serwera!), ale każde wywołanie metody Czytanie przerywa wykonywanie wcześniejsze tej metody bądź instancje te wykonywane sa po kolei.

22.5.3 Rozwiązanie z równoczesnym czytaniem

Spróbujemy naszkicować inne rozwiązanie. Z rozwiązania poprzedniego zachowamy protokól zapobiegający zagłodzeniu pisarzy. Natomiast czytanie staje się treścią metody supervisora. Nowa wersja procesu Reader zawiera instrukcję (NOWA, nie występuje w Loglanie'82!)

activate <*serwer*>.<*metoda*>(args).

```
unit Reader:process(node: integer, sv:supervisor);
begin
  return;
  do
    call sv.stamp; (* zarejestruj się *)
    call sv.startread; (* żądanie dostępu do zasobu *)
    activate sv.Czytanie(...);
    (* Czytanie, proces sv uruchomi procproc Czytanie *)
    (* jako nowy proces równolegle bądź współbieżnie *)
    call sv.endread;
    (* czytanie zakończone *)
    (* Inne czynności *)
  od
end Reader;
```

Instrukcja activate jest nową proponowaną instrukcją[1]. Jej semantyka to uruchomienie nowego procesu na tym samym komputerze co proces sv. Wymagane do tego jest zezwolenie allow Czytanie ze strony procesu sv. Wątek nowego procesu to treść metody Czytanie. Po wykonaniu ostatniej instrukcji proces taki znika automatycznie. Podczas wykonywania swych instrukcji procesprocedura ma dostęp do wszystkich zmiennych procesu sv.
Proces writer pozostawiamy bez zmian.
W procesie supervisor pojawi się procedura-proces Czytanie.

[1] Właściwie można by użyc słowa **send**, przewidzianego przez Bolka w 1988.

```
unit Supervisor:process(node:integer);
  var waiting_readers:integer, (* liczba zarejestrowanych czytelników *)
    writing:boolean, (* TRUE if a writer is writing *)
  (* the following variables concern the group of readers serviced
  in a single supervisor cycle *)
  cr:integer, (* total number of readers *)
  br:integer, (* number of readers which started reading *)
  er:integer; (* number of readers which finished reading *)
  unit stamp:procedure;
  begin
    waiting_readers := waiting_readers+1;
  end stamp;
  unit startread:procedure;
  begin
    br := br+1; (* next reader started the reading *)
    writing := false; (* no writer is writing *)
    allow Czytanie;
  end startread;
  unit endread: procedure;
  begin
    er := er+1; (* a reader finishes the reading *)
  end endread;
   unit Czytanie: processprocedure(...);
  (*  TAK! wywołanie call Czytanie uruchamia proces procedury wewnątrz procesu supervisor *)
   end Czytanie;
  unit startwrite:procedure;
  begin
    writing := true; (* a writer is writing *)
  end startwrite;
  unit endwrite:procedure;
  end endwrite;
begin
  return;
  do
    br := 0; er := 0;
    accept startread, startwrite;
    if writing (* if a writer was first *)
    then (* then we wait until it finishes the writing *)
      accept endwrite; (* and finish the service cycle *)
    else
      disable stamp;
      cr := waiting_readers+1; (* the number of waiting readers*)
      waiting_readers := 0; (* additional readers will be *)
      (* serviced in the next readers group *)
      enable stamp;
      while br < cr
      do
        accept startread, endread;
      od;
      while er < cr
      do
        accept endread
      od;
    fi;
  od
end supervisor;
```

wymyśl nazwę

Nowy rodzaj modułu jest:

- procesem, może być uruchomiony na rdzeniu komputera (równolegle), lub współbieżnie z procesem obejmującym i innymi instancjami procesami takich modułów.
- jego instancja ma dostęp do procesu obejmującego
- nowa instrukcja activate zamiast instrukcji call

Zasobem, którym procesy mogą się dzielic jest ekran. Ale ... Czy takim zasobem może byc np. plik?

Czy jest sens rozważać problem czytelników i pisarzy?
TAK. Proces Zasób ma metodę czytanie. Ta metoda rozpoczyna się od enable startread. Tzn. w trakcie czytania nowy proces może zgłosić chęć czytania. Proces Zasób może zgodzić się na czytanie – przez enable czytanie? lub accept czytanie.

Obiekt *sv* procesu supervisor zarządza dostępem do dzielonego zasobu. W jakich stanach może się znaleźć ten obiekt:

początek W tym stanie zarządca oczekuje na zgłoszne startread lub startwrite **accept** startread, startwrite,

pisanie W tym stanie zarządca oczekuje na operację pisania **accept** endwrite,

czytanie W tym stanie wyrózniamy trzy kolejne stany składowe tego stanu

- zapisz liczbę zarejestrowanych czytelników,
- dopóki liczba zarejestrowanych czytelników jest większa od liczby czytelników, którzy rozpoczęli czytanie oczekuj na startread lub start end,
- dopóki liczba czytelników, którzy zakończyli czytanie jest mniejsza od liczby zarejestrowanych czytelników oczekuj na start end,

...

Komentarze

Zauważ, że można się obejść bez instrukcji enable i disable. Wymaga to wprowadzenia dodatkowego procesu, który zajmowałby się wyłącznie rejestrowaniem zgłoszeń czytelników i na żądanie udostępniałby tę informacje procesowi superwizor.

Powazniejszy problem mamy z wykorzystaniem zezwoleń na operacje czytania. Procesy czytelników i proces zarządcy zasobu nie mają wspólnych danych. Zmienne każdego procesu są prywatne i niedostępne z zewnątrz. Na czym ma więc polegać równoczesne czytanie danych z zasobu przez różne obiekty procesu czytelnik? Czy to w ogóle jest możliwe?

Notatka 1 maja 2017
Będzie to możliwe gdy serwer potrafi uruchomić kolejne "wątki" procedury czytanie na swoich rdzeniach lub współbieżnie. Po rozpoczęciu procedury czytanie

w serwerze. Bo procedura ta musi być w serwerze. Zaczynamy od instrukcji enable czytanie w treści tej procedury. Wiemy, że nie ma konfliktu z poprzednio uruchomionymi wątkami czytania i możemy uruchomić kolejny wątek. Problem polega na tym, że proces wykonywania procedury czytanie, ma się zachowywać jak by był procesem. Alokacja na węźle? nie musimy się tym przejmować niech zajmie sie tym interpreter int. Ale trzeba go nieco zmienić! Najtrudniejsze jest wskazać int-owi, że ma postąpić inaczej. A jak ma postępować? Stworzyć nowy obiekt procesu anonimowego i uaktywnić go.
W nowej wersji Loglanu mogłoby to wyglądac tak:

```
unit Czytanie: wątek procedura (...);
...
end Czytanie;
```

gdzie wątek jest procesem. Obecna wersja Loglanu nie akceptuje takiej składni, niestety. (7 ERROR 303 COROUTINE/PROCESS ILLEGAL HERE AS PREFIX WATEK)

Miałem pomysł na ciekawy przypadek zadania czytelnicy i pisarze. Co to było? Aha, czy potrafimy opisać (z sensem) pracę przebiegu kompilatora (wieloprzebiegowego) tak jak to zrobili autorzy GIERAlgolu?
Każdy przebieg pobiera dane z jednego pliku i wysyła wyniki do kolejnego pliku. Jeśli pobieranie odbywa się z bufora A, a tymczasem komputer (inny procesor) ładuje do bufora B. I podobnie wysyłane sa wyniki do Bufora C podczas gdy komputer wysyła zawartość bufora D na dysk. Narysować układ tych procesów. Trochę podobny do pierścienia producentów i buforów pomiędzy nimi.

Wydaje się wskazanym, obmyślenie innego modelu procesu.
Zauważmy, że w pierwszym programie jaki obejrzeliśmy w tym rozdziale wiele obiektów procesu pisarz wypisywało na ekran. (Co skutkowało chaosem.) Zasób może mieśćic się w pamięci dyskowej np. na pliku. Wtedy równoczesne czytanie ma sens.

The idea of the above solution is that during the reading cycle only those readers are allowed to start reading which were waiting at the beginning of the cycle. All others must wait for the next reading cycle and a writer may be serviced before them. In this way the readers cannot starve the writers. As can easily be seen the solution would be simpler and more elegant if the alien call mechanism made available to a process the number of processes waiting for an alien call of the given procedure. In that case the procedure stamp and registration of waiting readers could be omitted. Instead a system count of waiting readers should be consulted. A simple extension to the alien call is described below. A solution using only the pure synchronous rendezvous would be more complex because the supervisor should consist of two separate processes: one of them for registering waiting readers and the second for actually controlling reading and writing.

22.6 Współpraca z bazą danych

Wiele obiektów procesów może współpracować z bazą danych. Dla uproszczenia przyjmiemy, że mamy do czynienia z obiektami procesów jednego z dwu rodzajów:

klient – obiekt takiego procesu (lub podtypu typu klient) zgłasza do bazy żądania (zadania) np. czy należy?, wstaw, usuń, min, następny – a więc operacje na kolejce priorytetowej,

serwer – obiekt procesu serwer zarządza swoim poddrzewem, ma sporą pamięc przeznaczoną na ten cel (rzędu 4GB lub więcej), potrafi komunikować się z podobnymi obiektami poprzez sieć

Obiekty – serwery tworzą graf (drzewo). Obiekt s typu serwer może zlecać zadanie innemu serwerowi q, a w tym czasie przyjmować zlecenie od klienta. Obiekt serwer jest więc i serwerem i klientem. Obsługa zleceń i przyjmowanie komunikatów o wynikach zleconych zadań odbywa się zgodnie z protokołem *alien call*. Równoległość jest zapewniona przez rozproszenie zadań na procesory połączone siecią.

Jeśli pomyślimy, że baza danych jest zorganizowana jako drzewo, np. drzewo BST lub B-drzewo, to operacje dostępu do bazy danych możemy podzielić na grupy w ten sposób:

c *czytanie* informacji,

m *modyfikacja* rekordu,

z wstawianie oraz usuwanie rekordu – pociąga *zmianę* drzewa,

i *intencja* zmiany - zawsze poprzedza zmianę.

Pary operacji konfliktowych i niekonfliktowych przedstawione są w poniższej tabelce zaczerpniętej z książki [**?**].

	c	m	i	z
c	+	-	+	-
m	-	-	+	-
i	+	+	-	-
z	-	-	-	-

Problem jest podobny do problemu czytelników i pisarzy. Przyjmujemy więc następujące ustalenia:

- w każdym procesie operacje intencji oraz zmiany (wstawiania lub usuwania) tworzą parę i występują w tej kolejności,
- operacje te występujące w jednym procesie, nie moga byc rozdzielane przez operacje intencji lub zmiany pochodzące z innego procesu,
- dopuszczalne natomiast jest wykonanie operacji czytania lub modyfikacji (zgłaszane przez inny proces) pomiędzy operacjami intencji i zmiany,
- operacje czytania i intencji nie są konfliktowe i mogą być wykonywane równocześnie przez wiele procesów,

- operacje intencji i modyfikacji także nie są konfliktowe i mogą być wykonywane równoczesnie.

Prowadzi to do następującej konkluzji: obiekt procesu supervisor może się znajdować w jednym z ośmiu stanów:

P początek cyklu, accept startread, startmod, startintent,

C czytanie, accept endread, startread, startintent,

CI czytanie oraz intencja accept endread, startread, startintent,

I intencja accept startread, startintent, endintent,

MI intencja oraz modyfikacja accept startmod, endmod

M modyfikacja accept startmod, endmod, startintent

U usun accept startdel,enddel

W wstaw accept startins, endins.

Moduł procesu Klient ma następującą budowę:

```
unit Klient: process(node: integer, sv:Serwer);
    unit DoCzytania: class;
    begin
    call sv.zarejestruj;
    call sv.startread;
    inner;
    call endread;
    end DoCzytania;
    unit DoPisania: class;
    begin
    call sv.intent;
    call sv.startwrite;
    inner;
    call endwrite;
    end DoPisania;
    unit Czytanie: DoCzytania procedure(...);
    ...
    end Czytanie;
    unit Wstawianie: DoPisania procedure(...);
    ...
    end Wstawianie;
    unit Usuwanie: DoPisania procedure(...);
    ...
    end Usuwanie;
begin
  (* treść Klienta *)
end Klient;
```

Moduł Klient może być rozszerzany na kilka sposbów (przez dziedziczenie) w zalezności od potrzeb konkretnej aplikacji.
Poniżej przedstawiamy moduł zarządzający dostępem do bazy danych.

```
unit Supervisor: process(node: integer);
    var stan: char;
    unit zarejestruj procedure ...
    unit startread: procedure ...
    unit startintent: procedure();
    unit endintent: procesure();
    unit startdel: procedure
    unit enddel: procedure
    unit begin
  stan:='P';
  return;
  do
    (* Początek cyklu: *)
    case stan
    when 'P': accept startread, startintent, startmod;
    when 'C': disable zarejestruj;
         zarejestrowanych:=

    when 'I': accept endintent, startread, startmod;
    when 'J' (* 'IF' *): accept startdel, startins;
    when 'M':
    when 'D' (* CI' *):
    when 'N' (* 'MI' *):
    when 'U': accept enddel;
    when 'W': accept endins;

    esac;
  od
end Supervisor
```

uzupełń brakujące metody: startread, ...

każda metoda kończąc wyznacza nowy stan

22.6.1 Propozycja obliczeń równoległych

Jak należy wykorzystać serwer z wieloma rdzeniami?
Myślę, że można nadchodzące wywołania metod serwera realizować nie na zasadzie alien call, lecz nieco bardziej liberalnej i elastycznej.
Jeśli serwer realizuje metodę m i od innego klienta nadchodzi żądanie wywołania metody np. m lub innej m', to nieużywany rdzeń może rozpocząć wykonywanie tej metody. Utworzy rekord aktywacji, odbierze parametry aktualne, etc.
Wymaga to zmiany w interpreterze, ale jest do pomyślenia.

Jak zaprogramowac obsługę bazy danych? Z żądaniami zwracają się klienci

Wskazówki do ćwiczeń

W tym rozdziale zapisujemy sugestie dla prowadzącego ćwiczenia. Oczywiście, należy tematy zajęć dopasowac do poziomu umiejętności i wiedzy słuchaczy. Inaczej powinniśmy zaplanować zajęcia ze słuchaczami pierwszego roku, a inaczej z słuchaczami lat wyższych.
Francuska nazwa *travaux dirigées* najlepiej oddaje sens pracy podczas ćwiczeń. Studenci wykonują pracę własną wspomaganą lub kierowaną przez prowadzącego ćwiczenia.
Błędem dydaktycznym byłoby zamienienie ćwiczeń w wykład.
Każdy student powinien mieć czas na zastanowienie się i przygotowanie swojej wypowiedzi.

1. Drukowanie
   Student powinien zyskać doświadczenie w pracy z kompilatorem i maszyna wirtualną.

2. Zadanie sprawdź czy klasa GaussC definiuje pierścień.
   Na ćwiczeniach poprzedzajacych wyraźnie zadaj jako zadanie domowe: *wyliczyć własności definiujące pierścień* czyli podaj definicję pierścienia.
   Podczas ćwiczeń: każdemu studentowi przydziel zadanie cząstkowe np. sprawdź przemienność operacji *mult* zdefiniowanej w klasie GaussC. Masz na to 20 minut.
   Po 20 minutach pracy własnej (ale z możliwością rozmowy z kolegami): wywołani studenci podchodzą kolejno do tablicy i tłumaczą jak uzasadnili soje wnioski.

3. Zadanie: Mnożenie macierzy liczb zespolonych.
   Zadanie to można zrealizowac na 8 różnych sposobów:

   - zastosować zwykłe (4m+2a) lub sprytne mnożenie liczb zespolonych (3m +5a),
   - zastosować zwykły algorytm mnożenia macierzy lub algorytm Winograda,
   - przedstawic dane jako tablicę obiektów typu complex lub jako parę tablic liczb rzeczywistych.

   Studenci mają zrealizować programy, oszacować koszty i na podstawie pomiarów czasu wyliczyć współczynniki wielomianów kosztu.
   Po zakończeniu prac zespoły przedstawiają swoje wyniki (w formie komunikatów na mini-seminarium).

4. Programowanie obiektowe.
   Warto poświęcić kilka zajęć na ten temat.
5. Współprogramy.
6. Procesy.

Bibliografia

[1] Joao Abreu, Vasco T. Vasconcelos, Isabel Nunes, Antonia Lopes, Luis S. Reis, and Alexandre Caldeira. ConGu, The Specification and the Refinement Languages. http://labmol.di.fc.ul.pt/congu/, March 2007.

[2] Amir D. Aczel. Wielkie twierdzenie Fermata. Rozwiązanie zagadki starego matematycznego problemu. Prószyński i Ska, Warszawa, 1998.

[3] P Amey. Logic versus Magic in Critical Systems. In Reliable Software Technologies - Ada Europe 2001, Lecture Motes in Computer Science 2043. Springer, 2001.

[4] Krzysztof Apt, Frank S. de Boer, and Ernst-Rüdiger Olderog. Verification of Sequential and Concurrent Programs. Springer, Berlin Heidelberg, 2010.

[5] J. Barnes. High Integrity Software. Addison-Wesley, London, 2003.

[6] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt. Verification of Object-Oriented Software, the KeY approach. LNAI 4334. Springer, Berlin, Heidelberg, 2007.

[7] Per Brinch-Hansen. Podstawy Systemów Operacyjnych. PWN, Warszawa.

[8] Bolesław Ciesielski. Alien call - a synchronization mechanism for distributed processes . Master's thesis, University of Warsaw, 1989.

[9] E.W.D. Dijkstra. Umiejętność Programowania. WNT, Warszawa, 1978.

[10] A Diller. Z: An Introduction to Formal Methods. J. Wiley, Chichester, 1990.

[11] H Ehrig and G. Mahr, editors. Fundamentals of Algebraic Specification 1. Springer, 1985.

[12] Erwin Engeler. Algorithmic Properties of Structures. Math. Systems Theory, 1:183–195, 1967.

[13] Andrzej Grzegorczyk. Zarys logiki matematycznej, volume 20 of Biblioteka matematyczna. PWN, Warszawa, drugie wydanie edition, 1969.

[14] Andrzej Grzegorczyk. Undecidability without Arithmetization. Studia Logica, 79(2):163–230, 2005.

[15] C.A.R. Hoare and Heng Jifeng. Unifying Theories of Programming. Prentice Hall, 1998.

[16] Laszlo Kalmár. Über ein Problem betreffend die Definition des Begriffes der allgemeinrekursiven Funktion. Zeit. math. Logik, 1:93–96, 1955.

[17] Stephen C. Kleene. General Recursive Functions of Natural Numbers. Mathematishes Annalen, 112:727–742, 1936.

[18] Donald Knuth. The Art of Programming. 1977.

[19] Jeffrey C. Lagarias, editor. The Ultimate Challenge: The 3x+1 Problem. American Mathematical Society, Providence R.I., 2010.

[20] Bertrand Meyer. Eiffel. P, W, 1987.

[21] G. Mirkowska, A. Salwicki, M. Srebrny, and A. Tarlecki. First-Order Specifications of Programmable Data Types. SIAM Journal on Computing, 30:2084–2096, 2000.

[22] G. Mirkowska, A Salwicki, and O. Świda. SpecVer - the methodology integrating specification, programming and verification. Fundamenta Informaticae, 85:343–357, 2008.

[23] G. Mirkowska, Andrzej Salwicki, and O. Świda. Verifying a Class: combining Testing and Proving. Fundamenta Informaticae, 95:305–324, 2009.

[24] Grażyna Mirkowska and Andrzej Salwicki. Algorithmic Logic. PWN and J.Reidel, Warszawa, 1987. "[Online; accessed 7-August-2017]".

[25] Grażyna Mirkowska and Andrzej Salwicki. The Algebraic Specification do not have the Tennenbaum property. Fundamenta Informaticae, 28:141–152, 1996.

[26] Andrzej Mostowski. Axiom of choice for finite sets. Fundamenta Mathematicae, 33:137–168, 1945.

[27] W.M. Ratajczak-Bartol and D. Szczepańska-Wasersztrum. Data Structure for Simulation Purpose in Loglan77. Technical Report 373, Institute of Computer Science, Polish Academy of Sciences, Warszawa, 1979.

[28] W.M. Ratajczak-Bartol and D. Szczepańska-Wasersztrum. Code of Simulation and other Classes. http://duch.mimuw.edu.pl/∼salwicki/EOP/PQclass.html, October 2007.

[29] S.Alagić and M.a. Arbib. Projektowanie programów poprawnych i dobrze zbudowanych. WNT, 1982.

[30] Andrzej Salwicki. On Algorithmic theory of Stacks. Fundamenta Informaticae, 3:311–332, 1980.

[31] Oskar Świda. SpecVer - Specification, Verification, Programming, a plugin into Eclipse. http://aragorn.pb.bialystok.pl/∼swida/svp, April 2007.

[32] Helmuth Thiele. Wissenschatstheoretische Untersuchungen in algorithmischen Sprachen. VEB Deutscher Verlag der Wissenschaften, Berlin Heidelberg, 1966.

[33] Yuri I. Yanov. O logicheskih schemach algoritmov. Problemy Kibernetiki, 1:75–127, 1958.

Indeks