Światły Programista

czyli zastosowania

rachunku programów

do
specyfikacji i weryfikacji oprogramowania
zilustrowane licznymi przykładami programów i
twierdzeń o programach wraz z dowodami.

książkę tę napisali
Grażyna Mirkowska ♡ Andrzej Salwicki
Instytut Informatyki UKSW & Dombrova Research
Dąbrowa Leśna

19 czerwca 2022

[chapter] ———————————————————————
———————————- F

Dąbrowa Leśna, dnia 19 czerwca 2022

Czytelnikowi, a dobrodziejowi naszemu, niniejszą makietę książki ośmielamy się przedłożyć, z prośbą

Prośba.

Moja Droga, Mój Drogi.

Masz przed sobą makietę książki. Pomóż nam ukończyć pracę nad nią. Przyślij nam uwagi, pytania, słowa oburzenia, ...
Każdy komentarz jest dla nas cenny. Ale nie trać czasu na wyliczanie literówek i innych błędów, które można poprawić automatycznie.
A oto nasze przykładowe pytania do Ciebie:

- Czy stworzyć osobny rozdział z dowodami poprawności algorytmów zamiast sekcji w rozdziale L5?
- Zmienić przykłady w przedmowie? Usunąć Fermata.(!)

———————————————————

Mamy jeszcze jedną prośbę pamiętaj, o naszych prawach autorskich. Nie rozdawaj tego tekstu. For your eyes only.

ROZDZIAŁ 1

Przedmowa

Felix qui potuit rerum cognoscere causas Wergiliusz

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 (tu pozostawiamy Ci do wyboru pozostałe 81 synonimy)...

Tytuł "Światły programista" nawiązuje do manifestu i metodologii o nazwie "literate programming" zaproponowanej przez Donalda E. Knutha zob. []. Trudno znaleźć odpowiednie tłumaczenie tego zwrotu na język polski. W słowniku angielskie słowo literate jest tłumaczone na "umiejący czytać i pisać", a jako drugie znaczenie tego słowa słowniki podają "wykształcony, edukowany, uczony". Zajrzyj na stronę www.literateprogramming.com by zapoznać się z motywacją podaną przez Knutha i jego współpracowników. Jest on przekonany o potrzebie równoczesnego lub naprzemiennego rozwoju projektu programistycznego i jego dokumentacji.

My podzielamy pogląd Donalda Knutha, ale podążamy dalej – w procesie tworzenia oprogramowania nie można ograniczać się do tworzenia kodu i dokumentacji. Czym bowiem jest dokumentacja? i do czego ma służyć?

Naszym zdaniem każdemu modułowi M oprogramowania towarzyszyć powinny: specyfikacja S, czyli spis

wymagań i weryfikacja $W(M,S)$, czyli argumenty, najlepiej w postaci sprawdzalnego dowodu, na rzecz tezy, że dany moduł M spełnia wymagania wymienione w specyfikacji. Uważamy, że programista (lub zespół inżynierów oprogramowania) biorący udział w tworzeniu pewnego projektu programistycznego powinien pracować z dokumentami trzech rodzajów:

- specyfikacje tj. dokumenty w których określone są wymagania,
- kod, a więc, programy, klasy i inne moduły oprogramowania,
- weryfikacja – dokumenty z tej grupy powinny dostarczać argumentów pozwalających na podejmowanie decyzji o zaakceptowaniu lub odrzuceniu kodu. [1]
  Inaczej niż wielu autorów, uważamy, że praca w tej dziedzinie poszerza wiedzę nie tylko o jednym programie. Tworząc dowód poprawności algorytmu dokładamy cegiełkę do odpowiedniej algo-rytmicznej teorii, liczb, stosów, drzew binarnych wyważonych , ...

Gwarancja zatrudnienia dla Ciebie. 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

[1] Mianowicie, jeśli dostarczono dowody na rzecz tezy: "kod P spełnia wymagania specyfikacji S"to należy wykonawcy zapłacić i można przejść do eksploatacji oprogramowania. Jeśli dowodów zabraknie lub sa one nieprzekonywujące to należy postąpić inaczej.

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 następującemu przykładowi.

Od ponad 80 lat, tysiące matematyków i informatyków próbują udowodnić, że poniższy, prosty algorytm Col zawsze (tzn. dla każdej liczby całkowitej, dodatniej n) zakończy obliczenia, por. [Lag10]

---

powtarzaj
Col: $\{$**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[2]. Nie umiemy jak dotąd udowodnić przypuszczenia Lothara Collatza (z roku 1937): dla każdego n program Col zakończy obliczenie. W tym czasie ludzkość potrafiła wejść na Mount Everest i polecieć na księżyc. A odpowiedzi na pytanie: czy program Col zawsze zakończy obliczenie? wciąż nie znamy.

Czasami jednak się udaje. Ponad trzysta lat zajęło setkom matematyków udowodnienie, że następujący algorytm PF nie zakończy obliczeń, por.[Acz98].

---

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

```
x:=0 & y:=0 & z:=0 & n:=0 ;
dalej := true;
powtarzaj
    | jeśli n > 2 ∧ x * y * z > 0
PF:  | 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. W codziennej praktyce stosujemy setki i tysiące programów i algorytmów[3]. Czy nie niepokoi Cię bezgraniczne zaufanie jakim, zdaje się, obdarzamy programy?

Własność stopu jest podstawową własnością semantyczną programu(-ów). A są jeszcze inne, równie ważne własności semantyczne. Będziemy o nich mówić w dalszych rozdziałach.
Podsumujmy, analiza programów jest zadaniem trudnym i fakt ten stanowi gwarancję pracy dla światłych programistów.

Zadania dla światłego programisty. Czego oczekuje szef od programisty? Mówiąc najkrócej: szefowi zależy na zmniejszeniu kosztów utrzymania oprogramowania wytworzonego przez firmę. Dawno temu zauważono, że koszt napisania oprogramowania jest 5-7 razy mniejszy od kosztów utrzymania oprogramowania. Ale co to znaczy? To znaczy, że po wytworzeniu oprogramowania trzeba je ulepszać, zmieniać i dopasowywać do zmieniającego się zapotrzebowania klienta.
Czy istnieją metody i narzędzia tworzące warsztat pracy dla eksperta (audytora) sprawdzającego czy oprogramowanie spełnia warunki wyliczone w specyfikacji? dla architekta oprogramowania, który ma

[3] Czasami słyszymy o fatalnych skutkach błędu w programie.

opisać specyfikację klas i metod jakie należy napisać? Koszty poprawiania programów zmniejszają się radykalnie jeśli programom towarzyszą dowody oczekiwanych własności semantycznych. Takich jak poprawność względem specyfikacji, skończoność obliczeń, ocena kosztu obliczeń itp. Programy dzisiejsze to nie tylko algorytmy ale także klasy. Programy są dużymi produktami o bardzo złożonej strukturze. Jakie wymagania stawia się dużym programom i jak sobie z nimi radzić? Okazuje się, że 1°moduły programów takie jak procedury, funkcje, klasy, ... mają nowe rodzaje własności semantycznych, 2°te własności wyrażają się inaczej niż własności semantyczne algorytmów.
Od pięćdziesięciu lat zajmujemy się rachunkiem programów czyli logiką algorytmiczną.

Czy komputer może nam pomagać? Wiemy już, że komputery nie wyeliminują Twojej pracy, ale być może mogą ją ułatwić? Dzisiejsze komputery mają większe możliwości.
Warto się zastanowić. Komputer Deep Blue firmy IBM wygrał w szachy z mistrzem świata w r. 1997. Dziś domowe komputery można łączyć w klastry. Można tworzyć większe programy. Być może powstaną programy ułatwiające analizę semantycznych własności programów. Nie będą to niezawodne wyrocznie, ale mogą powstać całkiem przydatne narzędzia. Wystarczy zmagazynować w takim oprogramowaniu analitycznym spory fragment wiedzy znanej ludziom.
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. Jeśli jakaś klasa ma być w domenie publicznej, to naturalne jest oczekiwanie, że oprócz jej treści podane zostaną argumenty uzasadniające uznanie jej za poprawną i bezpieczną.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,

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.[4]

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 ukrywają 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 sformułować po co napisano dany program? Czy użytkownika może spotkać niespodzianka? Dlaczego testujemy programy? Czy nie ma innej

[4]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...

drogi?
Twierdzimy, że dowodzenie jest Realną alternatywą dla testowania. W tej książce znajdziesz sporo argumentów przemawiających za naszą tezą.
Którą metodę weryfikacji wybrać? Otóż nie musisz wybierać. Uważamy bowiem, że raz napisany algorytm powinien być analizowany z wykorzystaniem narzędzi rachunku programów. Inaczej mówiąc, należy formułować odpowiednie twierdzenia o semantycznych własnościach algorytmu i je dowodzić. Nie należy jednak odrzucać testowania. Bardzo przydatne okazuje się przeprowadzanie eksperymentów z nowym programem. Wyniki eksperymentów mogą dostarczyć wiele ciekawych informacji. W ćwiczeniu polegającym na porównaniu ośmiu różnych algorytmów mnożenia macierzy liczb zespolonych to eksperymenty pozwolą nam obliczyć współczynniki w wielomianowych funkcjach kosztu, por ??. Czasem zaś eksperymenty mogą nas zainspirować i otworzyć oczy na nowe problemy. Karl Gauss najpierw eksperymentował tj. wykonywał różne obliczenia, by potem formułować twierdzenia i je dowodzić.

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.
Program ma przynajmniej jednego czytelnika – to autor programu. jakże częsta jednak jest sytuacja, w której trzeba zrozumieć program napisany przez innego człowieka.

Klasy i procedury z bibliotek klas są stosowane wielokrotnie. Dziś nie towarzyszy im żaden dowód poprawności, żadna analiza. Naszym zdaniem byłoby znacznie lepiej, gdyby klasa (bądź procedura lub funkcja) udostępniana do korzystania przez innych była analizowane i gdyby można było obejrzeć i zweryfikować dowód pewnej własności takiej klasy. Nie jest korzystne dla społeczności i przemysłu informatycznego patentowanie oprogramowania. A wiele krajów popiera patentowanie oprogramowania. Rozwój powinien odbywać się na zasadzie przyjętej w naukach matematycznych.
W pełni zgadzamy się z tezami Richarda Stallmana [Sta]. Społeczność powinna mieć dostęp do kodów źródłowych[5]. Od siebie dodajemy kolejny postulat: publikacji oprogramowania powinna towarzyszyć publikacja przedstawiająca argumenty na rzecz tezy, że oprogramowanie to posiada zakładane cechy. Społeczność informatyczna byłaby w stanie weryfikować takie dowody i ewentualnie je korygować. Tak jak to czyni społeczność matematyczna. W naukach dedukcyjnych badacz posługuje się twierdzeniami opublikowanymi przez innych autorów.
Miliardy użytkowników posługują się programami, o których nic nie wiadomo. Kto zna ich treść? Kto zna argumenty, które uzasadniałyby poprawność(lub inną cechę) wykorzystywanego programu? Czy zdajesz sobie sprawę, że tzw. gwarancja jakości produktu programistycznego, to sztuczka polegająca na tym, że to Ty płacisz za ubezpieczenie producenta oprogramowania. Producent oprogramowania ubezpiecza swoją firmę od odpowiedzialności za ewentualne szkody poniesione przez użytkownika(-ków). Koszt tego ubezpieczenia jest wliczony w cenę produktu.
Jesteśmy przekonani, że firmy informatyczne, które zaczną stosować metody dedukcyjne wygrają, ponieważ ich produkty będą tańsze.

---

[5]Na pewno odnosi się to do upublicznianych klas i innego oprogramowania. Jeśli oprogramowanie ma być wykorzystywane przez jedną tylko firmę, to nie oczekujmy publikacji kodu.

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[AA82], Apt, de Boer i Olderog[AdBO10], Hoare[HJ98], Dijkstra[Dij78]).

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 wyjątku, ...

Znajdziesz tu wiele przykładów programów. Wszystkie są napisane w języku programowania obiektowego i rozproszonego Loglan. O projekcie Loglan piszemy nieco dalej. W tym miejscu pragniemy się wytłumaczyć z dokonanego wyboru: otóz jest to jedyny znany nam język, w którym znajdują się potrzebne nam narzędzia programowania. Oczywiście możesz tworzyć swoje przykłady w Twoim ulubionym języku programowania, a raczej w kilku językach.W pewnym języku nie znajdziesz współprogramów, w innym modułów proces, etc.

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.

Luka. Od wielu lat (60+) prowadzimy wykłady i badamy rozmaite problemy wiążące się z semantyką programów. Coraz wyraźniej dostrzegamy dużą lukę w programach studiów informatycznych, a także w ofercie podręczników publikowanych z myślą o inżynierach oprogramowania i słuchaczach studiów informatycznych. ...

Mamy nadzieję, że książka jaką prezentujemy pomoże tym, którzy dostrzegaja lukę w kształceniu informatyków. Zgodzisz się, że w obecnym curriculum, przyszły informatyk uczy się pisania programów, potem poznaje algorytmikę tzn. istotne algorytmy i wazne struktury danych. Podczas kursu algorytmiki słuchacze i wykładowcy posługuja się pseudokodem i intuicyjnymi definicjami struktur danych. W tej książce ostarczamy narzędzi motywujących programistę do analizowania specyfikacj i dowodów, informatyka teoretyka do pracy z kodem. Na przykład, wielu autorów posługuje się pojęciem dag'ów, nikt nie korzysta z algorytmicnych aksjomatów struktur danych. Na dalszych stronach znajdziesz kompletne definicje stosów, kolejek, kopców, drzew BST i in. Z naszej książki dowiesz się na czy polegaja protokoły: call – wywołania procedury lub funkcji, alien – obcego wywołania metody jednego (obiektu) procesu z innego obiektu procesu, raise – protokół obsługi sytuacji wyjatkowej, i in. Nikt go nie

Jak korzystać z tej książki? To jest dość gruba książka i może być wykorzystana do różnych wykładów obecnie obowiązującego 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), programowanie 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 pamięcią obiektów (tzw. sterta), algorytm wyznaczania klasy dziedziczonej (rozszerzanej) w Javie i ...,
- algorytmy i struktury danych –

  - aksjomatyczne definicje struktur danych np. stosy zob. rozdział 16, kolejki FIFO, kontenery (dictionaries), kolejki priorytetowe, drzewa BST zob. rozdział 21, etc.,
  - dowody semantycznych własności algorytmów por.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.

0.0.1. Do informatyków pracujących zawodowo. O ile nam wiadomo, żadna uczelnia nie ma w programie treści przedstawionych w tej książce. Nie znamy też innej książki zawierającej podobne przesłanie.
Nawet pobieżne przeczytanie naszej książki może zachęcić Cię do przemyślenia Twego światopoglądu informatycznego. wylicz kilka pytań 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 określić fundamentalną różnicę pomiędzy jednym a drugim paradygmatem programowania? dlaczego rekomenduję paradygmat P?
Trochę poważniejsze problemy pojawiają się gdy firma zamierza zakupić oprogramowanie reklamujące się, że przy pomocy metod sztucznej inteligencji oprogramowanie to będzie w stanie wykryć błędy zapętlania się 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. Mówiąc obrazowo, omawiamy zastosowania rachunku programów ilustrując je programmami napisanymi w Loglanie. Ale to niecała prawda.

Przedstawiamy też liczne problemy jakie pojawiają się podczas projektowania języka programowania. Część z tych problemów została rozwiązana przez ekipę Loglanu. Oba projekty badawcze nadal są żywe i otwarte: wciąż napotykamy nowe problemy i (od czasu do czasu) uzyskujemy nowe wyniki.

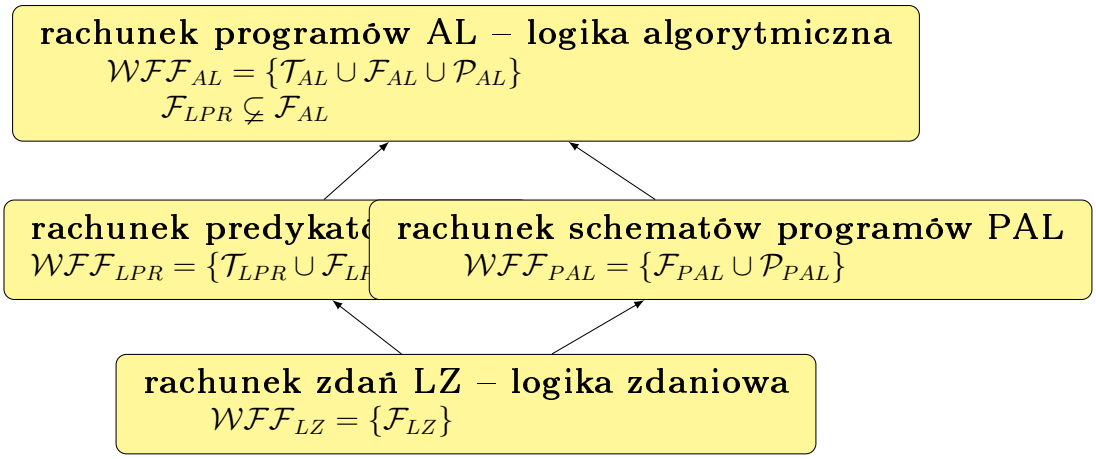
Rachunek programów. Celem projektu badawczego rachunek programów jest dostarczenie narzędzi przydatnych inżynierom oprogramowania w ich pracy nad projektowaniem oprogramowania (specyfikacja czyli sformułowanie wymagań) i później, podczas pracy polegającej na sprawdzeniu czy stworzone oprogramowanie jest 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, wprowadzenie kwantyfikatorów iteracji), G. Mirkowska (twierdzenie o pełności logiki algorytmicznej), prof. A. Kreczmar (1945–1996) (umieszczenie logiki algorytmicznej w hierarchii Kleene-Mostowskiego, badania programowalności w ciałach), prof. L. Banachowski (zastosowanie logiki algorytmicznej do badania częściowej poprawności programów, algorytmiczna aksjomatyzacja drzew binarnych), prof. H. Rasiowa (wielowartościowa logika algorytmiczna).

Podziękowania kierujemy także do dr Andrzeja Bieli (Uniwersytet Śląski) i profesora Wiktora Dańko (Politechnika Białostocka). Prof. Hanna Oktaba (Universidad Mexico), dr Wiesława M. Bartol, prof. Andrzej Szałas (Uniwersytet Warszawski) i prof. Uwe Petermann (HTWK Leipzig) w swoich rozprawach

doktorskich rozwinęli algorytmiczne teorie ciekawych struktur danych [?]bib i wnieśli znaczący wkład w projekt badawczy Loglan, o czym poniżej.
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[1958] (Uniwersytet w Kazaniu) – stworzył rachunek schematów programów, [Yan58]
- Erwin Engeler[1967](ETH Zürich) – wskazał na fakt, że własność stopu programu jest wyrażalna w języku $\mathcal{L}_{\omega_1\omega}$ logiki z nieskończonymi alternatywami, [Eng67]
- Helmut Thiele[1966](Humboldt Universität Berlin) – twórca systemu łączącego rachunek λ z logiką pierwszego rzędu, [Thi66]



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'. Znak $\mathcal{F}$, z odpowiednim indeksem, oznacza zbiór formuł(wyrażeń Boolowskich), znak $\mathcal{T}$ oznacza

zbiór wyrażeń np. wyrażeń arytmetycznych, znak $\mathcal{P}$ oznacza zbiór programów, lub schematów programów.

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 jeśli: po wykonaniu programu K, spełniona jest własność wyrażona przez formułę α.

Projekt Loglan. Celem projektu Loglan było znalezienie odpowiedzi na pytanie: czy można skonstruować język programowania obiektowego, który umożliwiałby także programowanie współbieżne i byłby wolny od rozmaitych problemów jakie kryły się w języku Simula67 [Salb]. 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, i in..

W pracach nad językiem Loglan brali udział: prof. Antoni Kreczmar, dr Wiesława Bartol, prof. Hanna Oktaba, prof. Tomasz Müldner(Acadia University), dr Marek Lao, Andrzej Salwicki i in.
Kompilator Loglanu powstał dzięki usilnej pracy zespołu kierowanego przez Antoniego Kreczmara: dr Danuty Szczepańskiej-Wasersztrum, dr Andrzeja Litwiniuka, Wojtka Nykowskiego, Marka Lao, prof. Pawła Gburzyńskiego (University of Alberta).

Dwukrotnie wymieniliśmy Antka Kreczmara (1945 – 1996). Bez niego nie osiągnęlibyśmy niczego. Jest on postacią niedocenioną a bardzo ważną dla polskiej informatyki. Wciąż nie możemy przeboleć jego przedwczesnej śmierci.
Pragniemy podkreślić rolę jaką w projekcie Loglan odegrał prof. Andrzej Janicki. Od początku uwierzył, że potrafimy osiągnąć postawione cele. Doprowadził do zawarcia umowy pomiędzy Zjednoczeniem "MERA", producentem komputerów Mera 400 i Instytutem Informatyki UW, czyli udzielił nam grantu na badania (1978-1982).

Wkład doktorantów. Wiele problemów zainspirowanych przez Loglan znalazło rozwiązanie w rozprawach doktorskich:
Danuta Szczepańska - system zgłaszania wyjątków i sygnałów oraz ich obsługi (wdrożenie w 1982, doktorat 1990),
Paweł Gburzyński - System automatycznego dowodzenia twierdzeń rachunku predykatów (zrealizowany w Loglanie 1982),
Hanna Oktaba - algorytmiczna teoria referencji oraz teoria systemu bezpecznego zarządzania obiektami (1982),
Andrzej Szałas - zbadanie na ile Loglan może być stosowany jako język tworzenia systemów czasu rzeczywistego,propozycje konstrukcji językowych rozszerzających Loglan i zbadanych eksperymentalnie. Podane zostały też reguły wnioskowania dla tych konstrukcji (1988).
Wiesława M. Bartol - zbadała własności systemu współprogramów (1984),
Uwe Petermann - logika algorytmiczna z operacjami częściowymi, koncepcja komunikacji procesów przez przerwania (1988),
Oskar Świda - wieloprocesorowy, rozproszony, wirtualny procesor loglanowski VLP i środowisko (1996),[Swi02]
Marek Warpechowski - algorytmy wyznaczania bezpośredniej superklasy w Javie i językach podobnych,
Andrzej Zadrożny - włożył wiele pracy w stworzenie środowiska dla repozytorium lem12.uksw.edu.pl

i repozytorium źródeł kompilatora Loglan'82 na stronach sourceforge.net.
Ta lista nie jest pełna.

Wkład studentów. 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, poźniej Linuxowych.
Wojtek Nykowski napisał parser do kompilatora Loglanu (1981).
Teresa Przytycka (dziś senior researcher in NIH) dodała do Loglanu debugger (1984).
Paweł Susicki przeniósł kompilator Loglanu do systemu Unix (1991.
Sebastien Bernard (Universite de Pau 1992) przeniósł Loglan na komputery Atari,
Frederic Pataud (Universite de Pau 1995) przeniósł Loglan do środowiska Windows95 na maszyny 32 bitowe, 1995.
Kamil Burzyński (2014) przeniósł środowisko VLP na platformę Windows XP i 7.
Należy wspomniec o A. Adamskim, P. Filipkowskim, A. Chwedoruku, J. Larrieu, R. Becourt i wielu innych.. Nie wymieniliśmy tu wszystkich, którzy przyczynili się do rozwoju Loglanu.

Współpraca międzynarodowa. Bardzo wiele zawdzięczamy profesorowi Hansowi Langmaackowi (Institut für Informatik, Universität zu Kiel): zauważył on i pomógł poprawić nieprawidłowość w semantyce wielkości nielokalnych, pomógł w instalacji Loglanu na m.c. Siemens (odpowiednik mainframe'a IBM),
dr Gianna Cioni (IASI CNR Roma) - kierowała grupą przenoszącą kompilator Loglanu na komputery VAX/VMS.
prof. Giorgio Ausiello (Universita "La Sapienza" Roma) - inspirator współpracy włosko-polskiej,

Pragniemy też podziękować kolegom w innych uczelniach: Tübingen, Bordeaux, Caen.

Książka, którą Ci oddajemy, stanowi wkład w kolejny projekt badawczy SpecVer, zobacz więcej na ten temat w repozytorium http://lem12.uksw.edu.pl/wiki/SpecVer.

Ćwiczenia.

1.1. Napisz odpowiedni algorytm realizujący polecenie: weź następną czwórkę liczb naturalnych.
Wskazówka. Każda para $\langle i, j]rangle$ liczb całkowitych otrzymuje swój unikalny numer w nieskończonej tablicy

1	3	5	7	9	11	13	15	17	19	21	...
2	6	10	14	18	22	26	30	34	38	42	...
4	12	20	28	36	44	52	60	68	76	84	...
8	24	40	56	...							...
16	48	80	112	...							...
32	96	160	...								...
64	192	320	...								...
128	...										...

Test.

Przeczytaj poniższy program i odpowiedz na dwa pytania.
1° Czy program zakończy obliczenie? Czy potrafisz opisać co jest wynikiem tego programu? Wstrzymaj się z wykonaniem programu tak długo jak potrafisz!
2° Czy potrafisz podać argumenty (przekonać), że trafnie odgadłeś?

```
program PawelG:
    var A: arrayof integer;
    var n,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
    readlnn;
    array A dim(1:n);
    for i := 1 to n do A(i) := 0 od;
    k :=1;
    call F;
    writeln("Czytaj dalej")
end PawelG;
```

Wsparcie tej książki na stronie WWW. Na stronie http://...

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

Jak zainstalować środowisko do pracy z Loglanem? Jeśli chcesz eksperymentowć i tworzyć własne programy w języku Loglan to powinieneś wykonać niezbyt skomplikowaną instalcję.[6]

Linux. Wersja dla systemu operacyjnego Linux z r. 2017, jest dostępna ze strony

https://sourceforge.net/projects/loglan82/

Wybierz Download i pingwina.

Windows. Istniejące wersje Loglanu dla systemu operacyjnego Windows 7 i XP nie zadowalają nas. Ale działają.

Nie mamy oferty dla systemu Windows 10.

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.

[6] W minionych trzydziestu latach pięciokrotnie ubiegaliśmy się o grant. Bez skutku. Z pomocą kolegów i studentów utrzymywaliśmy kompilator i przenosiliśmy na nowsze platformy, mniej więcej do r. 2017. Teraz juz nie dajemy rady aktualizować system do wciąż zmieniających się platform Linux i Windows.

Spis treści

Rozdział 1. Przedmowa iii

Rozdział 2. Wprowadzenie 1

Część 1. Rachunek programów 15

Rozdział 3. $\mathcal{L}_1$ Wyrażenia 17
1. Przykłady programów 22
2. Typy pierwotne 22
3. Składnia wyrażeń 25
4. Semantyka wyrażeń 31
5. Komputer $\mathcal{K}_1$ 38
6. Analiza 40
7. Aksjomaty typów pierwotnych 40
Ćwiczenia 50

Rozdział 4. $\mathcal{L}_2$ Programy liniowe 51
1. Przykład programu i jego obliczenia 51
2. Składnia 54
3. Semantyka 55
4. Komputer $\mathcal{K}_2$ i pojęcie obliczenia 56
5. Prawa rachunku programów 58
6. Kilka przykładów 61
7. Półgrupa podstawień i sieci 70
8. Co warto zapamiętać? 76
Ćwiczenia 77

Rozdział 5. $\mathcal{L}_3$ Programy elementarne 81
1. Składnia 81
2. Semantyka 83
3. Abstrakcyjna wirtualna maszyna $\mathcal{K}_3$ 84

4. Aksjomaty 84
5. Analiza programów 85
6. Kwantyfikator ograniczony 91
7. Funkcje pierwotnie rekurencyjne 92
8. Podsumowanie 94

Rozdział 6. Programowanie z tablicami 99
1. Tablice 100
2. Składnia języka $\mathcal{L}_4$ 103
3. Komputer $\mathcal{K}_4$ 105
4. Semantyka 109
5. Tablice dwuwskaźnikowe. 110
6. Rozwiązywanie układu równań 112
7. Mnożenie macierzy 114
8. Namawiamy do notacji matematycznej 125
9. Obiekty tablicowe – Podsumowanie 126
Ćwiczenia 127

Rozdział 7. $\mathcal{L}_5$ Programowanie z while 129
1. Programy iteracyjne 129
2. Komputer $\mathcal{K}_5$ 131
3. Aksjomat instrukcji while i reguła wnioskowania 131
4. Programowanie z instrukcją while 132
5. Własności instrukcji while 134
6. Algorytmiczne teorie typów pierwotnych 136
7. Przykłady 137

Rozdział 8. Rachunek programów 151
1. Rachunek programów 151
2. Kilka przykładów 156
3. Pełność AL 156
4. AL definiuje semantykę programów while 157
5. Definicje 162
Ćwiczenia 166

Rozdział 9. $\mathcal{L}_6$ Bloki 167
1. Składnia 167
2. Komputer $\mathcal{K}_6$ 169
3. Semantyka 174
4. Teoria $\mathcal{T}_6$ 177

Rozdział 10. $\mathcal{L}_8$ Funkcje 179

1. przykłady 181
2. Nieistotność definicji 182
3. Funkcjonały 185
Ćwiczenia 185

Rozdział 11. $\mathcal{L}_7$ Procedury 187
1. Przykłady procedur 188
2. Składnia 193
3. Komputer $\mathcal{K}_7$ 194
4. Semantyka 195
5. Dowód programu PawelG 204

Rozdział 12. $\mathcal{L}_e$ Sygnały i ich obsługa 217
1. Składnia 222
2. Semantyka 224
3. Przykłady 230
4. Porady i wnioski 230

Część 2. Programuj z klasą
lub
programowanie obiektowe 233

Rozdział 13. Klasy i obiekty $\mathcal{L}_7$ 235
1. Klasa 235
2. Obiekty 237
3. Scenariusz obiektu 241
4. Klasy mogą być zagnieżdżane 244
5. Uwagi o odśmiecaniu i defragmentacji 244
Klasa GaussC – liczb zespolonych całkowitych 245
6. Mnożenie macierzy liczb zespolonych 246
7. Klasa geometria Cyrkla i Linijki 247
8. Kolejki 249
9. Gdy trzeba zadeklarować funkcję f jako wynik innej funkcji h 254
10. Gsort 254
11. Komputer $\mathcal{K}_7$ 255
Ćwiczenia 256

Rozdział 14. Dziedziczenie albo rozszerzanie klas $\mathcal{L}_8$. 257
1. Reguła konkatenacji 257
2. Rozszerzanie innych modułów 267
Obietnice 267

3. Protokoły call i new 268
4. Statyczna struktura programu 269
5. Dynamiczna struktura 270
Ćwiczenia 271

Część 3. Teorie algorytmiczne niektórych struktur danych 273

Rozdział 15. Wprowadzenie 275
Motywacje 275
Dlaczego taki tytuł? 275
Co tu znajdę? 276
Struktury nieskończone definiowane przez klasy 276

Rozdział 16. Stosy 279
1. Specyfikacja stosów 279
2. Implementacje stosów 287
3. Abstrakcyjna klasa stosy z konkretną klasą MElem 291
4. Przykład dowodu programu na stosię 292
Ćwiczenia 293

Rozdział 17. Kolejki FIFO 295
Ćwiczenia 298

Rozdział 18. Zbiory skończone czyli kontenery 299
1. Specyfikacja kontenerów 299
2. Definicja instrucji forall 302
Ćwiczenia 304

Rozdział 19. Zarządzanie stertą obiektów 305
1. Specyfikacja 308
2. Model AK 313
3. Porównania i zastosowania 315
Ćwiczenia 317

Rozdział 20. Kolejki priorytetowe 319
1. Pojęcie kolejki priorytetowej 319
2. Niesprzeczność, tw. o reprezentacji, modele 320
3. Zastosowania 321
Ćwiczenia 321

Rozdział 21. Drzewa BST 323
1. Struktura drzew BST? 323
2. Modele. Tw. o niesprzeczności 326
3. Rozszerzenie struktury BST 328
4. Implementacja kolejek priorytetowych 333

Rozdział 22. Kopce 339
1. Definicja 339
2. Heapsort - sortowanie w kopcu 341
3. Czy to jest kolejka priorytetowa? 347

Rozdział 23. Zbiory nieskończone 369

Część 4. Programowanie z agentami 371

Rozdział 24. $\mathcal{L}_9$ Współprogramy 373
1. Moduły i obiekty współprogramów 373
2. Producent i konsument 383
3. Instrukcja detach 387
4. Licznik - dynamiczna tablica obiektów współprogramu. 389
5. Scalanie drzew BST – łańcuch dynamiczny 390
6. attach zastępuje call 392
7. Wieże Hanoi 396
8. Treegen 401
9. Przeszukiwanie z nawrotami 411
10. Symulacja. 418
11. Poprawność klasy Simulation 418
12. Specification of Simulation class 420

Rozdział 25. $\mathcal{L}_{10}$ Procesy 431
1. Programowanie współbieżne i rozproszone 431
2. Składnia i semantyka 436
3. Program Rozmowa 441
4. Producenci i konsumenci 446
5. Czytelnicy i pisarze 453
6. Współpraca z bazą danych 462
7. Rozmaite modele obliczeń 466
Ćwiczenia 471

Dodatek – Błędy czasu wykonania 473

Bibliografia 473

Dodatek. Bibliografia 473

Index 477
Dodatek. Indeks 477

ROZDZIAŁ 2

Wprowadzenie

Cele. Rachunek programów $\mathcal{AL}$ jest rozwijany od pięćdziesięciu lat z okładem. Naszym zamierzeniem było stworzenie narzędzi użytecznych w pracach nad:

1° tworzeniem specyfikacji tj. wymagań,
2° budowaniem oprogramowania implementującego te wymagania, oraz
3° sprawdzaniem poprawności oprogramowania z jego specyfikacją.

Do oznaczenia rachunku programów żywamy też nazwy logika algorytmiczna. Oba terminy: rachunek programów i logika algorytmiczna będziemy stosować zamiennie.
Pokażemy, że:

- rachunek programów jest właściwym narzędziem do wyrażania semantycznych własności algorytmów i struktur danych (w skrócie, tworzenia specyfikacji oprogramowania).
- rachunek programów jest odpowiednim narzędziem służącym do przeprowadzania analizy zgodności oprogramowania z zadaną specyfikacją.

W tej książce przedstawimy wiele przykładów takich analiz. Czasami spotykany jest termin weryfikacja oprogramowania.
O weryfikowaniu programów mówi się, że jest rzeczą pożądaną, ale nierealną, zbyt trudną. Przekonamy Cię, że pogląd ten nie ma uzasadnienia. Może trafniej jest uznać to za przesąd.

- Pokazujemy jak specyfikacja może być wyrażona w postaci zbioru formuł rachunku programów. Odnosi się to do algorytmów, a także do struktur danych.

- Przedstawimy przykłady dowodów poprawności oprogramowania względem zadanej specyfikacji.
- Przedstawimy też aksjomatyczną definicję języka programów iteracyjnch, deterministycznych.
.

A dokładniej, będziemy omawiać ciąg coraz bogatszych języków $\mathcal{L}_i$ programowania obiektowego i rozproszonego. Za każdym razem będziemy przedstawiać aksjomaty nowych konstrukcjii programistycznych jakie pojawiałą się w języku $\mathcal{L}_i$.
Krótko mówiąc chcemy Cię przekonać do tezy, że rachunek programów to narzędzie, jakiego potrzebujesz.
Nie możemy obiecać, że stosując rachunek programów rozwiążesz każdy problem pojawiający się w projekcie nad jakim pracujesz. Powinieneś jednak uzyskać lepszy wgląd w naturę tych problemów i pewną biegłość w stosowaniu narzędzi rachunku programów.
W pewnym momencie dojdziesz do przekonania, że pracując nad Twoim projektem poruszasz się w obrębie wielu różnych teorii algorytmicznych. Jesttak, ponieważ w każdym module mogą się pojawić nowe definicje czyli deklaracje funkcji, predykatów (tj. funkcji boolowskich), instrukcji atomowych tj nowych procedur. Do tego dochodzą jeszcze moduły klas, współprogramów i procesów oraz relacja dziedziczenia. Podsumowując, drogi czytelniku jesteś w sytuacji pana Jourdain, który zaskoczony został informacja, że mówi prozą. Aby należycie ocenić Twą pracę Twój szef i/lub Twój klient muszą zdać sobie sprawę z ogromu Twoich kwalifikacji.
Jest tak dlatego, że każdy program zawierający deklaracje funkcji (lub procedury lub klasy) powinien być analizowany w swojej własnej algorytmicznej teorii. Piszemy o tym obszerniej w podrozdziale ?? o definiowaniu funkcji.

Czym jest program? Zgodzimy się zapewne, że jest to napis. Ale na tym sprawa się nie kończy. Program jest opisem pewnej funkcji przekształcającej dane w wyniki. Pewne cechy programu kojarzymy z matematyką, np. w programie występują wyrażenia arytmetyczne i boolowskie. Ale uwaga inżyniera oprogramowania koncentruje się na instrukcjach. Pewne instrukcje są niepodzielne tj. atomowe. Przykładami takich instrukcji są instrukcja przypisania (zmiennej wartości wyrażenia), instrukcja drukowania, instrukcja czytania, i in. Programista ma do dyspozycji kilka operatorów programotwórczych: złożenie, instrukcja warunkowa, instrukcja powtarzania for, instrukcja iteracji while. Każdemu programowi P przypisujemy jego znaczenie dędące funkcją ze zbioru W wartościowań zmiennych w ten sam zbiór W. Nakpierw jednak trzeba okreslić czym jest obliczenie programu.
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 możemy "wziąć do ręki". Oznacza to tyle, że nie możemy działać na nieskończonych ciągach rozwinięć liczb niewymiernych, nie możemy operować 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.

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? I co to właściwie znaczy? Otóż program ma pewne własności syntaktyczne i bardziej nas interesujące, własności semantyczne. Własnością syntaktyczną pewnego programu P może być: wszystkie identyfikatory w programie są w nim zadeklarowane, lub występuje identyfikator id i nie ma deklaracji tego

identyfikatora, etc. Własności syntaktyczne odnoszą się do tekstu programu. Można o nich powiedzieć, że są własnościami statycznymi programu.

A własności semantyczne? Wymieńmy kilka podstawowych własności semantycznych: zatrzymywanie się programu czyli skończoność obliczeń, poprawność, równoważność programów, etc. Własności semantyczne odnoszą się do obliczeń programu. Są to własności dynamiczne, występują podczas dynamicznie zmieniającej się konfiguracji: $\langle$ lista instrukcji pozostających do wykonania, stan pamięci komputera $\rangle$.

Pojęcie algorytmu. Człowiek posługuje się algorytmami od tysiącleci. Zdziwienie czytelnika może wywołać fakt, że w starożytnym Babilonie znano i stosowano skomplikowane algorytmy przedstawiana liczb ułamkowych w postaci sumy ułamków o liczniku 1, np. zamiast ułamka $\frac{3}{5}$ stosowano wyrażenie $\frac{1}{2} + \frac{1}{10}$. Odkryto i codziennie używano (!) algorytmy niezbędne do dodawania, mnożenia, etc. ułamków.
Ale potrzeba podania precyzyjnej definicji pojęcia algorytmu zaistniała dopiero w XX wieku, gdy trzeba było rozstrzygnąć problem istnienia lub nieistnienia algorytmu rozstrzygającego o własności takiej jak: "formuła α jest twierdzeniem arytmetyki“ itp. W związku z tym stworzono wiele różnych definicji algorytmu [].
Z drugiej strony, gdy w latach czterdziestych dwudziestego wieku skonstruowano programowane maszyny liczące (komputery) konieczne stało się określenie jakie programy komputer potrafi wykonywać. I otwarła się puszka Pandory, niemal każdy chce stworzyć swój język programowania.
Powstało tysiące języków programowania.
Zauważmy, ze niewiele języków programowania stworzono z myślą o tym, by zapisywać w nich programy czytelne dla człowieka. W większości przypadków autorzy języka koncentrują się na tym by można było

zbudować kompilator. [1]
Proponujemy, by łatwość komunikacji pomiędzy człowiekiem – autorem programu i drugim człowiekiem – czytelnikiem programu stała się głównym celem przemysłu softwarowego.
Teza Churcha-Turinga powiada, że pomimo różnic wszystkie znane definicje algorytmu opisują tę sama klasę funkcji obliczalnych. popraw Mając na względzie bogactwo języków programowania i wielość koncepcji obliczalności zaniechamy prób podania definicji pojęcia algorytmu.
Natomiast warto przypomnieć istotne cechy algorytmu. W pracy Kołmogorowa i Uspienskiego [KU58] zawarto w miarę precyzyjne wyliczenie cech pojęcia algorytmu, na tyle jednak szerokie by obejmowało wiele rozmaitych koncepcji jakie pojawiały się od początku XX wieku.

a) Algorytm - wyznacza ciąg stanów (tj. algorytm działa dyskretnie)
b) Powtarzając obliczenie algorytmu z tymi samymi danymi początkowymi otzymamy te same wyniki.(algorytm działa deterministycznie)
c) (kroki algorytmu są działaniami elementarnymi)
d) Nie jest algorytmem przepis, który nie gwarantuje osiągnięcia wyniku.(wynik algorytmu jest określony). W teorii przepływów w sieciach spotykamy metodę Forda-Fulkersona, która może zapocząkowac obliczenie nieskończone i algorytm Edmondsa-Karpa, który zawsze zwraca wynika. W tym przypadku różnica w tekście-algorytmu jest niewielka.
e) Algorytm opisuje odwzorowanie ze zbioru danych w zbiór wyników (powtarzalność, uniwersalność zastosowań)

[1] Spróbuj przeczytać program napisany w języku Postscript lub Forth.

Nieco historii. Bardzo istotne dla historii informatyki było stwierdzenie, że potrzebujemy wyrażeń wyrażających własności semanyczne programów. Najwcześniej pojawiły się prace analizujące równoważność programów. Należy tu wymienić J. Yanova[] i Sh. Igarashi. Niestety, nie udało się w pełni scharakteryzować równoważności programów, ani równoważność (programów) nie determinuje semantyki programów. 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[].
Twórcy języka Algol 60 zamierzali do opisu składni dodać opis semantyki poszczególnych konstrukcji językowych. Zamiar ten jednak nie powiódł się.
Od początków XIX wieku prowadzono prace zmierzające do uporządkowania podstaw matematyki. Przypomnijmy, że w rachunku różniczkowym mówiono o wielkościach nieskończenie małych, ale jednak różnych od zera, itp. Gottlob Frege podał Zdania oznajmujące i rola Fregego oraz Boole'a.
W latach 30 XX wieku Alfred Tarski [] podał definicję pojęcia prawdziwości formuły, odróżnił wyrażenie zdaniowe (formuła) od wyrażenia nazwowego (term).
Definicja prawdziwości, rozumienie termu jako nazwy funkcji w odpowiedniej strukturze algebraicznej, formuły jako funkcji ze zboru wartościowań zmiennych w zbiór wartości logicznych uderza programistę: "przecież to mowa o programowaniu". Niestety, Tarski nie rozwinął swojej koncepcji na algorytmy. Dzisiaj zgadzamy się z tym, że jego wpływ na semantykę wyrażeń (w językach programowania) był decydujący, ale nie podjął on wyzwania by opisać semantykę algorytmów. Badacze pracujący w podstawach matematyki zajmowali się pojęciem obliczalności. Charakteryzowali zbiór funkcji obliczalnych. Nie zajmowali się językiem w którym funkcje obliczalne możnaby zapisać.

Młody adept "elektronicznej techniki obliczeniowej- - taki termin funkcjonował zanim wymyślono słowo informatyka – mógł podziwiać jak pojęcia spełniania i prawdziwości odpowiadały temu co napotykał w swej pracy z "maszyna matematyczną"i programami dla niej. Równocześnie odczuwał pewnien niedosyt ucząc się o funkcjach obliczalnych(tj. rekurencyjnych) i porównując to pojęcie z praktyką obliczeń na komputerze (ta nazwa powstała dużo później).

Alfred Tarski i Kurt Goedel znali się i spotykali. Można przypuszczać, że gdyby połączyli swe wysiłki, to jeszcze przed II wojną światową, mogłaby powstać teoria programów komputerowych. Cóż, nie istniały wtedy komputery.

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.

Różne definicje algorytmu, teza Churcha i brak formuł zdaniowych wyrażających własności algorytmów.

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

Garść pytań. Zbieramy tu pytania z jakimi informatycy powinni być obeznani. Każdy powinien samodzielnie wyrobić sobie pogląd jak odpowiedzieć na te pytania. W ten sposób chcemy zachęcić do wyrobienia sobie światopoglądu informatycznego.

- Czy kompilator przekształci instrukcję przypisania do najlepszej możliwej postaci?
- Czy powstaną kompilatory sygnalizujące błąd zapętlenia się obliczeń programu?
- Instrukcja for (przy spełnieniu kilku naturalnych ograniczeń) ma obliczenie skończone. Instrukcja while może mieć obliczenia nieskończone. Czy można się pozbyć tej instrukcji z języka?
- Czym są deklaracje funkcji i procedur występujące w programie?
- Zbiór funkcji obliczalnych = Zbiór funkcji programowalnych.
  Co z tego wynika?
- Czy twórcy języka Java słusznie postąpili zabraniając instrukcji dealokacji obiektu? Czy decyzja przyjęta przez autora języka C++ by pozwolić na używanie instrukcji delete jest lepsza?
- Dynamiczna czy statyczna semantyka wielkości nielokalnych?
- Czy protokół alien call można zaimplementować w Javie?
- Czy komputery wielordzeniowe i protokół alien call pasują do siebie?

Cele tej książki.

- 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 rozwiązać jakieś zadanie,
- nauczmy się argumentowania, że nasz program dobrze rozwiązuje postawione zadanie, powinniśmy posiadać umiejętność oceny kosztów, a także umiejętności obniżania kosztów eksploatacji naszego oprogramowania, umiejętność optymalizacji oprogramowania.

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

0.1. Prezentacja treści. Przedstawimy rosnący ciąg podjęzyków języka Loglan. i dla każdego z nich postaramy się określić składnię, semantykę i wirtualny komputer realizujący obliczenia programów napisanych w tym języku. Semantykę będziemy definiować podając odpowiednią definicję obliczenia. Podamy też aksjomaty rachunku programów odpowiednie dla kolejnego podjęzyka $\mathcal{L}_i$.
W każdym rozdziale znajdą się przykłady dowodów poprawności programów.

Formalizm oparty będzie na rachunku programów czyli logice algorytmicznej.

Poznawać będziemy rosnący ciąg podjęzyków języka Loglan'82.

$$\mathcal{L}_0 \subset \mathcal{L}_1 \subset \mathcal{L}_2 \subset \mathcal{L}_3 \subset \ldots \mathcal{L}_{10} \subset \text{Loglan}$$

i ciąg coraz mocniejszych abstrakcyjnych komputerów

$$\mathcal{K}_0 \sqsubset \mathcal{K}_1 \sqsubset \mathcal{K}_2 \sqsubset \mathcal{K}_3 \sqsubset \ldots \mathcal{K}_{10} \sqsubset \text{VLP}$$

a także ciąg teorii algorytmicznych

$$\mathcal{T}_0 \subset \mathcal{T}_1 \subset \mathcal{T}_2 \subset \mathcal{T}_3 \subset \ldots \mathcal{T}_{10}.$$

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, (ciągi (skończone) instrukcji przypisania)
- $\mathcal{L}_3$ – język programów elementarnych (instrukcje for oraz if),
- $\mathcal{L}_4$ – język programów z tablicami,
- $\mathcal{L}_5$ – język programów iteracyjnych (instrukcje while),

- ...
- Loglan

Każdemu językowi $\mathcal{L}_i$ odpowiada abstrakcyjny komputer $\mathcal{K}_i$, który potrafi wykonywać programy zapisane w tym języku.
Wirtualny komputer $\mathcal{K}_i$ potrafi wykonywać programy napisane w języku $\mathcal{L}_i$. Światły programista potrafi wykorzystać wiedzę zawartą w teorii $\mathcal{T}_i$ by przedstawić dowód, że program napisany w języku $\mathcal{L}_i$ ma takie a nie inne własności semantyczne.
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 własności programów z języka $\mathcal{L}_i$. Zamiar ten będziemy realizować dla języków opisanych w pierwszej 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? popraw!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.
Tym, którzy lubią wyzwania proponujemy by wzięli udział w programie badania jakie teorie opisują programowanie z współprogramami i procesami. Należy spodziewać się czegoś nowego.

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 2.1. 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. Zamiast pary słów: program nazwa_programu; możesz napisać krótko block. Tak, program jest blokiem. O blokach piszemy obszerniej w rozdziale 9, str. 167.
⌈ 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). rozwiń Będziemy też mówić wirtualny procesor loglanowski – VLP. Komputery $\mathcal{K}_i$ tworzą ciąg maszyn coraz mocniejszych i przybliżają loglanowski procesor wirtualny VLP.

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 można dowodzić własności semantyczne programów. TAK!
(5) W tym celu należy wyrazić własność semantyczną przez odpowiednią formułę.
(6) Wykazanie, że formuła ta jest prawdziwa to to samo co wykazanie, że prawdą jest iż 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 skonstruować system aksjomatów i reguł wnioskowania pozwalający przeprowadzać dowody formuł algorytmicznych, a więc semantycznych własności programów.
(10) Zbadać taki system, czy nie zawiera sprzeczności i czy jest kompletny.
(11) Gromadzić doświadczenia w stosowaniu narzędzia jakim jest rachunek programów.

Część 1

Rachunek programów

ROZDZIAŁ 3

$\mathcal{L}_1$ Wyrażenia

$$\mathcal{L}_1 \subsetneq \mathcal{L}_2 \subsetneq \mathcal{L}_3 \subsetneq \mathcal{L}_4 \subsetneq \mathcal{L}_5 \subsetneq \mathcal{L}_6 \subsetneq \mathcal{L}_7 \subsetneq \mathcal{L}_8 \subsetneq \mathcal{L}_9 \subsetneq \mathcal{L}_{10}$$

Ten rozdział poświęcony jest wyrażeniom. Wyrażenia, potocznie nazywane wzorami, są niezbędnym składnikiem każdego języka programowania. Co ważniejsze, nie sposób zrozumieć jak dany program działa, bez zrozumienia semantycznej treści wyrażeń w nim występujących. Pewne wyrażenia można zastąpić innymi, równoważnymi. To jest klucz do ulepszania programu.
Z drugiej strony, jedno i to samo wyrażenie może mieć zupełnie inny sens w różnych miejscach tego samego programu. Wartość wyrażenia zależy od kontekstu w jakim występuje.

Każdy język programownia $\mathcal{L}$ jest układem dwu zbiorów: alfabetu $\mathcal{A}$ oraz zbioru napisów poprawnie zbudowanych $\mathcal{W}$

$$\mathcal{L} = \langle \mathcal{A}, \mathcal{W} \rangle.$$

Z kolei zbiór $\mathcal{W} = \mathcal{E} \cup \mathcal{I} \cup \mathcal{D}$. Zbiór $\mathcal{E}$ jest zbiorem wyrażeń arytmetycznych, Boolowskich, znakowych, napisowych i innych, o którycych opowiemy w dalszych rozdziałach. Zbiór $\mathcal{I}$, to zbiór instrukcji. Zbiór $\mathcal{D}$ jest zbiorem deklaracji. Język $\mathcal{L}_1$ rozszerza język $\mathcal{L}_0$ i wprowadza wyrażenia oraz instrukcje czytania read. Omówimy nie tylko składnię wyrażeń, ale także przypisane im znaczenie, tzn. ich semantykę. Podręczniki programowania, na ogół, nie poświęcają odpowiedniej uwagi wyrażeniom i ograniczają się do przedstawienia składni wyrażeń.
W tym rozdziale omawiamy wyrażenia pięciu typów pierwotnych, ich semantykę oraz własności działań na elementach tych typów i relacji jakie zachodzą pomiędzy elementami. Typy pierwotne to integer,

real, Boolean, char (znak), string (napis). Wyrażeniom typów integer oraz real nadaje się wspólną nazwę: wyrażenia arytmetyczne. Znaczeniem wyrażenia całkowito-liczbowego integer jest k-argumentowa funkcja ze zbioru liczb całkowitych $\mathbb{Z}$ w zbiór $\mathbb{Z}$. Np. Znaczeniem wyrażenia ax^2+bx+c jest cztero-argumentowa funkcja

$$(ax^2 + bx + c)_{\mathbb{Z}} \colon \mathbb{Z}^4 \to \mathbb{Z}.$$

Sposób obliczania wartości tej funkcji jest oczywisty. Zauważ różnicę

$$(ax^2 + bx + c)_{\mathbb{R}} \colon \mathbb{R}^4 \to \mathbb{R}.$$

Ten sam napis może oznaczać zupełnie inną funkcję. Ponadto rożne napisy mogą oznaczać tę samą funkcję

$$((ax + b)x + c)_{\mathbb{R}} \colon \mathbb{R}^4 \to \mathbb{R}.$$

To spostrzeżenie uprzytamnia nam jak wiele można zyskać, lub stracić, jeśli potrafimy umiejętnie wykorzystać własności działań. Wyrażenia Boolowskie czyli formuły logiczne bezkwantyfikatorowe to także opisy funkcji. Tym razem wartości funkcji to prawda lub fałsz. Poprzednie uwagi stosują się do wyrażeń boolowskich. Wyrażenia arytmetyczne stanowią ważny składnik instrukcji przypisania (o tym piszemy w następnym rozdziale) a także stanowią czskładnik wyrażeń Boolowskich. Wyrażenia boolowskie stanowią ważny składnik instrukcji warunkowych iforaz instrukcji iteracji while.
Pozostałe dwa typy wyrżeń też są ważne.
Więc jeśli chcesz nie tylko napisać program, ale i zrozumieć jak on działa? Jeśli chcesz swemu programowi nadać optymalną postać, by działał bardziej wydajnie, to musisz opanować umiejętność zastępowania jednego wyrażenia innym równoważnym, o mniejszym koszcie. Precyzyjną definicję semantyki wyrażeń zawdzięczamy Alfredowi Tarskiemu [Tar33].

Wyrażenia są bardzo ważnym składnikiem zbioru napisów poprawnie zbudowanych, tj. napisów, które należą do języka. Ważne jest by, programista umiał udzielać odpowiedzi na pytania dotyczące składni, semantyki i analizy wyrażeń, podobne do wymienionych poniżej:

Składnia Czy dany napis ω jest poprawnie napisanym wyrażeniem?
Odpowiedzi na to pytanie udzieli kompilator. Dzięki niemu, nauka tworzenia poprawnych wyrażeń przychodzi nam łatwo. Nie musisz się martwić. Jeśli popełniłeś błąd, to kompilator wskaże Ci miejsce i rodzaj błędu.

Semantyka Czy wartości wyrażeń ω i θ są równe (tj. czy jedno wyrażenie można zastąpić drugim?
Jaki jest koszt danego wyrażenia?
Czy jest do pomyślenia wyrażenie o mniejszym koszcie i równe danemu?
Podobne pytanie to pytanie porównujące dwa wyrażenia arytmetyczne, czy formuła $(\omega < \theta)$ jest prawdziwa?

Analiza Czy potrafisz podać argumenty (dowód) uzasadniające Twój osąd? Czy napisany przez kogoś dowód równości $\omega = \theta$ jest poprawny? Czy argumenty w nim użyte są trafne?
Z jakich własności(tj. aksjomatów) typu pierwotnego (lub obiektowego) wyprowadzono podany dowód?

Pytania z drugiej i trzeciej grupy są trudniejsze i przyda się mieć pewien trening matematyczny.
W każdym języku programowania zbiór wyrażeń pełni ważną rolę. Każdy język programowania definiuje ten zbiór w nieco inny sposób. Różnice pomiędzy wyrażeniami w jednym i w drugim języku programowania nie są ani duże ani istotne.

W rozdziale 10 zobaczymy, że każdy program może definiować zbiór wyrażeń na swój sposób. Co więcej, przechodząc od jednego do drugiego modułu programu zmieniamy środowisko i zbiór wyrażeń dopuszczalnych w tym module. W tym rozdziale nie będziemy przykładać większej wagi do różnic pomiędzy definicjami wyrażeń w rozmaitych językach, i przyjmiemy, (bez zmniejszenia ogólności naszych rozważań,) uproszczone definicje wyrażeń całkowito-liczbowych i wyrażeń arytmetycznych typu real. W dalszych rozdziałach definicja wyrażenia będzie się zmieniać. Nie powinno Cię to zaskakiwać.

Wyrażenia występują też w matematyce i we wszystkich dziedzinach w których matematyka jest stosowana. Szerzej używana nazwa wzory jest bardzo często spotykanym odpowiednikiem pojęcia wyrażenia. W językach programowania zbiór wyrażeń dzieli się na zbiór wyrażeń arytmetycznych, znakowych i tekstowych i zbiór wyrażeń boolowskich. W podręcznikach logiki (matematycznej) zbiór wyrażeń dzieli się na zbiór termów tj. wyrażeń nazwowych i zbiór formuł tj. wyrażeń zdaniowych. Formuły bez kwantyfikatorów (tj. formuły otwarte) i wyrażenia boolowskie to mniej więcej to samo.
W latach trzydziestych XX wieku Alfred Tarski opublikował pracę [Tar33], w której zaproponował by wyrażenia pojmować jako opisy funkcji ze zbioru wartościowań zmiennych w zbiór odpowiednich wartości. W ten sposób zapoczątkowana została nowoczesna semantyka wyrażeń. Dla programistów ten sposób myślenia jest naturalny: W programie występują zmienne. W danej chwili zmienne te mają jakieś wartości - jest to wartościowanie lub inaczej stan pamięci. W wyrażeniu występują także znaki działań np. znak $+$ reprezentujący działanie dodawania $\oplus$ i znak $/$ oznaczający działania dzielenia $\oslash$. Znaczeniem wyrażenia $a + b \cdot c$ jest więc trój-argumentowa funkcja, która każdemu wartościowaniu zmiennych $v : \frac{a \quad b \quad c}{v_a \; v_b \; v_c}$ przyporządkowuje wartość $v_a \oplus (v_b \odot v_c)$. Poniżej podajemy dokładną definicję znaczenia przypisanego wyrażeniu. Na razie, zapamiętajmy, że dla typu pierwotnego $\mathbb{A}$ wyrażeniu ω przyporządkowane jest odwzorowanie $\omega_{\mathbb{A}}$ ze zbioru wartościowań zmiennych w uniwersum typu $\mathbb{A}$

$$\omega_{\mathbb{A}} \colon A^V \to A.$$

Wspomnijmy w tym miejscu, że szczególną rolę w pracy Tarskiego odgrywa nadzbiór zbioru wyrażeń boolowskich tj. funkcje zdaniowe. W wyrażeniach tych dopuszcza się kwantyfikatory $\forall$ - dla każdego oraz $\exists$ -istnieje. Tarski podał definicje spełniania i prawdziwości wyrażeń logicznych.

Zbiór V zmiennych jest wyliczony w deklaracjach programu i jest skończony. Wartościowanie v jest odwzorowaniem ze zbioru zmiennych V w zbiór wartości A tj. elementów ustalonego typu, $v \in A^V$.

Programy w języku $\mathcal{L}_1$ dopuszczają deklaracje zmiennych i stałych pięciu typów pierwotnych. Instrukcjami są instrukcje drukowania. Zauważ jednak, że teraz w instrukcjach języka $\mathcal{L}_1$, możemy stosować bogatszy repertuar wyrażeń niż było to możliwe w języku $\mathcal{L}_0$.
Instrukcje drukowania są elementem programu niezbędnym do ujawnienia obliczonych wartości wyrażeń. Raz napisany program można wykonywać wielokrotnie. Jeśli w programie wystąpią instrukcje czytania (read) to ułatwi to nam eksperymentowanie z zmienjącymi się danymi początkowymi. W przeciwnym przypadku powtarzane obliczenia będa dawały powtarzające się wyniki.

Po kolei omówimy różne rodzaje wyrażeń: wyrażenia całkowito-liczbowe $\mathcal{WZ}$, wyrażenia typu real (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 drugiej części tej książki. Szczególne wartości wyrażeń obiektowych to obiekty współprogramów oraz obiekty procesów. O tym będzie mowa w części trzeciej.
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

1. Przykłady programów

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

Przykład 3.1. Te trzy programy różnią się. Wielokrotne wykonanie programu Delta1 zawsze zwraca ten sam wynik. Program Delta2 może być bardziej użyteczny. Każde wykonanie tego programu wymaga podania wielkości a,b,c z klawiatury. Ale czużytkownik będzie wiedział czego program oczekuje?, Trzeci program Delta3 nawiązuje dialog z użtkownikiem i formułuje swoje oczekiania. .

```
program Delta1;
  const a=2, b=5, c=1
begin
  writeln("delta=",
    b*b-4*a*c)
end
```

```
program Delta2;
  var a,, b, c : real
begin
  readln(a,b,c);
  writeln("delta=",
    b*b-4*a*c)
end
```

```
program Delta3;
  var a,, b, c : real
begin
  writeln("podaj wartości a, b,c ");
  readln(a,b,c);
  writeln("delta=",
    b*b-4*a*c)
end
```

I jeszcze jeden przykład. Jaką rolę odgrywają deklaracje var zmiennych x i y?

Przykład 3.2. W tym przykładzie występują wyrażenia typu znakowego i zdaniowego.

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

2. Typy pierwotne

Typy pierwotne: Boolean $\mathbb{B}_0$, integer $\mathbb{Z}$, real $\mathbb{R}$, char (znak) $\mathbb{C}$, string(napis. tekst) $\mathbb{S}$ są strukturami algebraicznymi. Ma to dla nas znaczenie o tyle, że struktura algebraiczna to nie tylko zbiór wartości, lecz także działania na wartościach – elementach struktury. Programiści stosują nazwę struktury danych,

mając na myśli typy złożone definiowane przy pomocy klas i wskaźników. Warto tą nazwą objąć także typy pierwotne.
Zakłada się, że wirtualny komputer potrafi realizować działania opisane w wyżej wymienionych strukturach[1].

2.1. Typ integer. Typ integer jest strukturą algebraiczną

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

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 $\mathbb{Z}$ mamy dwie relacje $\ominus$ równości i $\oslash$ 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
$\div$	$: Z \times \{Z \setminus \{0\}\} \to Z$	dzielenie całkowitoliczbowe
modulo	$: Z \times \{Z \setminus \{0\}\} \to Z$	reszta z dzielenia całkowito liczbowego

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

$\ominus: Z \times Z \to \{0, 1\}$	równość
$\oslash: Z \times Z \to \{0, 1\}$	mniejszość

2.2. Typ Boolean. Ten typ to znana dwu-elementowa algebra Boole'a. Uniwersum składa się z dwu elementów 0 i 1, oznaczających odpowiednio wartości fałsz i prawda. W algebrze tej rozważamy działania dwuargumentowe: kresu górnego $\cup$ i kresu dolnego $\cap$ oraz jednoargumentowe działanie uzupełnienia $-$. Inne operacje mogą być zdefiniowane przy pomocy tych trzech.

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

[1]Być może zastanawiasz się dlaczego nie mówimy o innych typach pierwotnych np. long int, etc. Po poerwsze znaczenie tych nazw zmienia się dość szybko. Co ważniejsze pragniemy omówić własności działań, bo to jest wżniejsze.

$\cup$	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

2.3. Typ real. Ta struktura danych to zbiór R 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.

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

Mamy tu cztery działania dwuargumentowe:

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

Zwróć uwagę na dziedzinę operacji dzielenia. Wynik dzielenia nie jest określony gdy dzielnik jest równy 0.
Ponadto mamy, dwie dwuargumentowe relacje (lub jeśli wolisz dwie funkcje charakterystyczne relacji)

$\ominus\colon R \times R \rightarrow \{0,1\}$	równość
$\lessdot\colon R \times R \rightarrow \{0,1\}$	mniejszość

2.4. Typ znakowy - char. Ten typ to skończony zbiór znaków jakie mogą być wprowadzane z klawiatury lub wyświetlane na ekranie.

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

Nie ma żadnych operacji na znakach. Każdy znak jest atomem. Jedyną relacja to relacja równości znaków.

Uwaga. W kolejnych rozdziałach wprowadzimy funkcje *ord* i *char*.

$$ord\colon C \to Z$$

$$chr\colon Z \to C$$

I przy ich pomocy określimy relację mniejszości w zbiorze znaków.
Koniec uwagi

2.5. Typ tekstowy - string. Elementami tego typu są skończone ciągi znaków. W języku programowania Loglan'82 programista nie ma wielkiego wyboru, może zadeklarowć stałą typu string. Program nie może wykonywać żadnych działań na stringach. Jedynie podczas drukowania (instrukcja write) dokonywana jest konkatenacja tekstów. Program nie może wykorzystać wyniku konkatenacji tekstów.
Wiele języków, np. C, C++, Java i in. zezwala na przypisywanie zmiennym tekstowym wyniku konkatenacji dwu tekstów i dopuszcza inne działania na tekstach.
W Loglanie programista może przypisać zmmiennej typu string wartość pewnej stałęj tekstowej.

3. Składnia wyrażeń

W języku $\mathcal{L}_1$ zachowana zostaje opisana wcześniej struktura programu. Zbiór deklaracji zawiera deklaracje zmiennych i stałych typów pierwotnych. Zbiór wyrażeń języka $\mathcal{L}_1$ jest znacznie bogatszy od zbioru wyrażeń dopuszczalnych w języku $\mathcal{L}_0$. Zbiór instrukcji instrukcje drukowania, ale zauważ, występujące w nich wyrażenia są teraz bardziej złożone. Dla wygody eksperymentatora opisujemy też instrukcje read – wczytywania wartości zmiennych.

3.1. Deklaracje zmiennych i stałych.

Definicja 3.1. Niech ν oznacza identyfikator, ω niech będzie nazwą pewnego typu pierwotnego, $\omega \in$ {Boolean, integer, real, char, string}. Deklaracja zmiennej ma następująca postać

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

Mówimy: zmienna ν jest zadeklarowana, a także, zmienna ν jest 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 3.2. 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 c4="Witaj swiecie";
const c15= (c1+c2)/2;
```

Typy stałych c1, c2,c3,c4 są łatwe do odgadnięcia. Czy potrafisz wskazać typ stałej c15?

3.2. Wyrażenia typów pierwotnych. Zmienne i stałe są najprostszymi (atomowymi) wyrażeniami. Niech T będzie jednym z pięciu typów pierwotnych. Każdy

z pięciu zbiorów wyrażeń typu pierwotnego T jest algebrą A_T. Działaniami tej algebry są funktory działań w typie T.??

3.2.1. Wyrażenia całkowito-liczbowe $\mathcal{WZ}$. Podana poniżej definicja wyrażenia całkowito-iczbowego różni się od definicji jaką znajdziesz w Loglanie i większości innych języków programowania. Na ile istotna jest ta różnica? Spróbuj odpowiedzieć na to pytanie rozwiązując ćwiczenie ...

Definicja 3.3. Zbiór wyrażeń całkowito-liczbowych jest to najmniejszy zbiór napisów $\mathcal{WZ}$ taki, że

(i) każda zadeklarowana zmienna całkowito-liczbowa z należy do zbioru $\mathcal{WZ}$, $z \in \mathcal{WZ}$, każda liczba całkowita c należy do zbioru $\mathcal{WZ}$, $c \in \mathcal{WZ}$, każda zadeklarowana stała całkowito-liczbowa należy do zbioru $\mathcal{WZ}$,

(ii) jeśli do zbioru $\mathcal{WZ}$ należą napisy τ_1 oraz τ_2, to do zbioru $\mathcal{WZ}$ należą też napisy

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

Przykłady wyrażeń całkowito-liczbowych
...Dla dalszych rozważań istotne jest następujące spostrzeżenie

Lemat 3.1. Zbiór $\mathcal{WZ}$ wyrażeń całkowito-liczbowych jest algebrą z działaniami

$$\begin{aligned} +&: \mathcal{WZ} \times \mathcal{WZ} \to \mathcal{WZ} \\ -&: \mathcal{WZ} \times \mathcal{WZ} \to \mathcal{WZ} \\ *&: \mathcal{WZ} \times \mathcal{WZ} \to \mathcal{WZ} \\ \text{div}&: \mathcal{WZ} \times \mathcal{WZ} \to \mathcal{WZ} \\ \text{mod}&: \mathcal{WZ} \times \mathcal{WZ} \to \mathcal{WZ} \end{aligned}$$

określonymi w następujący sposób:

- dla każdego funktora dwuargumentowego $\{+, -, *, \text{div}, \text{mod}\}$, wyliczonego powyżej i kazdej pary termów τ_1 i τ_2 wynikiem działania $+$ jest term $(\tau_1 + \tau_2)$, ... wynikiem działania mod jest term $(\tau_1 \text{mod} \tau_2)$,

- każda liczba i każda stała całkowito-liczbowa jest wynikiem działania zero-argumentowego.

Zbiór zmiennych całkowito-liczbowych jest zbiorem generatorów tej algebry. Oznacza to tyle, że

Lemat 3.2. Dowolne odwzorowanie

$$v\colon V \to J$$

może być rozszerzone do homomorfizmu

$$h\colon \mathcal{WZ} \to J$$

Rzeczywiście, funkcję h definiujemy przez indukcję:

1) dla każdej zmiennej całkowito liczbowej x, $h(x) = v(x)$,
2) dla każdej liczby i każdej stałej $h(c) = c$,
3) przypuśćmy, że term τ jest postaci $(\tau_1 \odot \tau_2)$, znak $\odot$ oznacza jeden z pięciu funktorów dwuargumentowych, i załóżmy że funkcja h jest określona dla termów τ_1 i τ_2, kładziemy

$$h(\tau_1 \odot \tau_2) = h(\tau_1) \odot h(\tau_2).$$

Szczególny przypadek, warty rozwazenia to rozszerzenie funkcji

$$s : s\colon V \to \mathcal{WZ}.$$

Zgodnie z powyższymi uwagami funkcję s możemy rozszerzyc na cały zbiór $\mathcal{WZ}$ wystarczy przyjąć

$$s(\tau_1 \odot \tau_2) \stackrel{df}{=} \big(s(\tau_1) \odot s(\tau_2)\big)$$

Przykłady
Każde wyrażenie całkowito-liczbowe τ może być przedstawione jako drzewo. Uwaga o rozszerzeniu funkcji s sprowadza się do łatwej obserwacji:Dla każdego wyrażenia całkowito-liczbowego τ, zbiór liści drzewa D_τ tego wyrażenia jest równy zbiorowi zmiennych oraz stałych całkowito-liczbowych występujących w tym wyrażeniu.

3.2.2. 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 3.4. 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 $\mathcal{WB}$, napisy "true" oraz "false" należą do zbioru $\mathcal{WB}$.

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

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

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

$$(\alpha \text{ or } \beta), \qquad (\alpha \text{ and } \beta), \qquad \text{not } \alpha$$

3.2.3. Wyrażenia arytmetyczne typu real.

Definicja 3.5. Zbiór wyrażeń arytmetycznych typu real jest to najmniejszy zbiór napisów $\mathcal{WR}$ taki, że

(i) każda zmienna x którą zadeklarowano jako typu real należy do zbioru $\mathcal{WR}$, $x \in \mathcal{WR}$, każda liczba rzeczywista c należy do zbioru $\mathcal{WR}$, $c \in \mathcal{WR}$, każda zadeklarowana stała typu real należy do zbioru $\mathcal{WR}$,

(ii) jeśli do zbioru $\mathcal{WR}$ należą napisy τ_1 oraz τ_2, to do zbioru $\mathcal{WR}$ należą też napisy

$$(\tau_1 + \tau_2), (\tau_1 - \tau_2), (\tau_1 * \tau_2), (\tau_1 \,/\, \tau_2).$$

Przykłady wyrażeń typu real:
x, y gdy zmienne x i y są zadeklarowane typu real,
1.23, -2.34, 4.0E4
(x+(7.8-y))

Uwaga. Ani te przykłady, ani podana powyżej składnia nie wyczerpują definicji wyrażeń arytmetyxznych typu real!

3.2.4. Wyrażenia znakowe.

Definicja 3.6. Wyrażeniem znakowym jest zmienna zadeklarowana jako zmienna typu znakowego lub stała znakowa.

Przykłady.

3.2.5. Wyrażenia tekstowe.

Definicja 3.7. Wyrażeniem tekstowym jest zmienna zadeklarowana jako string lub stała tekstowa.

Przykłady.
"Witaj smutku"
s
Gdzie s jest zmienną zadeklarowaną jako string.
drzewa wyrażeń
bigskip
Każde wyrażenie, niezależnie od jego typu, może być przedstawione w postaci drzewa.
Przykłady.

3.3. Instrukcje. Instrukcjami języka są polecenia drukowania: write, por. sekcja 7.6 i polecenia read, zob. poniżej.

3.4. Instrukcja read. W tym podrozdziale poznamy instrukcję read. Ta atomowa instrukcja pozwala przypisać zmiennej wartość wprowadzaną z klawiatury komputera.
Przykłady
Składnia
Polecenie wczytaj wartość w i przypisz ją zmiennej ma następującą postać

$$\text{read}(z)$$

Zmienna z może być zmienną typu integer, real, char lub string[2]. Przypadek gdy zmienna z jest typu char

[2]Przypominamy, w obecnej wersji języka Loglan zmienna typu string nie może być argumenteminstrukcji read

jest najprostszy: jeden znak c naciśnięty na klawiaturze zostaje przypisany zmiennej z. Polecenie read(z) gdy z jest zmienną typu integer lub real spowoduje odczytanie maksymalnie długiej sekwencji znaków, która reprezentuje liczbę typu zgodnego z typem zmiennej z. Polecenie readln ...
Instrukcje read pozwalają na zainicjalizowanie początkowego wartościowania zmiennych.

4. Semantyka wyrażeń

Znaczeniem wyrażenia jest odwzorowanie, które danemu wartościowaniu v zmiennych przypisuje element ze zbioru wartości U. Niech ω będzie wyrażeniem, v wartościowaniem zmiennych $- v : V \to U$.

$$\omega_{\mathbb{A}} : U^V \to U$$

4.1. Wyrażenia całkowito-liczbowe. Znaczeniem wyrażenia całkowito-liczbowego τ jest funkcja $\tau_{\mathbb{Z}}$ ze zbioru Z^V wartościowań zmiennych w zbiór liczb całkowitych

$$\tau_{\mathbb{Z}} \colon Z^V \to Z.$$

Obliczanie wartości wyrażenia
Wartość $\tau(v)$ wyrażenia τ dla ustalonego wartościowania v wyznacza się zgodnie z następującą procedurą:

Wyrażenie τ	wartość $\tau(v)$		wynosi
τ **jest liczbą** l	$\tau(v)$	$\stackrel{df}{=}$	l
τ **jest stałą** l	$\tau(v)$	$\stackrel{df}{=}$	l
τ **jest zmienną** z	$\tau(v)$	$\stackrel{df}{=}$	$v(z)$
$(\tau_1 + \tau_2)$	$(\tau_1 + \tau_2)(v)$	$\stackrel{df}{=}$	$(\tau_1(v) \oplus \tau_2(v))$
$(\tau_1 - \tau_2)$	$(\tau_1 - \tau_2)(v)$	$\stackrel{df}{=}$	$(\tau_1(v) \ominus \tau_2(v))$
$(\tau_1 * \tau_2)$	$(\tau_1 * \tau_2)(v)$	$\stackrel{df}{=}$	$(\tau_1(v) \otimes \tau_2(v))$
$(\tau_1 \,\mathbf{div}\, \tau_2)$	$(\tau_1 \,\mathbf{div}\, \tau_2)(v)$	$\stackrel{df}{=}$	$\begin{cases} \tau_1(v) \div \tau_2(v)) & \textbf{gdy } \tau_2(v) \neq 0 \\ \textbf{nieokreślone} & \textbf{w.p.p.} \end{cases}$
$(\tau_1 \,\mathbf{mod}\, \tau_2)$	$(\tau_1 \,\mathbf{mod}\, \tau_2)(v)$	$\stackrel{df}{=}$	$\begin{cases} \tau_1(v) \,\mathbf{modulo}\, \tau_2(v)) & \textbf{gdy } \tau_2(v) \neq 0 \\ \textbf{nieokreślone} & \textbf{w.p.p.} \end{cases}$

3

Podana powyżej definicja znaczenia cieszy się dwoma bardzo ważnymi własnościami

Lemat 3.3. Dla każdego wyrażenia całkowito-liczbowego τ i dla każdego wartościowania zmiennych istnieje obliczenie skończone wartości wyrażenia $\tau(v)$, które ma nie więcej kroków niż długość wyrażenia τ.

Dowód. Krok obliczenia polega albo na odczycie wartości zmiennej x w wartościowaniu v, albo na wykonaniu odpowiedniej operacji na otrzymanych wcześniej wartościach podwyrażeń $\tau_1(v)$ i $\tau_2(v)$. Dla termów o długości 1 teza jest oczywiście prawwdziwa. Załóżmy , że teza jest pawdziwa dla wszystkich termów o długości nie większej niż k Dowód przebiega przez indukcję ze względu na długość wyrażenia □

Lemat 3.4. (o jednoznaczności) Wartość wyrażenia jest wyznaczona jednoznacznie. Jeśli istnieją dwa różne obliczenia wartości termu τ dla wartościowania zmiennych v, to wyniki obliczeń są równe.

Przykłady
pokaz jednoznaczność Można obliczać w różnych porządkach. Obliczenie odwiedza wierzchołki drzewa

[3] w rzeczywistości wirtualny komputer wykryje taką sytuację i zgłosi sygnał błędu "division by zero"

wyrażenia ... 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 całkowitoliczbowego ω znajdź najbardziej optymalne równoważne wyrażenie, może okazać się bardzo trudne.
Co więcej, umiejętność udowodnienia, że pewne wyrażenia ω i θ są równe, tj. funkcje $\omega_{\mathfrak{A}}$ i $\theta_{\mathfrak{A}}$ opisywane przez te wyrażenia są równe, ma zasadniczą wartość w procesie analizowania semantycznych własności programu. Zobaczymy to wielokrotnie w dalszym ciągu.
Zadania
Własności (aksjomaty) struktury integer, struktury real.

4.2. Wyrażenia arytmetyczne typu real. Obliczanie wartości wyrażeń arytmetycznych typu real definiujemy w sposób analogiczny ... Znaczeniem wyrażenia τ typu real jest funkcja $\tau_{\mathbb{R}}$ ze zbioru R^V wartościowań zmiennych w zbiór liczb rzeczywistych $\mathbb{R}$

$$\tau_{\mathbb{R}} : R^V \to R.$$

Obliczanie wartości wyrażenia
Wartość $\tau(v)$ wyrażenia τ dla ustalonego wartościowania v wyznacza się zgodnie z następującą procedurą:

Wyrażenie τ	wartość $\tau(v)$		wynosi
τ jest liczbą l	$\tau(v)$	$\stackrel{df}{=}$	l
τ jest stałą l	$\tau(v)$	$\stackrel{df}{=}$	l
τ jest zmienną z	$\tau(v)$	$\stackrel{df}{=}$	$v(z)$
$(\tau_1 + \tau_2)$	$(\tau_1 + \tau_2)(v)$	$\stackrel{df}{=}$	$(\tau_1(v) \oplus \tau_2(v))$
$(\tau_1 - \tau_2)$	$(\tau_1 - \tau_2)(v)$	$\stackrel{df}{=}$	$(\tau_1(v) \ominus \tau_2(v))$
$(\tau_1 * \tau_2)$	$(\tau_1 * \tau_2)(v)$	$\stackrel{df}{=}$	$(\tau_1(v) \otimes \tau_2(v))$
$(\tau_1 \,/\, \tau_2)$	$(\tau_1 \,/\, \tau_2)(v)$	$\stackrel{df}{=}$	$\begin{cases} \tau_1(v) \oslash \tau_2(v)) & \textbf{gdy } \tau_1(v) \neq 0 \\ \textbf{nieokreślone} & \textbf{w.p.p.} \end{cases}$

Podana powyżej definicja znaczenia cieszy się dwoma bardzo ważnymi własnościami

Lemat 3.5. Dla każdego wyrażenia τ typu real i dla każdego wartościowania zmiennych obliczenie wartości $\tau(v)$ istnieje obliczenie skończone, które ma nie więcej kroków niż długość wyrażenia τ.

Dowód. Krok obliczenia polega albo na odczycie wartości zmiennej x w wartościowaniu v, albo na wykonaniu odpowiedniej operacji na otrzymanych wcześniej wartościach podwyrażeń $\tau_1(v)$ i $\tau_2(v)$. Dla termów o długości 1 teza jest oczywiście prawwdziwa. Załóżmy , że teza jest pawdzziwa dla wszystkich termów o długości nie większej niż k Dowód przebiega przez indukcję ze względu na długość wyrażenia $\square$

Lemat 3.6. (o jednoznaczności) Wartość wyrażenia jest wyznaczona jednoznacznie. Jeśli istnieją dwa różne obliczenia wartości termu τ dla wartościowania zmiiennich v, to wyniki obliczeń są równe.

Przykłady
Wyrażenia $\tau : ((4 * (x * x)) - (9x + 5))$ i $\eta : ((4 * x - 9) * x) + 5$ mają tę samą wartość, wyznaczają tę samą funkcję $\tau_N \colon N \to N$
...
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.

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

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

Funkcja ta jest określona przez indukcję ze względu na długość wyrażenia w następujący sposób: Niech V oznacza zbiór zmiennych. Wartościami zmiennych boolowskich sa albo true albo false. Czasami zamiast 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
$(\tau \leq \nu)$	$val((\tau \leq \nu), v) = val(\tau, v) \leqslant val(\nu, v)$
$q \in V_{Boolean}$	$val(q, v) = v(q)$
$\neg\alpha$	$val((\neg\alpha), v) = -val(\alpha, v)$
$(\alpha \vee \beta)$	$val((\alpha \vee \beta), v) = val(\alpha, v) \cup val(\beta, v)$
$(\alpha \wedge \beta)$	$val((\alpha \wedge \beta), v) = val(\alpha, v) \cap val(\beta, v)$

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.
Jeśli wartością wyrażenia boolowskiego α dla wartościowania zmiennych v jest true, to mówimy, że wartościowanie v spełnia warunek α. Jeśli każde wartościowanie v spełnia warunek α, to mówimy warunek (formuła) α jest prawdziwy.

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

Optymalizacja? Dla dalszych rozważań istotne jest następujące spostrzeżenie

Lemat 3.7. Zbiór $\mathcal{WB}$ wyrażeń boolowskich jest algebrą z działaniami

$$\begin{aligned} \text{and}&\colon \mathcal{WZ} \times \mathcal{WZ} \to \mathcal{WZ} \\ \text{or}&\colon \mathcal{WZ} \times \mathcal{WZ} \to \mathcal{WZ} \\ \text{not}&\colon \mathcal{WZ} \times \mathcal{WZ} \to \mathcal{WZ} \end{aligned}$$

określonymi w następujący sposób:

- dla każdego funktora dwuargumentowego $\{\text{and}, \text{or}, \text{not}\}$, wyliczonego powyżej i kazdej pary wyrażeń boolowskich α i β, wynikiem działania and jest wyrażenie boolowskie $(\alpha \text{ and } \beta)$, wynikiem działania or jest wyrażenie boolowskie $(\alpha \text{ or } \beta)$, wynikiem działania not jest wyrażenie boolowskie $(\text{not}\,\alpha)$.
- stałe boolowskie true i false są wynikami odpowiedniego działania zero-argumentowego.

Zbiór zmiennych boolowskich i zbiór elementarnych wyrażeń boolowskich postaci $(\tau_1 \odot \tau_2)$ (znak $\odot$ reprezentujeznak eówności, znak mniejszości znak większości lub znak negacji takiej relacji) jest zbiorem generatorów tej algebry. Oznacza to tyle, że

Lemat 3.8. Dowolne odwzorowanie

$$v\colon V \to J$$

może być rozszerzone do homomorfizmu

$$h\colon \mathcal{WZ} \to J$$

Rzeczywiście, funkcję h definiujemy przez indukcję:

1) dla każdej zmiennej całkowito liczbowej x, $h(x) = v(x)$,
2) dla każdej liczby i każdej stałej $h(c) = c$,
3) przypuśćmy, że term τ jest postaci $(\tau_1 \odot \tau_2)$, znak $\odot$ oznacza jeden z pięciu funktorów dwuargumentowych, i załóżmy że funkcja h jest określona dla termów τ_1 i τ_2, kładziemy

$$h(\tau_1 \odot \tau_2) = h(\tau_1) \odot h(\tau_2).$$

Szczególny przypadek, warty rozwazenia to rozszerzenie funkcji

$$s : s\colon V \to \mathcal{WZ}.$$

Zgodnie z powyższymi uwagami funkcję s możemy rozszerzyc na cały zbiór $\mathcal{WZ}$ wystarczy przyjąć

$$s(\tau_1 \odot \tau_2) \stackrel{df}{=} \big(s(\tau_1) \odot s(\tau_2)\big)$$

Przykłady

> Nota 1. Każde wyrażenie całkowito-liczbowe (każde wyrażenie arytmetyczne typu real, każde wyrażenie boolowskie) może być przedstawione jako drzewo.
> Liśćmi tego drzewa są zmienne i stałe odpowiedniego typu. Jeśli wyrażenie jest zmienną lub stałą to liść ten jest też korzeniem drzewa. W przeciwnym przypadku wyrażenie ma postać $(\omega_1 \odot \omega_2)$. Korzeniem drzewa jest wtedy operator $\odot$. Lewym poddrzewem jest drzewo odpowiadające wyrażeniu ω_1.

4.4. Wyrażenia znakowe. W Loglanie wartością wyrażenia znakowego jest znak. W tej książce i w języku Loglan na znakach nie wykonuje się żadnych operacji. Bądź przygotowany na to, że w innych językach programowania możesz napotkać wiele działań na znakach.

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

5. Komputer $\mathcal{K}_1$

$$\mathcal{K}_0 \sqsubset \mathcal{K}_1 \sqsubset \mathcal{K}_2 \sqsubset \mathcal{K}_3 \sqsubset \mathcal{K}_4 \sqsubset \mathcal{K}_5 \sqsubset \mathcal{K}_6 \sqsubset \mathcal{K}_7 \sqsubset \mathcal{K}_8 \sqsubset \mathcal{K}_9 \sqsubset \mathcal{K}_{10}$$

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) cztery elementy: wartość w_2, operator $\circledcirc$, wartość w_1 i nawias otwierający, oblicz wartość $w = w_1 \circledcirc w_2$ i wpisz w na stos. Jeżeli czwartym elementem nie jest nawias otwierający to BŁĄD,
F: pobierz wartość w z wierzchołka stosu; jeśli stos jest niepusty to zgłoś Błąd w przeciwnym przypadku zwróć obliczoną wartość wartość w.
Błąd błędy różnego rodzaju powinny być zgłaszane w odmienny sposób.

Twierdzenie 3.9. 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. teza indukcyjna? 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.
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

6. 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.

7. Aksjomaty typów pierwotnych

Testowanie czy dowodzenie?
Analiza programu dokonywana jest w odpowiedniej teorii. W języku $\mathcal{L}_1$ omawianym 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.

7.1. Aksjomaty algebry Boole'a. Wyrażenia Boolowskie, tj. formuły otwarte, można analizować posługując się semantyką opisaną powyżej. Jest jeszcze inna droga, można przekształcać wyrażenia Boolowskie w inne, równoważne wyrażenia posługując się, podanymi poniżej aksjomatami algebr Boole'a.

Poniżej podajemy zestaw aksjomatów.

(l_1) $\forall_{a,b\in B_0}\, a \cup b = b \cup a$ $\forall_{a,b\in B_0}\, a \cap b = b \cap a$

działania $\cup$ i $\cap$ są przemienne

(l_2) $\forall_{a,b,c\in B_0}\, a \cup (b \cup c) = (a \cup b) \cup c$ $\forall_{a,b,c\in B_0}\, a \cap (b \cap c) = (a \cap$

działania $\cup$ i $\cap$ są łączne

(l_3) $\forall_{a,b\in B_0}\, (a \cap b) \cup b = b$ $\forall_{a,b\in B_0}\, (a \cap (a \cup b) = a$

prawa pochłaniania, tj. prawa absorpcji

(l_4) $\forall_{a,b,c\in B_0}\, a \cap (b \cup c) = (a \cap b) \cup (a \cap c)$ $a \cup (b \cap c) = (a \cup b) \cap (a \cup$

prawa rozdzielności działań **$\cup$ i $\cap$ względem sieb**

(l_5) $\forall_{a,b\in B_0}\, (a \cap -a) \cup b = b$ $\forall_{a,b\in B_0}\, (a \cup -a) \cap b = b$

zero $(a \cap -a)$ i jedność $(a \cup -a)$

W językach programowania operacje boolowskie są oznaczane inaczej, zazwyczaj tak.

język	operacja				
Matematyka	$\cup$	$\cap$	$-$	$\top$	$\bot$
Matematyka2	$\vee$	$\wedge$	$\neg$	1	0
Loglan	or	and	not	true	false
C++, Java	||	&&	!	1	0

Uwaga. Oprócz operacji or oraz and w języku Loglan występują działania orif oraz andif. Więcej na ten temat napiszemy w rozdziale 5 o programowaniu elementarnym.

Koniec uwagi.

Czasami interesuje nas tylko odpowiedź na jedno pytanie: czy wartość wyrażenia boolowskiego α jest stale równa true niezależnie od wartościowania zmiennych występujących w tym wyrażeniu. W takim przypadku możemy posłużyć się poniższym zestawem aksjomatów - każde wyrażenie o schemacie ujętym w jeden z poniższych aksjomatów ma stale wartość true. Ponadto możemy posługiwać się regułą odrywania.

Jesli wyrazenia boolowskie α i $(\alpha \Rightarrow \beta)$ mają wartość stale równą true, to wyrazenie β ma także wartość równą true.

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

7.2. 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 $\mathbb{Z}$), 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
$$a + b = b + a \quad \textbf{oraz} \quad a \cdot b = b \cdot a,$$
- (Łączność) Dla każdej trójki liczb całkowitych a, b, c zachodzą
$$(a + (b + c)) = ((a + b) + c) \quad \textbf{oraz}$$
$$(a \cdot (b \cdot c)) = ((a \cdot b) \cdot c),$$
- (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
$$a + 0 = a \quad \textbf{oraz} \quad a \cdot 1 = a,$$
- (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,
$$\forall_{a\in\mathbb{Z}}\ a > 0 \vee a = 0 \vee a < 0$$
- (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 5 podamy algorytmiczny aksjomat liczb całkowitych, bardziej przydatny w analizowaniu algorytmów.

7.3. Aksjomaty liczb rzeczywistych. Do wnioskowania o własnościach programów wykonywanych w strukturze liczb rzeczywistych będziemy wykorzystywać aksjomaty ciała uporządkowanego Archimedesowskiego.

- (Przemienność dodawania)
$$\forall_{x,y\in R}\, x + y = y + x\ ,$$
- (Łączność dodawania)
$$\forall_{x,y,z\in R}\, (x + (y + z)) = ((x + y) + z)\ ,$$
- (Rozdzielność mnożenia względem dodawania)
$$\forall_{x,y,z\in R}\, (x + y) \cdot z = x \cdot c + y \cdot z$$
- (Jedność dodawania)
$$\exists_{0\in R}\forall_{x\in R}\, x + 0 = x,,$$
- (Addytywna odwrotność)
$$\forall_{x\in R}\exists_{-x\in R}\, x + -x = 0$$
- (Przemienność mnożenia)
$$\forall_{x,y\in R}\, x \cdot y = y \cdot x,$$
- (Łączność mnożenia)
$$\forall_{x,y,z\in R}\, (x \cdot (y \cdot z)) = ((x \cdot y) \cdot z),$$
- (Jedność mnożenia)

$$\exists_{1\in R}\forall_{x\in R}\, x\cdot 1 = x,$$

- (Odwrotność)
$$\forall_{x\in R}\, x\neq 0 \Rightarrow \exists_{x^{-1}\in R}\, x\cdot x^{-1} = 1$$
- (Porządek w zbiorze R)
$$\forall_{x\in R}\, \neg(x<x)$$
$$\forall_{x,y,z\in R}\, ((x<y\wedge y<z)\Rightarrow x<z)$$
$$\forall x,y(x<y\vee x=y\vee y<x)$$
- (aksjomat Archimedesa) dla każdej pary liczb dodatnich $0<y, y<x$ istnieje liczba całkowita, dodatnia k taka, że $x<k*y$.
$$\forall_{y\in R}\forall_{x\in R}(0<y<x\Rightarrow \exists_{k\in N}\, x<k\cdot y)$$

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 $x<k*y$.

> Nota 2. Zauważmy, że aksjomaty, będące równościami, możemy przedstawić graficznie jako równoważności drzew.

7.4. 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 każdy element zbioru C jest różny od pozostałych.

7.5. Aksjomaty typu pierwotnego string. W języku Loglan'82 nie występuje operacja konkatenacji na tekstach. Ale instrukcje drukowania właśnie tę operację realizują, dopisując kolejne znaki i teksty do poprzednio wydrukowanych. Natomiast w wielu innych językach programowania konkatenacja napisów jest

operacją języka wraz z licznymi innymi operacjami na napisach.

7.5.1. 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 zawiera operator konkatenacji (oznaczany *), stałe (tj. atomy) i równość.

$*\colon S \times S \to S$

$=\colon S \times S \to B_0$

Atomami są wszystkie znaki ze zbioru Char.

Aksjomaty

(A1) $$x * (y * z) = (x * y) * z$$

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

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

(A4) $$\neg('a' = x * y)$$

(A5) $$\neg('9' = x * y)$$

(A6) $$('B' \neq '7')$$

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 objaśniał ten aksjomat następująco: 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.
Profesor Andrzej Grzegorczyk [Grz05] wykazał, że w takiej teorii (nawet z dwoma tylko atomami) można reprezentować każdą funkcję programowalną.
Stąd stosując rozumowanie przekątniowe można wyprowadzić następujące twierdzenie:

Twierdzenie 3.10. Teoria konkatenacji jest nierozstrzygalna.

Inaczej mówiąc, posługując się tylko operacją konkatenacji i kwantyfikatorami możemy zbudować formuły na tyle skomplikowane, że nie istnieje jeden ogólny algorytm rozstrzygania ich prawdziwości.
W językach Java i C++ operacja konkatenacji na tekstach jest dopuszczalna. Z dowodu przeprowadzonego przez Grzegorczyka wynika, że nawet po odrzuceniu pozostałych typów w języku można wyrazić (zaprogramować) dowolną funkcję obliczalną. Miłośników języka Snobol[] ta wiadomość zapewne

ucieszy. Rozdział ten kończymy krótkim wprowadzeniem w tworzenie najprostszych programów. Umiejętność drukowania napisów i liczb jest niezbędna dla przeprowadzania własnych eksperymentów z programami.

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

Język $\mathcal{L}_0$.

$\mathcal{L}_0 \subsetneq \mathcal{L}_1 \subsetneq \mathcal{L}_2 \subsetneq \mathcal{L}_3 \subsetneq \mathcal{L}_4 \subsetneq \mathcal{L}_5 \subsetneq \mathcal{L}_6 \subsetneq \mathcal{L}_7 \subsetneq \mathcal{L}_8 \subsetneq \mathcal{L}_9 \subsetneq \mathcal{L}_{10}$

W programach 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. Odpowiednio zwiekszać się będzie zbiór instrukcji drukowania.

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(argumenty);writeln

możesz zastąpić przez

writeln(argumenty).

Przykłady

```
program druk1;
begin
  writeln("Witaj!")
  writeln;
  writeln("01234567890123456789012345678901");
  write(3.1415926:12:7);write(' ');write('a'); writeln(3.1415926:12:3);
  writeln("01234567890123456789012345678901");
  writeln("quick brown fox jumps over the leazy dog")
end
```

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.
Komputer wyposażony jest w pamięć programu i ekran (lub drukarkę).
Drukowanie na ekranie przebiega nieco inaczej niz na drukarce(papierze). Ta różnica nie powinna sprawiać Ci kłopotu. RYSUNEK komputera K0
Wykonanie instrukcji writeln spowoduje przejście do nowej linii. Wykonanie instrukcji write('c') polega na wypisaniu znaku 'c' na ekranie.

Zaawansowane drukowanie. Możesz wzbogacić repertuar drukowanych znaków. Urządzenia drukujące zdolne sa do drukowania znaków o różnym kroju (ang. fonts).
Można także kierować drukowanie do pliku i przekazywac takie pliki do późniejszego odczytywania.

Drukowanie wzbogacone. Drukowanie na ekranie komputera możemy urozmaicić stosując tzw. kody ucieczki (ang. ESC – escape). Znak ESC ma numer 27 stąd polecenie drukowania znaku chr(27).
Polecenie drukowania znaków półgrubych (ang. bold)

```
write( chr(27), "[1m")
```

Kolejne znaki pojawiające się na ekranie będą 'tłuściejsze' - półgrube.

Polecenie drukowania znaków pochylonych - kursywa

```
write( chr(27),"[3m")
```

Kolejne znaki pojawiające sie na ekranie będą drukowane pismem pochylonym.
Polecenie drukowania znaków normalnie

```
write( chr(27), "[0m")
```

Kolejne znaki pojawiające się na ekranie będą drukowane normalnie.
Możesz też próbować wydrukować tekst migający (ang. blinking)

```
write( chr(27), "[5m")
```

lub odwracając kolory tła i znaku (ang. reverse)

```
write( chr(27), "[7m")
```

Drukowanie znaków z podkreśleniem (ang. underscore) nastąpi po wydaniu takiego polecenia:

```
write( chr(27), "[4m").
```

Możesz drukować znaki kolorowe:

```
write( chr(27), "[32m"); na zielono
write( chr(27), "[31m"); na czerwono
write( chr(27), "[30m"); na czarno
write( chr(27), "[33m"); na żółto
write( chr(27), "[34m"); na niebiesko
write( chr(27), "[35m"); magenta
write( chr(27), "[36m"); cyan
write( chr(27), "[37m"); na biało
```

lub na kolorowym tle

```
write( chr(27), "[40m"); na czarnym tle
write( chr(27), "[41m"); na czerwonym tle
write( chr(27), "[42m"); na zielonym tle
write( chr(27), "[43m"); na żółtym tle
write( chr(27), "[44m"); na niebieskim tle
write( chr(27), "[45m"); tło magenta
write( chr(27), "[46m"); tło cyan
write( chr(27), "[47m"); na białym tle
```

Polecenia sterujące położeniem kursora na ekranie

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. Ten podrozdział możesz na razie opuścić.
Program może posługiwac sie plikami. W tymm miejscu omawiamy pliki tekstowe. Inne rodzaje plików są opisane gdzie indziej [SW91].
Deklaracja plików 'wewnętrznych' programu ...

... ?

Ćwiczenia

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?

3.2. Czy wyrażenia $(\frac{a+r^2}{a-r^2} + \frac{a-r^3}{a+r^2})$ i $(\frac{2a+r^2-r^3}{a-r^2+a+r^2})$ są równej wartości?

3.3. 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. Co można powiedzieć o wartościach wyrażeń?

$$(\alpha \vee \beta) \text{ i } (\beta \vee \alpha)$$

3.5. Zmodyfikuj komputer $\mathcal{K}_1$ tak by ...

3.6. Zauważyłaś, że gramatyka wyrażeń przyjęta w tym rozdziale jest bardzo restrykcyjna i wymaga by każde działanie było ujęte w nawiasy według wzoru $(arg_1\ op\ arg_2)$. Zmodyfikuj składnię wyrażeń całkowitoliczbowych tak by programista nie musiał wypisywać wszystkich nawiasów. Uzasadnij swoją propozycję.

3.7. W tym zadaniu posługujemy się gramatyką wyrażeń całkowito-liczbowych przyjętą w poprzednim ćwiczeniu 3.6. Zmodyfikuj komputer $\mathcal{K}_1$ tak by akceptował wyrażenia poprawne według Ciebie.

3.8. Zmodyfikuj komputer $\mathcal{K}_1$ tak by ...

ROZDZIAŁ 4

$\mathcal{L}_2$ Programy liniowe

$$\mathcal{L}_0 \subsetneq \mathcal{L}_1 \subsetneq \mathcal{L}_2 \subsetneq \mathcal{L}_3 \subsetneq \mathcal{L}_4 \subsetneq \mathcal{L}_5 \subsetneq \mathcal{L}_6 \subsetneq \mathcal{L}_7 \subsetneq \mathcal{L}_8 \subsetneq \mathcal{L}_9 \subsetneq \mathcal{L}_{10}$$

W tym rozdziale po raz pierwszy pojawiają się algorytmy i ich obliczenia.
Programy liniowe, jakie będziemy rozpatrywać w tym rozdziale są bardzo ważną częścią każdego algorytmu. Wiele programów można utożsamiać z grafami, w których występują wierzchołki dwojakiego rodzaju: testy (czyli wyrażenia boolowskie) oraz bloki instrukcji przypisania, zob. [Flo67, GMP71]. Wszystkie programy omawiane w tym i następnych trzech rozdziałach mogą być tak właśnie przedstawiane.Program liniowy jest ciągiem instrukcji przypisania. program napisany w assemblerze jest zbiorem programów liniowych oddzielonych instrukcjami skoku. Kompilacja programu napisanego w języku wyższego poziomu zwraca własnie taki program. Warto więc przyjrzeć się programom liniowym i dostrzec problemy oferowane przez ten rodzaj programów. Najpierw musimy zrozumieć jak to działa, czyli trzeba zroumieć semantykę tych programów. Wydaje się to proste, ale z programami tej postaci wiąże się wiele niełatwych problemów.

1. Przykład programu i jego obliczenia

Na początek, omawiamy prosty przykład programu liniowego. Naszym celem jest wyrobienie pewnych intuicji. Na ile one są trafne zobaczymy w kolejnych podrozdziałach. Rozdział ten rozpoczynamy przykładem w którym pojawiają się następujące pojęcia:

- instrukcja przypisania, program liniowy – po stronie składni,

- obliczenie programu, zmiana stanu pamięci – po stronie semantyki.

Przykładowy program. Rozpatrzmy poniższy program $L1$

```
program L1;
  const a= 1,b= -3,c= 5,d= 11 ;
  var Aa,l: integer;
  var e,f, g: integer
begin
  e:= a *c;
  f:= b*d;
  g:= (a+b)*(c+d);
  Aa:= e-f;
  l:= g-e-f
end
```

Przykład obliczenia. Poniżej przedstawiamy sześć kolejnych stanów obliczenia programu $L1$. Kazdy stan tj. konfigurację rekordu aktywacji programu przedstawiono jako tabelkę. Na zielonym tle widać instrukcje jakie pozostaja do wykonania. Na tle bladoróżowym są instrukcje już wykonane, można o nich zapomnieć. W tabeli Pamięci na czerwonym tle zaznaczono wynik ostatniej wykonanej instrukcji (przypisania).

Stan początkowy rekordu aktywacji

Pamięć

a	b	c	d	Aa	I	e	f	g
1	-3	5	11	0	0	0	0	0

Ciąg instrukcji

e:= a *c;
f:= b*d;
g:= (a+b)*(c+d);
Aa:= e-f;
I:= g-e-f

Po pierwszej instrukcji

Pamięć

a	b	c	d	Aa	I	e	f	g
1	-3	5	11	0	0	5	0	0

Ciąg instrukcji

e:= a *c;
f:= b*d;
g:= (a+b)*(c+d);
Aa:= e -f;
I:= g -e-f

Po wykonaniu dwóch instrukcji

Pamięć

a	b	c	d	Aa	I	e	f	g
1	-3	5	11	0	0	5	-33	0

Ciąg instrukcji

e:= a *c;
f:= b*d;
g:= (a+b)*(c+d);
Aa:= e -f;
I:= g -e-f

Po wykonaniu trzech instrukcji

Pamięć

a	b	c	d	Aa	I	e	f	g
1	-3	5	11	0	0	5	-33	-32

Ciąg instrukcji

e:= a *c;
f:= b*d;
g:= (a+b)*(c+d);
Aa:= e -f;
I:= g -e-f

Po czterech instrukcjach

Pamięć

a	b	c	d	Aa	I	e	f	g
1	-3	11	5	38	0	5	-33	-32

Ciąg instrukcji

e:= a *c;
f:= b*d;
g:= (a+b)*(c+d);
Aa:= e -f;
I:= g -e-f

Po pięciu instrukcjach

Pamięć

a	b	c	d	Aa	I	e	f	g
1	-3	11	5	38	-4	5	-33	-32

Ciąg instrukcji

e:= a *c;
f:= b*d;
g:= (a+b)*(c+d);
Aa:= e -f;
I:= g -e-f

I to jest koniec obliczenia tego programu. 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.
Zamiast przepisywać nasz program na nowo, zmieniając wartości stałych a,b,c,d, można napisać cztery instrukcje read(a); read (b); read(c); read(d). Trzeba też zadeklarować wielkości a,b,c,d jako zmienne, typu integer lub real.
Pamiętaj wyniki programu nie zostaną Ci ujawnione jeśli nie napiszesz instrukcji write.

Dodaj instrukcje read przed instrukcjami przypisania i instrukcję write(A,B) na końcu programu. Możemy zignorować wartości zmiennych e, f, g.

Czy można zaobserwować jakąś ogólną prawidłowość dotyczącą tego programu? Tak zmieniony program opisuje pewne odwzorowanie ze zbioru $\mathbb{Z}^4$ w zbiór $\mathbb{Z}^2$.

2. Składnia

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

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

Program w języku $\mathcal{L}_2$ ma budowę zgodną z przyjętą wcześniej definicją 2.1. Instrukcją programu jest (oprócz instrukcji drukowania i wczytywania) instrukcja przypisania. Potrzebna jest jeszcze relacja zgodności typów zmiennej i wyrażenia. Ta definicja dalszym ciągu będzie uzupełniana i modyfikowana. Na razie wystarczy nam takie jej sformułowanie.

Definicja 4.1. Niech dane będą zmienna z typu prostego T i wyrażenie τ typu prostego T'. Typy zmiennej z i wyrażenia τ są zgodne gdy są równe, $T = T'$. [1]

Definicja 4.2. Niech z będzie zmienną i niech τ będzie wyrażeniem. Zakładamy, że typy zmiennej i wyrażenia są zgodne.
Instrukcją przypisania jest napis postaci

$$z \leftarrow \tau \ .$$

> Uwaga. Symbol operacji przypisania $\leftarrow$ pojawił się tu nie przypadkiem. Wprowadzenie tego symbolu pozwala odsunąć w czasie dyskusję, która ortografia jest lepsza ta z Algolu i Pascala tzn. $z := \tau$, czy

[1] Pojęcie zgodności typów zostanie zmodyfikowane kolejno, w rozdziałach 6 i ??.

też dominująca dziś (zob. C++, Java i in.) notacja w której równość = jest wykorzystywana jako operator przypisania. Notacja $z = \tau$, 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 tej książce, w przykładach może pojawić się symbol :=, zamiennie z preferowanym przez nas, znakiem $\leftarrow$. □

Przykład 4.1. Który z poniższych napisów nie jest instrukcją przypisania?
z := a*b+b*b-3*a*b+a*a

Zmienne a,b,z powinny być tego samego typu integer lub typu real. [2]
u := v<u and v or u – ten napis nie jest instrukcją przypisania, dlaczego?
y:= x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x

Definicja 4.3. Instrukcją czytania read jest napis postaci

$$read(x)$$

gdzie x jest zmienną typu integer, real, znak lub string.

3. Semantyka

Znaczeniem instrukcji przypisania s jest funkcja $s_{\mathfrak{A}}$ odwzorowująca zbiór $\mathcal{W}$ stanów pamięci w ten sam zbiór.
zob. R.2

Definicja 4.4. Stan pamięci lub wartościowanie zmiennych to odwzorowanie przypisujęce zmiennym zadeklarowanym w programie wartości określonego typu. Wartością zmiennej boolowskiej jest jedna z dwu stałych logicznych 0 lub 1. Wartością zmiennej typu integer jest liczba całkowita, wartością zmiennej typu real jest liczba rzeczywista. Wartością zmiennej typu string jest tekst. Wartością zmiennej typu znakowego (char) jest znak.

[2] Jak to zobaczymy dalej jedna z tych trzech zmiennych może być typu real.

Będziemy pisać

$$v\colon V \to U$$

przykłady powyżej

Definicja 4.5. Instrukcja przypisania s ma postać $z \leftarrow \tau$, znaczeniem tej instrukcji jest funkcja, która stanowi pamięci v przypisuje stan pamięci v' w taki sposób

$$v'(x) \stackrel{df}{=} \begin{cases} \tau_{\mathfrak{A}}(v) & \text{gdy } x = z, \\ v(x) & \text{w przeciwnym przypadku.} \end{cases}$$

Półgrupa Instrukcje przypisania tworzą półgrupę ...ROZWINĄĆ!

4. Komputer $\mathcal{K}_2$ i pojęcie obliczenia

Komputer $\mathcal{K}_2$ umie nie tylko obliczać wartości wyrażeń, ale także potrafi obliczoną wartość przypisać zmiennej jako jej nową wartość. Pamięcią komputera $\mathcal{K}_2$ jest zbiór V zmiennych wyznaczony przez ciąg deklaracji programu.
Stanem pamięci, oznaczamy go przez v, jest odwzorowanie przypisujące każdej zmiennej $x \in V$ pewną wartość w zbiorze wartości typów pierwotnych.

Rozważmy instrukcję przypisania w jej ogólnej postaci $x := \tau$. Instrukcje programu wykonywane są po kolei i zmieniają stan pamięci (tzn. wartościowanie zmiennych).

Niech v oznacza wartościowanie zmiennych. Niech s oznacza skończony ciąg instrukcji atomowych $s_1, s_2, \ldots s_k$.

Definicja 4.6. Stanem obliczenia, lub konfiguracją, nazywamy parę uporządkowana $c = \langle v, \mathrm{s} \rangle$

Niech P będzie programem

```
program P;
  D        (* ciąg deklaracji *)
begin
  I        (* ciąg instrukcji atomowych *)
end
```

Definicja 4.7. Obliczeniem programu P jest ciąg stanów $c_0, c_1, ..., c_n$ taki, że

(1) stan c_0 jest parą $\langle v_1, I\rangle$ gdzie wartościowanie v_1 jest wartościowaniem początkowym wyznaczonym przez deklaracje D programu, ciąg I jest ciągiem wszystkich instrukcji (atomowych) programu P.

(2) jeśli w danym stanie $c_i = \langle v_i, i, i_2, ..., i_k\rangle$ ciąg instrukcji jest niepusty to do obliczenia należy stan c_{i+1} spełniający warunek
- jeśli pierwszą instrukcją i jest instrukcja przypisania $x \leftarrow \tau$, to następny stan jest parą $\langle i_{\mathbb{A}}(v_i), i_2, ..., i_k\rangle$
- jeśli pierwszą instrukcją jest instrukcja drukowania write(...) to następny stan jest parą $\langle v_i, i_2, ..., i_k\rangle$ (Na wyjściu dopisano to co trzeba).
- jeśli pierwszą instrukcją jest instrukcja read(z) to następny stan jest parą $\langle v_{i+1}, i_2, ..., i_k\rangle$. Wartościowanie v_{i+1} jest (nieformalnie) określone określone następująco:

$$v_{i+1}(x) = \begin{cases} v_i(x) & \text{gdy } x \neq z \\ l & \text{w.p.p., gdzie liczba l jest wczytana} \end{cases}$$

(3) jeśli w danym stanie $c_i = \langle v_i, \emptyset\rangle$ ciąg instrukcji jest pusty to stan ten jest końcowym stanem obliczenia programu P.

Przykład obliczenia znajduje sie na początku tego rozdziału.
Nietrudno zauważyć, że obliczenie każdego programu liniowego jest skończone. Tym niemniej, wraz z programami o tak prostej budowie pojawiają się pytania, na które czasami trudno odpowiedzieć:

- czy dany program P obliczy i wydrukuje wartość wyrażenia ω? np. czy po wykonaniu instrukcji

  z := a*b+b*b-3*a*b+a*a

  obliczymy wartość wyrażenia $(a+b)^2$?
- jak uzasadnić naszą odpowiedź i przekonać do niej innych?
- czy potrafimy podać najlepszy sposób obliczenia wartości wyrażenia ω? np. czy to jest

najlepszy sposób obliczenia potęgi x^{17}?
y:= x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x

- czy potrafimy poprawnie wskazać które zmienne i które instrukcje programu liniowego są niezbędne?
- i wiele podobnych pytań.

5. Prawa rachunku programów

Jak widać nie wystarczy opanować umiejętność napisania programu liniowego. Potrzebne nam są narzędzia do przeprowadzania analizy. W procesie badania własności programów liniowych pomocny będzie następujący schemat aksjomatu instrukcji przypisania

$$\{x \leftarrow \tau\}\alpha(x) \Leftrightarrow \alpha(x/\tau) \qquad Ax_{18}$$

Jak należy czytać powyższy schemat? Jest to tylko matematyczna stenografia. Po prawej stronie mamy wyrażenie boolowskie, zob. poniżej. Po lewej stronie mamy formułę logiczną zbudowaną według wzorca: <program> <formuła>. Jest to formuła języka rachunku programów. Napisy tej postaci będą nam odtąd towarzyszyć.
Na napis Ax_{18} składają się cztery elementy: instrukcja przypisania "x:=τ", formuła oznaczona literą α, znak równoważności formuł logicznych $\Leftrightarrow$ i napis $\alpha(x/\tau)$. Napis $\alpha(x/\tau)$ oznacza wyrażenie powstające z formuły $\alpha(x)$ gdy równocześnie zastąpimy wszystkie wolne wystąpienia zmiennej x w formule α przez term τ.

Przykład 4.2. Niech α będzie formułą $(y-j) < 7*y+c \vee 10 = (x+y)$ i niech napis τ będzie termem $(x+y*j)$. Wtedy $\alpha(y/(x+y*j)) = ((x+y*j)-j) < 7*(x+y*j)+c \vee 10 = (x+(x+y*j))$

Nietrudno zauważyć, że napis $\alpha(x/\tau)$ jest formułą. Można to udowodnić.

Lemat 4.1. Niech $\alpha(x)$ będzie dowolnym wyrażeniem boolowskim, niech τ będzie dowolnym wyrażeniem typu real.
Napis $\alpha(x/\tau)$ jest wyrażeniem boolowskim.

Dowód. Jeśli w wyrażeniu boolowskim elementarnym postaci $\tau_1 < \tau_2$ wsystkie wystąpienia zmiennej x zastąpimy przez term τ to napis $\tau_1(x/\tau) < \tau_2(x/\tau)$ jest wyrazeniem boolowskim.
Załóżmy zatem, że wyrażenie $\alpha(x)$ jest postaci $\alpha_1(x) \vee \alpha_2(x)$. Z załozenia indukcyjnego oba wyrażenia $\alpha_1(x/\tau)$ oraz $\alpha_2(x/\tau)$ są wyrażeniami boolowskimi. Wobec tego napis $\alpha_1(x/\tau) \vee \alpha_2(x/\tau)$ jest wyrażeniem boolowskim. Reszta dowodu przebiega podobnie. □

Możemy się wobec tego domyślać, że napis $\{x \leftarrow \tau\}\alpha(x)$ po lewej stronie znaku $\Leftrightarrow$ w aksjomacie Ax_{18} też jest formułą logiczną. I owszem, tak własnie jest – jest to przykład formuły algorytmicznej. Język rachunku programów zawiera takie formuły oprócz znanych nam formuł rachunku predykatów.

Całość formuły Ax_{18} czytamy:
dla każdego stanu pamięci v, następujące dwa warunki są równoważne:

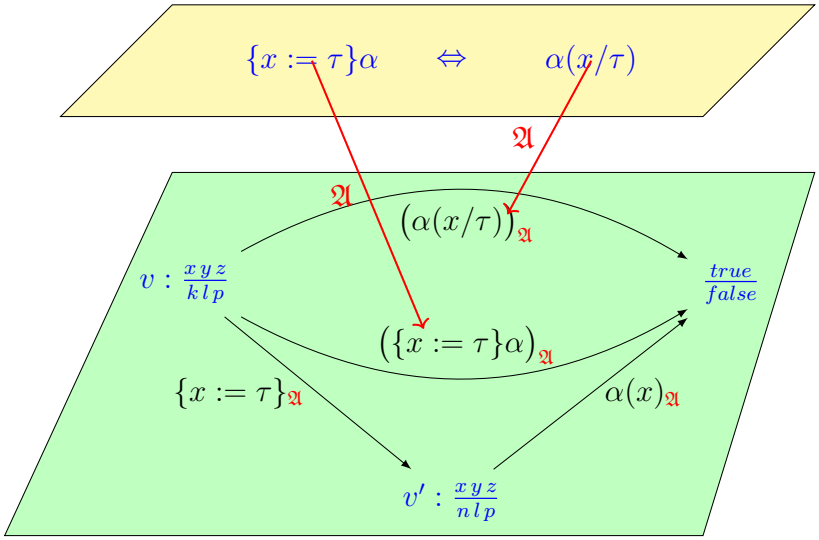
(ii) formuła $\alpha(x/\tau)$ jest spełniona przez stan pamięci v,

(i) formuła $\alpha(x)$ jest spełniona przez stan pamięci v' uzyskany po wykonaniu instrukcji przypisania $\mathsf{x}{:=}\tau$ w początkowym stanie pamięci v.

Przykład.
$\{y := x + 7\}(x + y < 2) \Leftrightarrow (x + x + 7 < 2)$
Sens tego aksjomatu można odczytać z rysunku 1. Aksjomat stwierdza, że diagram na tym rysunku jest przemienny. A więc, wartość formuły algorytmicznej $\{x := \tau\}\alpha(x)$, którą wyznaczamy

(1) obliczając wartość l wyrażenia τ po prawej stronie instrukcji przypisania,
(2) przypisując tę wartość zmiennej x , która wystepuje po lewej stronie instrukcji,
(3) obliczając wartość formuły $\alpha(x)$ następującej po programie, dla uzyskanego nowego stanu pamięci,

jest równa wartości formuły $\alpha(x/\tau)$ obliczonej dla początkowego stanu pamięci. dowód
Kolejnym prawem rachunku programów jest



Rysunek 1. Aksjomat instrukcji przypisania jest tautologią

na górnej płaszczyznie wydrukowaliśmy text aksjomatu, na dolnej płaszczyznie objaśniamy co się dzieje podczas obliczeń i wskazujemy znaczenie (tj. semantykę) aksjomatu, Litera $\mathfrak{A}$ oznacza dowolny typ (na razie przyjmij, że może to być typ integer lub typ real). Czerwone strzałki łączą formułę $\alpha(x/\tau)$ (odpowiednio {x:=τ}α) z funkcją $\alpha(x/\tau)_{\mathfrak{A}}$ (odpowiednio $\{x := \tau\}(\alpha)_{\mathfrak{A}}$). Stan pamięci v' i dodatkowe strzałki ilustrują sposób obliczania wartości formuły, $\{x := \tau\}(\alpha)$, zauważ, wartość $v'(x) = n = \tau_{\mathfrak{A}}(v(x))$.

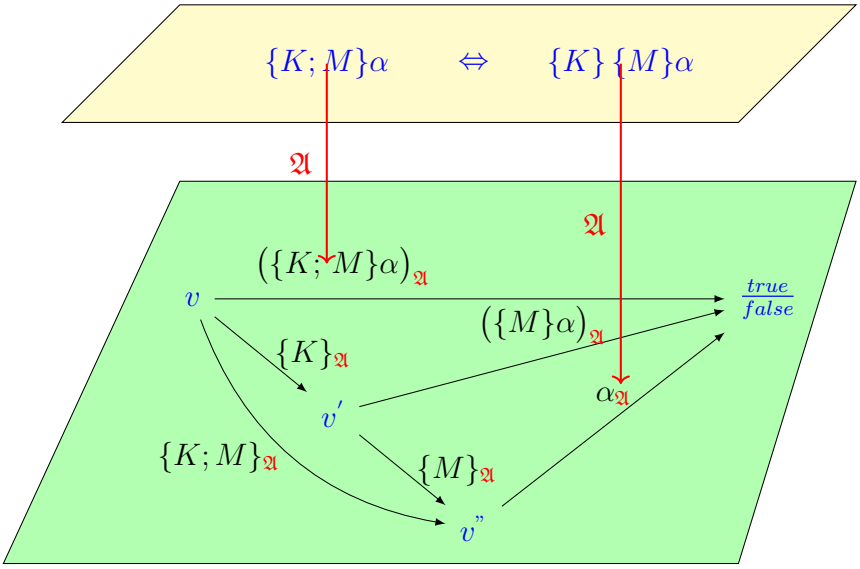
$$\{K; M\}\alpha \Leftrightarrow \{K\}\{M\}\alpha \qquad Ax_{19}$$

Każda formuła zbudowana zgodnie z tym schematem jest tautologią. Zobacz rysunek 2

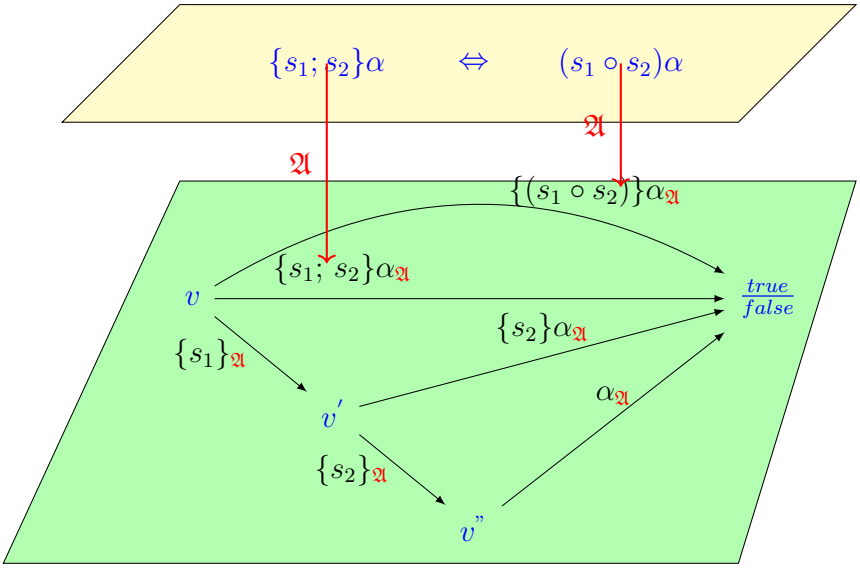
W przypadku gdy oba programy K oraz M są instrukcjami przypisania możemy wykorzystać operację składania takich instrukcji.Aksjomat Ax_{19} pyjmuje postać

$$\{s_1; s_2\}\,\alpha \Longleftrightarrow \{(s_1 \circ s_2)\}\alpha$$

Popatrzmy na rysunek 3 Aksjomaty te zastosujemy w poniższym dowodzie.



Rysunek 2. Aksjomat instrukcji złożonej jest tautologią



Rysunek 3. Operacja złożenia instrukcji przypisania

6. 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 prawdziwa jest poniższa implikacja?

$$(x = k \wedge y = l) \Rightarrow \{t := x; x := y; y := t\}(x = l \wedge y = k)$$

Zastosujemy aksjomat instrukcji przypisania ax18 i aksjomat instrukcji złożonej ax19.

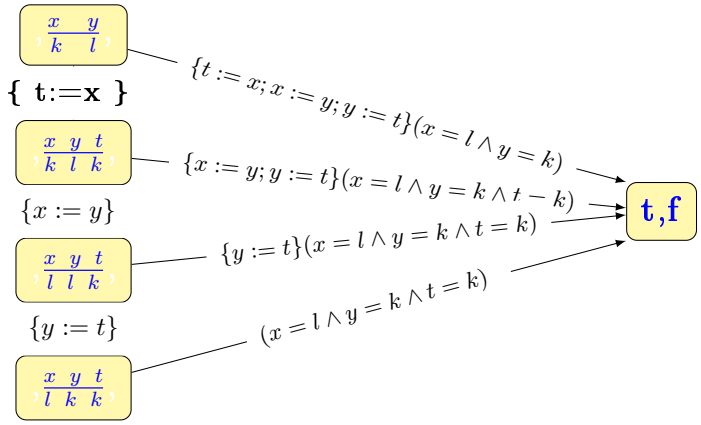
Nr	Formuła	Podstawa
(1)↓	$(x = k \wedge y = l) \Rightarrow \{t := x; x := y; y := t\}(x = l \wedge y = k)$	
(2)↓	$(x = k \wedge y = l) \Rightarrow \{t := x; x := y\}\{y := t\}(x = l \wedge y = k)$	≡1, ax19 ↑
(3)↓	$(x = k \wedge y = l) \Rightarrow \{t := x; x := y\}(x = l \wedge t = k)$	≡2, ax18 ↑
(4)↓	$(x = k \wedge y = l) \Rightarrow \{t := x\}\{x := y\}(x = l \wedge t = k)$	≡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 kazda z tych formuł jest tautologią.

Można także narysować diagram obliczenia wartości formuły

$$(x = k \wedge y = l) \Rightarrow \{t := x; x := y; y := t\}(x = l \wedge y = k).$$

Zobacz rysunek 4.. Rozpatrujemy takie wartościowania v, które spełniają poprzednik implikacji.
Zastanów się co różni te dwie analizy.
Które rozumowanie należy uznać za lepsze?
Oba są poprawne. Ta obserwacja jest zapowiedzią twierdzenia o pełności logiki algorytmicznej, zob. [MS87].
Zastosowanie. ...



Rysunek 4. Sprawdzenie tautologii rachunku programów
$(x = k \wedge y = l) \Rightarrow \{t := x; x := y; y := t\}(x = l \wedge y = k)$

Uogólnienie. To samo rozumowanie można powtórzyć gdy zmiennne x,y i t są innego typu T. 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 złożoną strukturę. W programach inżynierskich i naukowych zdarzają się instrukcje przypisania zajmujące 2 i więcej linii.
Natomiast komputer (lub maszyna wirtualna) ma ograniczony repertuar poleceń. Można powiedzieć, że są to instrukcje przypisania, w których występuje conajwyżej jeden operator. Dla przykładu, w niektórych komputerach starszych generacji mieliśmy do dyspozycji tylko takie instrukcje

polecenie	co robi
LOAD x	ACC:=x
ADD x	ACC := ACC+x
SUB x	ACC := ACC-x
MULT x	ACC :=ACC * x
DIV x	ACC := ACC / x
STORE x	x := ACC
...	...

W poleceniach mogą pojawiać się różne zmienne. Kompilator dodaje pomocnicze zmienne do tych zadeklarowanych w programie przez programistę.

Definicja 4.8. Instrukcja przypisania jest elementarna jeśli zawiera co najwyżej jeden operator.

Możemy teraz obejrzeć przykład ilustrujący zadanie kompilatora.

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

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

Twierdzenie 4.2. Niech z będzie zmienną typu pierwotnego T, a napis τ wyrażeniem tego samego typu T. Istnieje ciąg rlementarnych instrukcji przypisania s taki, że instrukcja $z := \tau$ jest równoważna ciągowi poleceń przypisania s.

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

W dalszych rozdziałach zobaczymy, że zbiór instrukcji przypisania może być powiększany. Programista może tworzyć nowe typy. Mogą się pojawiać nowe operatory oznaczjące nowe działania.
Powyższe twierdzenie pozostanie w mocy.

Wykorzystuj własności struktury danych. W poprzednich przykładach nie wykorzystano własności struktury danych (czyli typu) T. Jest to zaleta ponieważ nasze obserwacje mają uniwersalny charakter,

ale często wykorzystanie własności typu T w którym pracujemy mogą przynieć więcej korzyści. 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 4.3. Czy po wykonaniu programu P zachodzi warunek α?
tu napisać dość długi program, który daje się skrócić

Przykład 4.4. Wcześniej widzieliśmy przykład instrukcji przypisania
y:=x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x Można ją rozłożyć (zastąpić) ciągiem 17 elementarnych instrukcji przypisania. Ale można uzyskać ten sam efekt taniej

$$\left\{\begin{array}{l} \mathsf{z} := \mathsf{x}; \\ \mathsf{y} := \mathsf{x}; \\ \mathsf{z} := \mathsf{z} * \mathsf{z}; \\ \mathsf{z} := \mathsf{z} * \mathsf{z} : \\ \mathsf{z} := \mathsf{z} * \mathsf{z}; \\ \mathsf{z} := \mathsf{z} * \mathsf{z}; \\ \mathsf{z} := \mathsf{z} * \mathsf{y}; \end{array}\right\} (\mathsf{z} = \mathsf{x}^{17})$$

Wykorzystaliśmy prawo łączności mnożenia. Udało się zmniejszyć liczbę operacji mnożenia. Ten pomysł znajdzie jeszcze inne zastosowania.

Inny przykład

Przykład 4.5. An example of a straight-line program that computes the fast Fourier transform (FFT) on four inputs is given below. (The FFT is introduced in Section 6.7.3.) Here the function

Wykorzystuj prawa algebry Boole'a. W podobny sposób możemy wykorzystywac prawa algebry Boole'a.

Przykład 4.6. 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

$$(1)\ \Re \vdash \left\{ \begin{array}{l} \text{t1:=a+b;} \\ \text{t2:=c+d;} \\ \text{t3:=a*c;} \\ \text{t4:=b*d;} \\ \text{A:=t3 - t4;} \\ \text{B:=t1*t2-t3-t4;} \end{array} \right\} (A = a*c-b*d \wedge B = a*d+b*c)$$

W powyższym napisie $\Re$ oznacza zbiór aksjomatów typu real (zob. 7.3). Czyli formuł wyrażających łączność i przemienność dodawania, rozdzielnośc mnożenia względem dodawania, ...
Napis ten czytamy: z aksjomatów $\Re$ można udowodnić $\vdash$ formułę algorytmiczną (1)
Rzeczywiście, Formuła ta jest równoważna formule następującej

$$(2)\ \left\{ \begin{array}{l} \text{t1:=a+b;} \\ \text{t2:=c+d;} \\ \text{t3:=a*c;} \\ \text{t4:=b*d;} \\ \text{A:=t3 - t4;} \end{array} \right\} (A = a*c-b*d \wedge t1*t2-t3-t4 = a*d+b*c)$$

Ta z kolei jest równoważna formule
(3)

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

po kolei otrzymujemy
(4)

$$\left\{ \begin{array}{l} \text{t1:=a+b;} \\ \text{t2:=c+d;} \end{array} \right\} (a*c-b*d = a*c-b*d \wedge t1*t2-a*c-b*d = a*d+b*c)$$

pierwszy człon koniunkcji jest prawdziwy i nie stanowi problemu w dalszych rozważaniach (tj. możemy go pomijać).

$$(5)\quad \{\ \text{t1:=a+b;}\ \}\,(t1*(c+d)-a*c-b*d = a*d+b*c)$$

Pozostaje do sprawdzenia prawdziwość drugiego członu koniunkcji.

(6) $$((a+b)*(c+d)-a*c-b*d=a*d+b*c)$$

Stosujemy aksjomat rozdzielności mnożenia względem dodawania i skracamy wyrazy o przeciwnych znakach.
Przechodzimy od formuły (3.i) do równoważnej formuły (3.i+1). Każda taka równoważność jest twierdzeniem arytmetyki liczb typu real. Ostatnia formuła (6) jest twierdzeniem arytmetyki. Wobec tego pierwsza formuła (1) jest twierdzeniem arytmetyki liczb typu real.

Najsłabszy warunek wstępny. Zacznijmy od definicji. Niech napis K będzie programem,a β wyrażeniem boolowskim (formułą). Niech $\mathfrak{A}$ oznacza jeden z typów (pierwotnych lub zdefiniowanych przez pewną klasę C). Jak zobaczymy w dalszej części tej książki deklaracje klas umożliwiają wprowadzanie nowych typów. Pożyteczne będzie ...

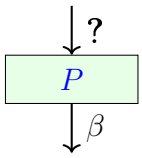
Definicja 4.9. Najsłabszym warunkiem wstępnym formuły β względem programu K nazywamy formułę α spełniającą następujące kryteria

(i) Dla dowolnych danych początkowych (tj. dla dowolnego wartościowania zmiennych programu), które spełniają formułę α, istnieją wyniki programu K (tj. obliczenie jest udane) i spełniają formułę β. Inaczej mówiąc, formuła α jest warunkiem wstępnym formuły β względem programu K.

(ii) Dla dowolnej formuły δ, jeżeli formuła δ jest warunkiem wstępnym formuły β względem programu K to implikacja $(\delta \Rightarrow \alpha)$ jest prawdziwa.

Zadanie znalezienia najsłabszego warunku wstępnego można zilustrować rysunkiem 5.

Gdy program K jest ciągiem instrukcji przypisania (abstrahujemy od instrukcji write i read), a formuła β jest wyrażeniem boolowskim, por. 3.2.2 str.29, to najsłabszy warunek wstępny α dla formuły β ze względu na program K można efektywnie wyznaczyć



Rysunek 5. Znaleźć najsłabszy warunek wstępny

posługując się aksjomatami instrukcji przypisania oraz instrukcji złożonej, rachunku programów Ax_{18} i Ax_{19}. Jeśli wykażemy, że formuła α jest równoważna najsłabszemu warunkowi wstępnemu formuły β ze względu na program K tj. $nww(\beta, K) \Leftrightarrow \alpha$ i ponadto wykażemy, że implikacja $(\alpha \Rightarrow \beta)$ ma dowód wyprowadzony z aksjomatów typu (pierwotnego) T tj. $T \vdash (\alpha \Rightarrow \beta)$ to możemy także uznać, że program K jest poprawny względem tej pary formuł.

$$\alpha \Rightarrow \{K\}(\beta)$$

Przykład 4.7. Jaki jest najsłabszy warunek wstępny dla formuły $z = (a+b)^2$ ze względu na program {z := a*b+b*b-3*a*b+a*a}? Odpowiedź: następująca formuła algorytmiczna

$$\{\mathsf{z := a*b+b*b-3*a*b+a*a}\}(z = (a+b)^2)$$

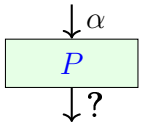
która jest równoważna formule (wyrażeniu boolowskiemu)

$$(a-b)^2 = (a+b)^2$$

czyli $ab = 0$, co jest równoważne formule $(a = 0 \vee b = 0)$.

Najmocniejszy warunek końcowy – programu liniowego. Często stawiamy sobie pytanie: co jest najmocniejszym warunkiem końcowym pewnego programu P i warunku początkowego α?
Inaczej mówiąc, jeśli wiemy, że dane początkowe spełniają warunek początkowy α i wiemy, że obliczenie programu P dla tych danych jest udane, to co wiadomo o wynikach v' tego obliczenia?

Definicja 4.10. Najmocniejszym warunkiem końcowym warunku początkowego α ze względu na program P, jest jest formuła β,która spełnia następujące warunki:



Rysunek 6. Znaleźć formułę β, która zastąpi znak '?' zapytania

(i) dla każdego wartościowania danych v (stanu poczatkowego), jeśli stan pamięci v spełnia warunek początkowy α i jesli obliczenie programu P jest udane i wartościowanie końcowe v' jest wynikiem, to formuła β jest spełniona przez wartościowanie v' (tj. formuła $beta$ jest warunkiem końcowym dla warunku początkowego α i programu P),
(ii) dla każdej formuły δ, jeśli formuła ta jest warunkiem końcowym dla warunku początkowego α i programu P, to implikacja $(\alpha \Rightarrow \delta)$ jest prawdziwa.

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

Przykład 4.8. Let us consider the program M of the form
x:= insert(x,y)
in the data structure of queues Q with the natural interpretation of the functor insert. We assume that x is a variable of type queue, and y a variable of type element of queue.
The weakest precondition of a formula $\neg empty(x)$ with respect to the program M is the formula $\neg empty(insert(x, y))$.
Observe, that in the data structure Q, the formula $\neg empty(insert(x, y))$ is valid under each valuation.
The situation is not always so simple. Let K be a program of the form:
while ¬first(x)=y $\wedge$ ¬empty(x)
do
x:=delete(x)
od

The weakest precondition of a formula α of the form
$(\neg empty(x) \ \wedge \ first(x) = y)$
in the structure of queues Q is the condition which states that y is an element of queue x. This condition can be written as the following infinite disjunction (which however does not belong to the language FOL) :
$((\neg empty(x) \wedge first(x) = y) \wedge$
$(\neg empty(delete(x)) \wedge first(delete(x)) = y) \wedge ...$
$(\neg empty(delete^{n-1}(x)) \wedge first(delete^{n-1}(x)) = y) \wedge ...)$
If the initial valuation is such, that for some n, $v(x)$ is an n-element queue and $v(y) \in v(x)$, then after removing at most $(n-1)$ first elements we obtain a queue with first element $v(y)$.

7. Półgrupa podstawień i sieci

W tym podrozdziale spojrzymy na instrukcje przypisania nieco inaczej. Wcześniej ?? widzieliśmy, że zbiór $\mathcal{W}_T$ wyrażeń danego typu $\mathcal{T}$ tworzy algebrę. Instrukcję przypisania s postaci $x \leftarrow \tau$ możemy pojmować jako funkcję ze zbioru zmiennych (typu T) w zbiór wyrażeń $\mathcal{W}_T$, tego samego typu $s\colon V_T \to \mathcal{W}_T$ taką, że

$$s(y) = \begin{cases} \tau & \text{gdy } y = x \\ y & \text{w pozostałych przypadkach} \end{cases}.$$

Instrukcje przypisania o tej własności nazywać będziemy elementarnymi. Odwzorowanie s daje się rozszerzyć ze zbioru zmiennych $V_\mathcal{T}$ na cały zbiór $\mathcal{W}_T$ wyrażeń typu $\mathcal{T}$. Wystarczy w dowolnym wyrażeniu ω wyznaczyć wszystkie wystąpienia zmiennej y i następnie każde z nich zastąpić wyrażeniem τ. Nowy napis powstały w ten sposób oznaczamy $\omega(y/\tau)$. Wiemy, por. ??, że jest to wyrażenie typu $\mathcal{T}$.

W niektórych językach programowania instrukcja przypisania może być instrukcją równoczesnego przypisania.

$$x_1 \leftarrow \tau_1 \& \ \ldots \ x_k \leftarrow \tau_k$$

Znaczenie takiej instrukcji jest wyznaczone w ten sposó, że najpierw należy wyznaczyć wartości $\tau_1(v), \ldots, \tau_k(v)$

wyrażeń po prawej stronie operatora przypisania, a następnie wartości te przypisać odpowiednim zmiennym $x_1, \ldots, x_k$.
W dalszym ciągu rozważać będziemy odwzorowania $s\colon V_T \to \mathcal{W}_T$. Napis $\omega(x_1/\tau, \ldots, x_k/\tau_k)$ oznaczać będzie wynik równoczesnego zastąpienia wszystkich wystąpień zmiennych $x_1, \ldots, x_k$ w wyrażeniu ω przez odpowiednie wyrażenia $\tau_1, \ldots, \tau_k$. Zbiór instrukcji przypisania tworzy półgrupę.
Niech s_1 i s_2 będą dwoma instrukcjami przypisania. Złożeniem jest instrukcja $s_1 \circ s_2$ określona w następujący sposób

$$(s_1 \circ s_2)(y) \stackrel{df}{=} s_1(s_2(y)) \quad \text{dla każdej zmiennej } y$$

czyli złozeniem dwóchinstrukcji przypisania równoczesnego

$s_1 : \quad x_1 \leftarrow \tau_1 \quad \ldots \quad x_k \leftarrow \tau_k$
$s_2 : \quad x_1 \leftarrow \nu_1 \quad \ldots \quad x_k \leftarrow \nu_k$

jest następująca instrukcja

$s_1 : \quad x_1 \leftarrow \tau_1(x_1/\nu_1 \ldots x_k/\nu_k) \quad \ldots \quad x_k \leftarrow \tau_k(x_1/\nu_1 \ldots x_k/\nu_k)$.

Jak nietrudno zauważyć działanie złożenia (czyli superpozycji) jest łączne.

$$((s_1 \circ s_2) \circ s_3) = (s_1 \circ (s_2 \circ s_3))$$

Jednością tej półgrupy jest instrukcja i przypisujące (każdej) zmiennej y wyrażenie y.

$$x_1 \leftarrow x_1 \& \ldots x_k \leftarrow x_k$$

Przypomnijmy, w każdym programie zbiór zadeklarowanych, tj. dostępnych, zmiennych jest skończony.

Przykład 4.9. Następujący ciąg elementarnych instrukcji przypisania

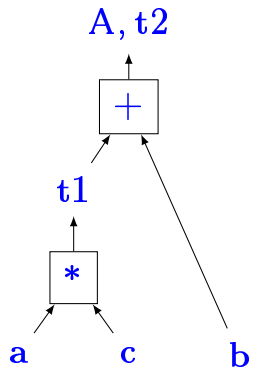
$t1 := a * c;$
$t2 := t1 + b;$
$A := t2;$

realizuje instrukcję {A :=a*c+b & t1:=a*c & t2:=a*c+b}. Ale ponadto, przypisuje zmiennym $t1$ i $t2$ wartości wyrażeń $a * c$ oraz $a * c + b$.

Można także uznać, że powyższy program jest równoważny jednej instrukcji przypisania równoczesnego

$$t1 := a * c \ \& \ t2 := a * c + b \ \& \ A := a * c + b.$$

Te trzy pojedyncze przypisania można przedstawić na poniższym diagramie



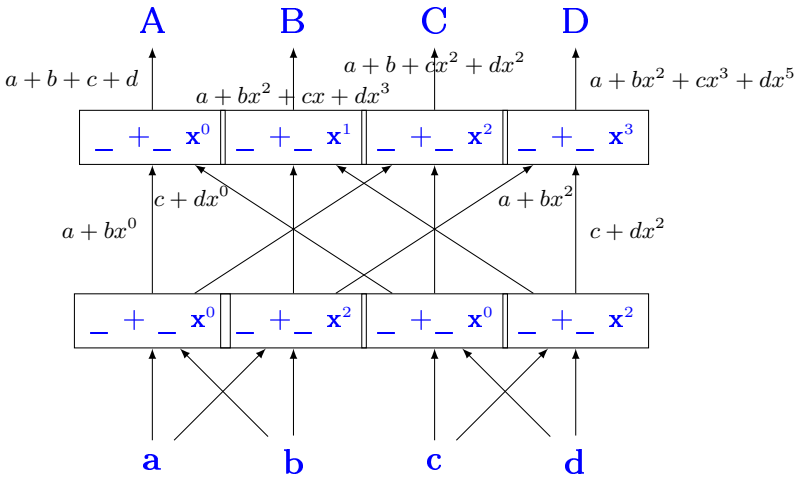
Warto przyjrzeć się nieco ciekawszemu przykładowi.

Przykład 4.10. (przygotowanie do FFT)
Mamy następujący program liniowy

(7)
```
A := a+b+c+d;
B := a+b*x*x+c*x+d*x*x*x;
C := a+b+c*x*x+d*x*x;
D := a+b*x*x+c*x*x*x+d*x*x*x*x*x;
```

Wykonanie powyższego programu wymaga 12 dodawań i 20 mnożeń.
Te same wyniki da następujący program liniowy.

(8)
```
t1 := a+bx^0;
t2 := c+dx^0;
t3 := a+b*x*x ;
t4 := c+d*x*x;
A := t1+t2x^0;
B := t3+t4*x;
C := t1 + t2*x*x;
D := t3 + t4*x*x*x;
```



Rysunek 7. Diagram programu 8
W prostokątach zapisano operacje – podkreślenia to miejsce na argumenty operacji. Zmienne a, b, c, d są argumentami programu, a wynikami są A, B, C, D.

W tym programie mamy 8 dodawań i 10 mnożeń. Mnożenia przez $\mathbf{x}^0$ można pominąć. Czy te programy są równoważne? To łatwo sprawdzić. Formuła

$$\left\{\begin{array}{l} t1 := a + b * x^0; \\ t2 := c + d * x^0; \\ t3 := a + b * x * x; \\ t4 := c + d * x * x; \\ A := t1 + t2 * x^0; \\ B := t3 + t4 * x; \\ C := t1 + t2 * x * x; \\ D := t3 + t4 * x * x * x; \end{array}\right\} \left(\begin{array}{l} A = a + b + c + d; \\ B = a + bx^2 + cx + dx^3; \\ C = a + b + cx^2 + dx^2; \\ D = a + bx^2 + cx^3 + dx^5; \end{array}\right)$$

jest twierdzeniem teorii pierścieni, a więc rzeczywiście programy te są równoważne.
Można jeszcze zmniejszyć liczbę mnożeń, ale nie to jest najważniejsze. W zastosowaniach (FFT) zamiast x^0, x^1, x^2, x^3 wystąpią stałe.

W dalszym ciągu będziemy rozważać takie właśnie instrukcje przypisania jednoczesnego. Zauważmy parę faktów:

- przypisanie identycznościowe x:=x jest jednością obustronną tej półgrupy, jest to więc monoid,
- każda instrukcja przypisania jednoczesnego daje się rozłożyć na iloczyn elementarnych instrukcji przypisania (do jednej zmiennej),
- Niech s_1 oznacza instrukcję $x := \tau$,i niech s_2 oznacza instrukcję $x := \omega$. Jeśli wyrażenie ω nie zawiera zmiennej x to złożenie $s_1 \circ s_2 = s_2$, czyli w tym przypadku instrukcja s_1 jest lewym zerem instrukcji s_2. Ta obserwacja ma pewne znaczenie.

Ogólniej, kompilator każdego języka programowania wykorzystuje podobne fakty w procesie ulepszania (tj. optymalizacji) skompilowanego programu. W podręcznikach dotyczących kompilacji a także w tekstach o algorytmice występuje pojęcie grafu acyklicznego skierowanego (ang. directed acyclic graph – dag) przypisanego programowi liniowemu.
Ponad sto lat temu zaobserwowano odpowiedniość pomiędzy wyrażeniami boolowskimi zbudowanymi (wyłącznie) ze zmiennych boolowskich a obwodami elektrycznymi łączonymi szeregowo bądź równolegle. Istnieje obszerna literatura tego przedmiotu.
Rola dag'ów uznana w środowisku twórców kompilatorów powoli jest też doceniana wśród osób zajmujących się algorytmiką. W książce Savage'a [Sav98] znajdujemy 1° spostrzeżenie, że pomiędzy programami liniowymi a grafami acyklicznymi skierowanymi istnieje a jednoznaczne odwzorowanie i 2° wiele ciekawych faktów pokazującycg jak to spostrzeżenie wykorzystać w celu obniżenia kosztu algorytmu. W podręcznikach i monografiach dotyczących budowy kompilatorów zob. [ALSU07, App98] pojęcie to jest wykorzystywane w rozdziałach dotyczących optymalizacji kodu wynikowego.

Sieć programu liniowego.
Istnieje wzajemnie jednoznaczna odpowiedniość pomiędzy zbiorem programów liniowych i zbiorem grafów cyklicznych zorientowanych (ang.dag).
Każdej (elementarnej) instrukcji przypisania postaci

$x \to \tau$ odpowiada graf(drzewo) określone w ten sposób

$$\Delta_\tau \to x$$

gdzie Δ_τ jest drzewem wyrażenia τ, zamiast strzałki w prawo możesz użyć strzałkę do góry.

$$\begin{array}{c} x \\ \uparrow \\ \Delta_\tau \end{array}$$

Instrukcji (jednoczesnego) przypisania $x_1 \leftarrow \tau_1 \& \ \ldots \ x_k \leftarrow \tau_k$ przypisujemy dag

$$\begin{array}{ccc} x_1 & \ldots & x_k \\ \uparrow & \ldots & \uparrow \\ \Delta_{\tau_1} & \ldots & \Delta_{\tau_1} \end{array}$$

Natomiast programowi liniowemu

$$\left\{ \begin{array}{l} x_1 \leftarrow \tau_1; \\ \ldots \\ x_2 \leftarrow \tau_2; \end{array} \right\}$$

przypisujemy graf

$$\begin{array}{cc} x_1 & x_2 \\ \uparrow & \uparrow \\ \Delta_{\tau_1} & \Delta_{\tau_2} \left(\begin{array}{c} x \\ \uparrow \\ \Delta_\tau \end{array} \right) \end{array}$$

Wyrażeniom można przyporządkować drzewa będące ich diagramami graficznymi, zob. poprzedni rozdział ??. Instrukcji i przypisania postaci $x := \tau$ przyporządkowujemy drzewo Δ_i w ten sposób, że do drzewa δ_τ dodajemy nowy wierzchołek, umieszczamy w nim zmienną x, synem tego wierzchołka będzie korzeń drzewa δ_τ. Wierzchołek ten będzie korzeniem drzewa Δ_i. Programom liniowym możemy przyporządkować sieci (inaczej dagi tj. grafy skierowane, acykliczne).

Definicja 4.11. Niech P będzie ciągiem instrukcji przypisania, tj. programem liniowym.

$$x_1 := \tau_1;\ x_2 := \tau_2;\ \ldots;\ x_k := \tau_k$$

Niech Δ_i będzie drzewem instrukcji $x_i := \tau_i$, i=1,. . . ,k. Siecią programu P jest graf G_P skonstruowany w k krokach w następujący sposób:

(1) na początek, j=k i graf G_P jest drzewem Δ_j instrukcji $x_j := \tau_j$,
(2) jeśli utworzono graf G_P wykorzystując ciąg $\{\Delta_i\}_{i=j+1}^{k}$ drzew instrukcji $x_{j+1} := \tau_{j+1}\,; \ldots ; x_k := \tau_k$ i $j \neq 1$ to do grafu G_P dołączamy drzewo Δ_j instrukcji $x_j := \tau_j$ i dodajemy krawędzie łączące korzeń x_j drzewa Δ_j z wszystkimi liśćmi grafu G_P które zawierają zmienną x_j.
(3) jeśli $j = 1$ to kończymy.

Przykład
Przykład

8. Co warto zapamiętać?

Instrukcja przypisania jest napisem postaci $x := \tau$, gdzie x jest zmienną a τ jest wyrażeniem. Każde wyrażenie jest zbudowane ze zmiennych, znaków działań (i relacji) oraz nawiasów. Znaczeniem instrukcji przypisania jest funkcja $(x := \tau)_{\mathfrak{A}}$ określona na zbiorze wartościowań zmiennych (czyli stanów pamięci) W. Wartościami tej funkcji są także stany pamięci

$$(x := \tau)_{\mathfrak{A}} \colon W \to W.$$

Program liniowy to ciąg instrukcji przypisania. Programista powinien mieć następujące umiejętności

- uzasadnić, ze dane dwa programy liniowe są równoważne lub podać kontrprzykład,
- uzasadnić, że po wykonaniu programu liniowego P końcowy stan pamięci będzie spełniać warunek zapisany jako wyrażenie boolowskie β'

W rozwiązywaniu takich zadań pomocne są: własności typu pierwotnego zob. poprzedni rozdział Wyrażenia, aksjomat instrukcji przypisania Ax_{18}, aksjomat instrukcji złożonej Ax_{19}.

Ćwiczenia

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?

4.2. Czy instrukcje przypisania równoczesnego tworza półgrupę?

4.3. Scharakteryzuj lewe zero przypisania s

4.4. Ilu mnożeń potrzeba by obliczyć iloczyn wielomianów (ax+b)*(cx+d)?

4.5. Zastanów się czy program SWAP może zamienić wartości zmiennych innego typu (boolean, real, char, string)?

4.6. Jak narysować program liniowy?

4.7. Trudne. Dla danego programu liniowego znajdź jego najszybszy odpowiednik. Optymalizacja

4.8. Udowodnij, że następujący program P oblicza poprawnie iloczyn dwu macierzy o wymiarze 2×2. Należy więc wykazać, że dla każdej ósemki liczb a, b, c, d, e, f, g, h, program P oblicza liczby A, B, C, D takie ,ze zachodzi równość

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

Program P

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

Należy więc wykazać, że prawdziwa jest formuła

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

czyli, udowodnić, poniższą formułę

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

4.9. Czy dla każdego programu liniowego P istnieje równoważny z nim program liniowy Q, taki że w programie Q wszystkie instrukcje read są na początku i wszystkie instrukcje write są na końcu tego programu?

4.10. Zastanów się, czy każdy program liniowy można przekształcić tak, by wszystkie instrukcje read poprzedzały każdą inną instrukcję programu i wszystkie instrukcje write wystąpiły na końcu programu?

4.11. Czy dag'i programumu liniowego $x_1 := \tau_1; \ldots x_k : \tau_k$ i instrukcji przypisania równoczesnego $x_1 := \tau_1 \,\&\, \ldots \,\&\, x_k := \tau_k$ są równe??

4.12. Udowodnij twierdzenie 4.2

ROZDZIAŁ 5

$\mathcal{L}_3$ Programy elementarne

$$\mathcal{L}_1 \subsetneq \mathcal{L}_2 \subsetneq \mathcal{L}_3 \subsetneq \mathcal{L}_4 \subsetneq \mathcal{L}_5 \subsetneq \mathcal{L}_6 \subsetneq \mathcal{L}_7 \subsetneq \mathcal{L}_8 \subsetneq \mathcal{L}_9 \subsetneq \mathcal{L}_{10}$$

Język programów elementarnych $\mathcal{L}_3$ ma te same deklaracje co język programów liniowych $\mathcal{L}_2$.

$$\mathbb{D}_3 = \mathbb{D}_2$$

Zbiór instrukcji atomowych jest także ten sam co w języku $\mathcal{L}_2$.

$$at_3 = at_2$$

Natomiast pojawiają się dwie nowe operacje na programach: if – operator tworzenia instrukcji warunkowej i for – operator tworzenia instrunkcji powtarzania.

Przykład 5.1. Czy zgadniesz co oblicza ten algorytm?

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

Wynik zależy od (początkowych) wartości zmiennych x i k. Program nie zmienia wartości tych zmiennych. Zmienne i, j oraz c są zmiennymi pomocniczymi. Co można powiedzieć o końcowej wartości zmiennej s?

1. Składnia

W poniższej definicji wprowadzamy nowe konstrukcje programotwórcze i zestawiamy je z tymi wcześniej zanymi.

Definicja 5.1. Napisy poprawnie poprawnie zbudowane różnice i podobieństwa:

- Zbiór $\mathcal{W}$ wyrażeń języka $\mathcal{L}_3$ jest taki sam jak języku $\mathcal{L}_2$ programów liniowych.

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

- 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.

$$at_3 = at_2$$

  Przypomnijmy, są to instrukcje przypisania oraz instrukcje czytania i drukowania.

- Zbiór instrukcji jest najmniejszym zbiorem napisów zawierającym instrukcje atomowe i zamkniętym ze względu na trzy operacje:
  p0) (złożenia) jeśli K i M są ciągami instrukcji to ich złożenie K, M jest instrukcją,
  p1) (rozgałęzienia) jeśli napis γ jest wyrażeniem boolowskim, a napisy K oraz M są skończonymi ciągami instrukcji to instrukcją jest także napis postaci

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

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

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

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

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

  jest instrukcją powtarzania.

2. Semantyka

Podobnie jak wcześniej, znaczenie programu jest opisane przez podanie definicji relacji bezpośredniego następstwa w zbiorze stanów tj. konfiguracji. Wystarczy opisać dwa nowe przypadki: Obliczenie programu to ciąg stanów $\{c_i\}_i$ taki, że $1^\circ c_0$ jest stanem początkowym rekordu aktywacji programu, 2°jeśli w stanie c_i ciąg instrukcji do wykonania jest niepusty i I oznacza pierwszą instrukcję tego ciągu, to następny stan c_{i+1} ...

Bezpośrednim następnikiem stanu

$$\langle v, \textbf{if } \gamma \textbf{ then } K \textbf{ else } M \textbf{ fi}; \textbf{s}\rangle \mapsto \begin{cases} \langle v, K; \textbf{s}\rangle & \textbf{gdy } \gamma(v) = \textbf{true} \\ \langle v, M; \textbf{s}\rangle & \textbf{gdy } \gamma(v) = \textbf{false} \end{cases}$$

Bezpośrednim następnikiem stanu

$$\langle v, \textbf{for } i := A \textbf{ to } C \textbf{ do } M \textbf{ od}; \textbf{s}\rangle \mapsto \langle v, \left\{ \begin{array}{l} i := A; \quad aux2 := C; \\ \textbf{if } i \leq aux2 \textbf{ then} \\ \quad M; \\ \quad \textbf{for } i := i+1 \textbf{ to } aux2 \\ \qquad \textbf{do } M \textbf{ od} \\ \textbf{fi}; \end{array} \right\}; \textbf{s}\rangle$$

Powyższa reguła przejścia od konfiguracji w której pierwszą do wykonania instrukcją jest instrukcja for może wydawać się dziwaczną. Jest to jednak precyzyjny opis zmiany stanu. Oczywiście twórca kompilatora i maszyny wirtualnej ma wiele możliwości by osiągnąć ten sam efekt, być może posługując się innymi środkami.

Łatwo sprawdzić, że obliczenie instrukcji for sprowadzi się do instrukcji złożonej $\{i := i + 1; M\}$ powtórzonej $C - A$ razy.

Lemat 5.1. Niech v będzie dowolnym wartościowaniem zmiennych, niech s będzie dowolnym ciągiem

instrukcji (z języka $\mathcal{L}_3$). Jeśli spełniona jest nierówność $C \geq A$, to następujące dwie konfiguracje są równoważne, tj. można jedną zastąpić przez drugą.

$$\langle v, \textbf{for } i := A \textbf{ to } C \textbf{ do } M \textbf{ od}; \mathbf{s}\rangle$$

$$\langle v, \{\ i := A; \quad aux2 := C\ \}; \underbrace{\left\{ \begin{array}{l} M; \\ i := i+1 \end{array} \right\}}_{(C+1-A)\times}; \mathbf{s}\rangle$$

3. Abstrakcyjna wirtualna maszyna $\mathcal{K}_3$

Wykonywanie każdego programu P z języka L_3 polega na wykonywaniu kolejnych instrukcji zapisanych w rekordzie aktywacji programu P.
Komputer $\mathcal{K}_3$ instrukcje if oraz for wykonuje w sposób opisany powyżej w definicji relacji bezpośredniego przejścia z jednego stanu do następnego. Tak jak to napisaliśmy w definicji obliczenia.

4. Aksjomaty

Co robi program elementarny? Jakie sa efekty jego działania?

Znaczenie instrukcji języka $\mathcal{L}_2$ określają następujące schematy aksjomatów

Znaczenie instrukcji for opisują trzy schematy aksjomatów:

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

f2) $(B = A) \Rightarrow (\{\textbf{for } i := A \textbf{ to } B \textbf{ do } I \textbf{ od}\}\alpha \Leftrightarrow (\{i := A;\ I;\ i := i+1\}\alpha))$

f3) $(\{\textbf{for } i := A \textbf{ to } B+1 \textbf{ do } I \textbf{ od}\}\alpha \Leftrightarrow$
$\{\textbf{for } i := A \textbf{ to } B \textbf{ do } I \textbf{ od};\ I;\ i := i+1\}\alpha)$
poniżej znajdziesz dwie, czasami przydatne, odmiany schematu f3).

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

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

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 }.

Uwaga 5.2. Zapewne dostrzegłeś, irytującą, instrukcję i:=i+1 dopisaną na końcu tych aksjomatów po prawej stronie równoważności. Jeżeli zmienna i nie występuje w formule α jako zmienna wolna, to nie ma problemu. Programista powinien jednak być świadom tego, że zakończenie instrukcji for – w Loglanie ma taki właśnie efekt. W innych językach programowania – Java, C++ jest inaczej. Zechciej to samemu zbadać.
W rozdziale o instrukcji bloku pokażemy jak sobie poradzić z tą niedogodnością.

Znaczenie instrukcji warunkowej opisują formuły o następującym schemacie

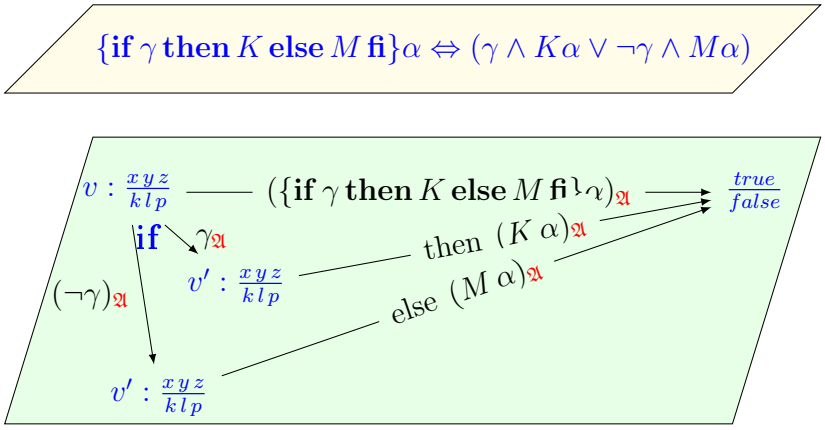
$$\{\text{if } \gamma \text{ then K else M fi }\}\alpha \Leftrightarrow ((\gamma \wedge \text{K}\,\alpha) \vee (\neg\gamma \wedge \text{M}\,\alpha))$$

w powyższym schemacie γ oznacza wyrażenie boolowskie, a wyrażenia K oraz M oznaczają ciągi instrukcji.
prove f1,f2, f3

5. Analiza programów

Lemat 5.3. Jeśli $A \leq B$, to po wykonaniu programu for i:= A to B do I od zachodzi $i = B+1$

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. Później jeszcze wartość zmiennej i jest zwiększona o 1 i wynosi B+1.
Załóżmy, że teza lematu jest prawdziwa dla $B = k$.



Rysunek 1. Aksjomat Ax_{20} instrukcji warunkowej jest tautologią
na górnej płaszczyznie wydrukowaliśmy text aksjomatu, na dolnej płaszczyznie objaśniamy co się dzieje podczas obliczeń i wskazujemy znaczenie (tj. semantykę) aksjomatu, Litera $\mathfrak{A}$ oznacza dowolny typ (na razie przyjmij, że może to być typ integer lub typ real).

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

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

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

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

Twierdzenie 5.5. 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

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

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

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

Zacznijmy od tautologii

$$s = \sum_{i=0}^{n} \tau(i) \Rightarrow s = \sum_{i=0}^{n} \tau(i) \tag{9}$$

Ta formuła jest równoważna następującej

$$s = \sum_{i=0}^{n} \tau(i) \Rightarrow s + \tau(n+1) = \sum_{i=0}^{n} \tau(i) + \tau(n+1) \tag{10}$$

Z definicji znaku Σ otzymujemy formułę

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

Stosując aksjomat instrukcji przypisania otrzymujemy

(12) $$s = \sum_{i=0}^{n} \tau(i) \Rightarrow \{i := n+1\} \left(s + \tau(i) = \sum_{i=0}^{n+1} \tau(i) \right)$$

Ponownie stosujemy ten sam aksjomat

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

Zastosujmy regułę wnioskowania $\boxed{\text{R2:} \dfrac{\alpha \Rightarrow \beta}{M\alpha \Rightarrow M\beta}}$ i aksjomat Ax_{20}

(14)

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

Stosując aksjomat (f3) instrukcji for, otrzymujemy, że dla każdej liczby naturalnej n prawdziwa jest implikacja

(15)

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

co kończy dowód kroku indukcyjnego.
Wobec tego, dla każdej liczby naturalnej n zachodzi

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

□

Własności programów z języka $\mathcal{L}_3$ c.d. Nie tylko suma jest obliczalna w języku $\mathcal{L}_3$. Zobaczmy, że można udowodnić, że także iloczyny, alternatywy i koniunkcje oraz kilka innych operacji daje się zaprogramować w języku $\mathcal{L}_3$.

Twierdzenie 5.6. Niech $\tau(i)$ będzie wyrażeniem arytmetycznym. Następujący program

$$\left\{\begin{array}{l} p := 1; \\ \textbf{for } i := 1 \textbf{ to } \mathbf{n} \\ \textbf{do} \\ \quad p := \tau(i) * p \\ \textbf{od} \end{array}\right\} \left(p = \prod_{i=1}^{\mathbf{n}} \tau(i) \right)$$

oblicza wartość iloczynu wielkości $\tau(1) * \tau(2) * \ \ldots \ * \tau(n)$.

Podobne spostrzeżenie odnosi się do kwantyfikatora ogólnego

Twierdzenie 5.7. 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

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

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

Przykład 5.2. Mamy napisać program obliczający $\sum_{i=1}^{k} i^2$.
Korzystając z twierdzenia 5.5 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 5.3. Należy obliczyć sumę $\sum_{i=1}^{k} \frac{x^i}{i!}$.

Zaczynamy od ponownego wykorzystania twierdzenia 5.5.

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

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 5.5. (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 5.8 i 5.9 przytoczonych poniżej.

Lemat 5.8. (Obliczanie potęgi)

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

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

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

Lemat 5.9. (Obliczanie silni)

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

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

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

Zechciej porównać koszty wykonania obu tych programów.

6. Kwantyfikator ograniczony

W dowodach poprawności przyda się nam następujące

Twierdzenie 5.10. (o wprowadzaniu instrukcji for przed kwantyfikator ograniczony)
Niech $P(i)$ będzie programem, niech $\alpha(i)$ będzie formułą.
Każda formuła zbudowana według następującego schematu jest twierdzeniem logiki algorytmicznej.

$$\vdash \left(\bigwedge_{i=1}^{n} \{P(i)\}\, \alpha(i) \wedge \bigwedge_{1 \le i < j \le n} \left(\{P(i)\}\, \{P(j)\} \alpha(i) \equiv \{P(i)\}\, \alpha(i) \right) \right)$$

$$\Rightarrow \left\{ \begin{array}{l} \textbf{for } i \leftarrow 1 \textbf{ to } n \\ \textbf{do} \\ \quad P(i) \\ \textbf{od} \end{array} \right\} \forall_{i=1}^{n}\ \alpha(i)$$

Dowód. napisać ten dowód □

Twierdzenie to znajduje wiele zastosowań w programowaniu z tablicami, zob. następny rozdział.
W naturalny sposób twierdzenie to uzasadnia stosowanie następującej reguły wnioskowania

$$\frac{\bigwedge_{i=1}^{n} \{P(i)\}\, \alpha(i)\ ,\ \bigwedge_{1 \le i < j \le n} \left(\{P(i)\}\, \{P(j)\} \alpha(i) \equiv \{P(i)\}\, \alpha(i) \right)}{\{\textbf{for } i \leftarrow 1 \textbf{ to } n \textbf{ do } P(i) \textbf{ od}\}\, \forall_{i=1}^{n}\ \alpha(i)}$$

7. Funkcje pierwotnie rekurencyjne

W latach 30 dwudziestego wieku zaistniała potrzeba zdefiniowania pojęcia funkcji obliczalnej. Naturalnymi kandydatami stały się funkcja następnika i funkcja zero. Zauważono, że jesli funkcje f ig są obliczalne, to obliczalne jest też złożenie (superpozyja) tych funkcji. Jeśli obliczalne są funkcje $g(x)$ oraz $h(n,x,y)$ to obliczalna jest funkcja $f(n,x)$ określona indukcyjnie następującymi wzorami

$$f(0,x) \stackrel{df}{=} g(x)$$

$$f(n+1,x) \stackrel{df}{=} h(n,x,f(n,x))$$

Lemat 5.11. Dla każdej pary liczb naturalnych n,x istnieje liczba naturalna y taka, że $f(n,x) = y$.
Jesli istnieją dwa różne obliczenia wartości wyrażenia $f(n,x)$ to otrzymane wyniki sa równe.

Definicja 5.2. Zbiór funkcji pierwotnie rekurencyjnych $Prek$ to najmniejszy zbiór funkcji zawierający następnik, zero oraz dwie funkcje $I(x,y) = x$ i $J(x,y) = y$ i zamknięty ze względu na superpozycję i rekursję prostą.

Jest oczywiste, że złożenie (superpozycja) funkcji pierwotnie rekurencyjnych jest funkcją pierwotnie rekurencyjną. W dowodzie tej własności wykorzystujemy aksjomat instrukcji złożonej Ax_{20}.
Pozostaje do wykazania, 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 dla kazdego x i dla każdego n zachodzi

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

Rzeczywiście. Dla $n = 0$ mamy

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

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

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

Z własności f3) otrzymujemy formułę równoważną

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

Łatwo dostrzec, że końcowa wartość zmiennej aux jest równa $h(k,x,f(k,x))$ czyli jest równa $f(k+1,x)$. Powyższe rozumowanie pozwala uznać za udowodnione ponizsze twierdzenie

Twierdzenie 5.12. Rozważmy programy operujące tylko na typie integer i zbiór $\mathfrak{F}$ funkcji obliczanych przez te programy. Zbiór funkcji pierwotnie rekurencyjnych jest równy zbiorowi $\mathfrak{F}$.

W latach 20 XX wieku Dawid Hilbert zapytał czy zbiór funkcji (pierwotnie) rekurencyjnych zawiera wszystkie funkcje obliczalne. Jego dwaj asystenci Wilhelm Ackermann i Gabriel Sudan skonstruowali dwa kontrprzykłady. Pokazali mianowicie funkcje obliczalne, które nie dają się zdefiniować przy pomocy superpozycji i schematu rekursji prostej. Funkcja Ackermanna jest dziś lepiej znana niż funkcja Sudana. O funkcji Ackermanna możesz poczytać w powojennym wydaniu książki Kalejdoskop Matematyczny Hugo Steinhausa [] lub na Wikipedii. Obie funkcje rosną szybciej niż dowolna funkcja pierwotnie rekurencyjna. Co to oznacza? 1°Przy obliczeniu wartości funkcji $A(6)$ szybko zabraknie nam pamięci by zapisywać kolejne cyfry wyniku. 2°Ale nie martw się, zanim do tego dojdzie to umrze wiele pokoleń. W ten, nieco makabryczny, sposób zwracamy Ci uwagę na aspekt czasowy. Chociaż funkcja Ackermanna jest obliczalna to dla rodzaju ludzkiego nie ma to znaczenia. Już dla niewielkich argumentów czas potrzebny do obliczenia wyniku jest niewyobrażalnie wielki.
W następnym rozdziale wprowadzimy instrukcję iteracji while i będziemy kontynuować nasze rozważania.

8. Podsumowanie

Światły programista wie, że następujące funkcjonały tzn. operacje na funkcjach zwracające funkcje sa programowalne w języku $\mathcal{L}_3$ programów elementarnych:

Suma: $\sum_{i=1}^{n} \tau(i)$	$\left\{ \begin{array}{l} \textbf{s:=0;} \\ \textbf{for i:= 1 to n do} \\ \quad \textbf{s:= s+}\tau\textbf{(i)} \\ \textbf{done} \end{array} \right\} (s = \sum_{i=1}^{n} \tau(i))$
Iloczyn: $\prod_{i=1}^{n} \tau(i)$	$\left\{ \begin{array}{l} \textbf{s:=1;} \\ \textbf{for i:= 1 to n do} \\ \quad \textbf{s:= s*}\tau\textbf{(i)} \\ \textbf{done} \end{array} \right\} (s = \prod_{i=1}^{n} \tau(i))$
Maksimum: $\mathop{\mathbf{Max}}_{i=1}^{n} \tau(i))$	$\left\{ \begin{array}{l} \textbf{s:=-}\infty\textbf{;} \\ \textbf{for i:= 1 to n do} \\ \quad \textbf{s:= if s>}\tau\textbf{(i)} \\ \quad \textbf{then s:=} \tau\textbf{(i) fi} \\ \textbf{done} \end{array} \right\} (s = \mathop{\mathbf{Max}}_{i=1}^{n} \tau(i))$
Minimum: $\mathop{\mathbf{Min}}_{i=1}^{n} \tau(i))$	$\left\{ \begin{array}{l} \textbf{s:=}\infty\textbf{;} \\ \textbf{for i:= 1 to n do} \\ \quad \textbf{s:= min(s,}\tau\textbf{(i))} \\ \textbf{done} \end{array} \right\} (s = \mathop{\mathbf{Min}}_{i=1}^{n} \tau(i))$
alternatywa : $\bigvee_{i=1}^{n} \alpha(i)$	$\left\{ \begin{array}{l} \textbf{s:=false;} \\ \textbf{for i:= 1 to n do} \\ \quad \textbf{s:= s} \vee \alpha(i) \\ \textbf{done} \end{array} \right\} (s = \bigvee_{i=1}^{n} \alpha(i))$
koniunkcja: $\bigwedge_{i=1}^{n} \alpha(i)$	$\left\{ \begin{array}{l} \textbf{s:=true;} \\ \textbf{for i:= 1 to n do} \\ \quad \textbf{s:= s} \wedge \alpha(i) \\ \textbf{done} \end{array} \right\} (s = \bigwedge_{i=1}^{n} \alpha(i))$
kwantyfikator ograniczony ogólny : $\forall_{i=1}^{n} \alpha(i)$	$\left\{ \begin{array}{l} \textbf{s:=true;} \\ \textbf{for i:= 1 to n do} \\ \quad \textbf{if} \neg \alpha(i) \textbf{ or } \neg\textbf{s} \\ \quad \textbf{then s:= false fi} \\ \textbf{done} \end{array} \right\} (s = \forall_{i=1}^{n} \alpha(i))$
minimum ograniczone $:(\mu i \leq n)[\alpha(i)]$	$\left\{ \begin{array}{l} \textbf{for j:= 0 to n do} \\ \quad \textbf{if } \alpha(j) \textbf{ then exit fi} \\ \textbf{done ;} \\ \textbf{j:=0;} \end{array} \right\} (j = (\mu i \leq n)[\alpha(i)]$
indukcja ograniczona	$\left\{ \begin{array}{l} \textbf{i:=0; aux:=g(x);} \\ \textbf{for i :=1 to n do} \\ \quad \textbf{aux := h(i-1,x,aux)} \\ \quad \textbf{if aux > j(n,x) then} \\ \quad\quad \textbf{aux:=0} \end{array} \right\} (aux = f(n,x))$

Wprowadzanie kwantyfikatora ogólnego ograniczonego
Zakładamy, że dla każdego $1 \leq i \leq n$ prawdziwa jest formuła $\{P(i)\}\alpha(i)$, a nawet więcej ...

$$\frac{\forall_{i=1}^{n} \{P(i)\}\, \alpha(i)}{\left\{\begin{array}{l}\textbf{for } i \leftarrow 1 \textbf{ to } n \textbf{ do} \\ \quad P(i) \\ \textbf{od}\end{array}\right\} \forall_{i=1}^{n}\ \alpha(i)}$$

8.1. Wnioski.

A) Jeden wniosek, to postulat by odważniej 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 być 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, nazwijmy je, algorithmpub.sty i algorithmkod.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.

Ćwiczenia.

5.1. Napisz program wyznaczajacy największą z trzech liczb x, y, z i udowodnij jego poprawność.

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 ... ?

5.3. Udowodnij, że indukcja ograniczona jest definiowalna przy pomocy rekursji prostej.

ROZDZIAŁ 6

Programowanie z tablicami

$$\mathcal{L}_0 \subsetneq \mathcal{L}_1 \subsetneq \mathcal{L}_2 \subsetneq \mathcal{L}_3 \subsetneq \mathcal{L}_4 \subsetneq \mathcal{L}_5 \subsetneq \mathcal{L}_6 \subsetneq \mathcal{L}_7 \subsetneq \mathcal{L}_8 \subsetneq \mathcal{L}_9 \subsetneq \mathcal{L}_{10}$$

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

poniżej W tym rozdziale pojawiają się obiekty. Chociaż nie jest to ich pełny zakres - nie ma obiektów klas. Napiszmy jednak postulaty dotyczace obiektów – słowami:
a) kill – jego aksjomat
b) zbiór obiektów jest skończony
czy mozna "zapętlić" wskaźniki do obiektów tablicowych?
var A,B arrayof arrayof real;
A[3]:=B[7];B[7]:=A[3]
c) typy obiektów tablicowych - aksjomat A in T
obiekt tablicowy przypisany zmiennej A jest typu wymienionego w deklaracji zmiennej A

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,)

7. Jak reagować na sygnały błędów? Np. ————————————

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 łączyć:, 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?

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 oraz składanie instrukcji (;).
Oprócz typów prostych pojawiają się typy tablicowe.

Deklaracje.

Definicja 6.1. Deklaracją A jest napis postaci

$$\text{var } A\text{: } \{\text{arrayof }\}^{+}\ \mathrm{T}$$

gdzie $A \in$ Identyfikator, T $\in$\{boolean, integer, real, char, string \}, znak $^{+}$ mówi powtórzono skończoną liczbę razy(conajmniej raz). Jest to równocześnie deklaracja zmiennej A i deklaracja arrayof^{+} T.

Przykład 6.2. Jest deklaracją zmiennej tablicowej napis

$$\text{var X: array of integer ,}$$

także napisy

var B,D: arrayof real, C: arrayof arrayof integer

są deklaracjami zmiennych tablicowych.

Definicja 6.2. Deklaracją w języku $\mathcal{L}_4$ jest deklaracja zmiennej tablicowej lub deklaracja z języka $\mathcal{L}_3$.

$$\mathcal{D}_4 \stackrel{df}{=} \mathcal{D}_3 \cup \{\text{zbiór napisów postaci} : \text{var A : arrayof T}\}$$

gdzie A jest identyfikatorem zmiennej, a T jest typem prostym {boolean, integer, real, char, string } lub typem tablicowym.

Ciąg deklaracji $\mathbb{D}$ występujący w programie $\mathbb{P} \in D_4^*$ to skończony ciąg deklaracji zmiennych prostych i stałych (jak w języku $\mathcal{L}_3$) i deklaracji zmiennych tablicowych.

Wyrażenia. W zbiorze wyrażeń dopuszczalne są Wyrażenia tablicowe. Zbiory wyrażeń typu prostego są większe niż w języku $\mathcal{L}_3$. Jeśli T jest typem to napis postaci ą zmienne indeksowane postaci $A[i]$ gdzie A jest zmienną typu array of T.
zmienna tablicowa A jest wyrażeniem tablicowym,
Wyrażeniem obiektowym tablicowym jest zmienna zadeklarowana jako arrayof T oraz generator tablicy
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 całkowitoliczbowymi.
Przykład

Instrukcje. Niech A i B będą zmiennymi tablicowymi, l i u wyrażeniami arytmetycznymi typu całkowitoliczbowego..

Definicja 6.3. Zbiór At_4 instrukcji atomowych języka $\mathcal{L}_4$ zawiera instrukcje drukowania, instrukcje przypisania oraz opisany poniżej zbiór instrukcji atomowych tablicowych

$$At_4 \stackrel{df}{=} At_3 \cup \left\{ \begin{array}{l} \text{array A dim(l:u),} \\ \text{A:=B,} \\ \text{B:= copy(A),} \\ \text{kill(A)} \end{array} \right\}$$

Definicja 6.4. Zbiór $\mathcal{I}_4$ instrukcji języka $\mathcal{L}_4$ jest najmniejszym zbiorem napisów zawierającym zbiór instrukcji atomowych At_4 i zamkniętym ze względu na operacje tworzenia instrukcji warunkowych (if), instrukcji powtarzania (for) i składania, tj. łączenia instrukcji znakiem średnika (;).

2.1. Semantyka. Znaczenie zmiennej tablicowej. Wyznaczanie zmiennej tablicowej: jeśli zmienna indeksowana jest postaci $A[\tau]$ to trzeba wyznaczyc wartość $l = \tau(v)$ wyrażenia indeksującego τ i nastepnie należy sprawdzić czy w obiekcie tablicowym A istnieje zmienna A[l] .

3. 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:

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

(2) Przypisz zmiennej tablicowej A obiekt tablicowy B A := B
W efekcie wykonania tej instrukcji nowa wartość zmiennej A to obiekt tablicowy będący wartością wyrażennia tablicowego B.

(3) 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

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

(4) Usuń obiekt tablicowy

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!

(5) 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 tab
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

(6) 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.
Narysuj diagramy ilustrujące różnicę pomiędzy tymi instrukcjami! 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.

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
```

4. 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.

$$\mathcal{T}_4 \vdash A \text{ in arrayof } T \vee A = \text{none}$$

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:

(AxArr)

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

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

popraw ten aksjomaT: jak zapisać, że A jest różne od wszystkich dotychczasowych obiektów, jak zapisać, że liczba obiektów tablicowych jest skończona? uwzględnij typ T
Aksjomat przypisania
PRE $\Rightarrow$ {A(i):=wyr }(POST)

Twierdzenie 6.1. Zakładamy, że B jest tablicą n-elementową $B_1, \ldots, B_n$. 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, tzn. jest prawdą, że $j = k \Rightarrow P(i)(j = k)$. Poniższa reguła wnioskowania jest poprawna w systemie Loglan

$$\mathcal{T}_4 \vdash \frac{\forall_{i=1}^{n} \{P(i)\} (B(i) = \tau(i))}{\left\{ \begin{array}{l} \textbf{for } i \leftarrow 1 \textbf{ to } n \textbf{ do} \\ \quad P(i) \\ \textbf{od} \end{array} \right\} \forall_{i=1}^{n} (B(i) = \tau(i))}$$

Dowód. Zastosujemy poprzednie twierdzenie wprkwog ... □

Przydatna też będzie odmiana powyższego twierdzenia

Twierdzenie 6.2. Jeśli

$$\mathcal{T}_4 \vdash \frac{\exists_{i=1}^{n} \{P(i)\} (B(i) = \tau(i))}{\left\{ \begin{array}{l} \textbf{for } i \leftarrow 1 \textbf{ to } n \textbf{ do} \\ \quad P(i) \\ \textbf{od} \end{array} \right\} \exists_{i=1}^{n} (B(i) = \tau(i))}$$

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?

. Co zrobić gdy tablica A 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.

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 $A(3) = 7 \wedge B(3) =$

7. aksj.? zastosuj dwukrotnie aksj. przypisania 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 $\boxed{\{x := \tau\}\alpha(x) \Leftrightarrow \alpha(x/\tau)}$ jest prawdziwa. przemyśl tablice dwuwskaźnikowe 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. 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.

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

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

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

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

Dowód. Najpierw zauważmy, że na mocy twierdzenia 5.5, dla $k = n, n-1, \ldots, 1$ zachodzą formuły

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

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

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

Każda z tych formuł jest równoważna odpowiednio następującej

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

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

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.

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

□

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 [].

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 5.5 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 $\forall_{j=1}^{n}$ można wyrazić przez pętlę for porównaj tw. 6.1.

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

Jeszcze raz stosujemy ten sam fakt by wprowadzić kwantyfikator $\forall_{i=1}^{n}$ za program

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

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?
Co jest lepsze? sprawdzenie czy macierze A i B mają odpowiednie kształty (n x k) i (k x m). Czy też odpowiednia obsługa błędu reference to none?

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.

```
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, roz-
    miaru 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 $j, 1 \leq j \leq n,$ $\qquad W_j = \sum_{i=1}^{n \div 2} A_{j,2i-1} * A_{j,2i}$

```
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 $j, 1 \leq j \leq n$, $V_j = \sum_{i=1}^{n \div 2} B_{2i-1,j} * B_{2*i,j}$

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

```
44:       C[i, j] := s − W[i] − V[j];
```

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

```
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 $i, j, 1 \leq i \leq n, 1 \leq j \leq n$, $C_{i,j} = \sum_{k=1}^{n}$

```
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

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

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

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

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

Zwróć uwagę na to, że zewnętrzna instrukcja for uzasadnia wprowadzenie kwantyfikatora ograniczonego, a wewnętrzna instrukcja for oblicza sumę. Instrukcja for ma jeszcze wiele innych zastosowań.

Lemat 6.7. Dla każdego $j, 1 \leq j \leq n$, 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. $\forall_{1 \leq i \leq n}\ \forall_{1 \leq j \leq n}$

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

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

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

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

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

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

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

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 $A[i, n] * B[n, j]$. □

---

8. Namawiamy do notacji matematycznej

Zauważ, że przyjęcie konwencji zaproponowanej w poprzednim rozdziale por.8.1, upraszcza zapis algorytmów macierzowych.
Mnożenie macierzy (zwykłe) zapisuje się tak

```
\[\bigl\{\mathop{\forall}_{1\leq␣i␣\leq␣n}␣\,␣\mathop{\forall}_{1\leq␣j␣\leq␣n}
␣C_{ij}\leftarrow␣\sum_{1\leq␣k␣\leq␣n␣}A_{ik}*B_{kj}
␣\bigr\}␣\]␣
```

co może dać w druku taki efekt

$$\Big\{ \mathop{\forall}_{1\leq i\leq n}\ \mathop{\forall}_{1\leq j\leq n} C_{ij} \leftarrow \sum_{1\leq k\leq n} A_{ik} * B_{kj} \Big\}$$

a jeśli zastosujemy TeX'a z odpowiednim stylem, (może stworzysz taki styl?) da program z potrójną pętlą for.

```
for i := 1 to n do
   for j:= 1 to n do
      s := 0;
      for k :=1 to n do s := s+ A[i,k]*B[k,j] od;
      C[i,j] := s
   od
od
```

Algorytm Winograda także zapisze się krócej

$$\left\{ \begin{array}{l} m \leftarrow n \,\mathbf{div}\, 2; \\ \forall_{1\leq j\leq n}\, V_j \leftarrow \sum_{1\leq i\leq m}(B_{2*i-1,j} * B_{2*i,j}); \\ \forall_{1\leq j\leq n}\, W_j \leftarrow \sum_{1\leq i\leq m}(A_{i,2*j-1} * A_{i,2*j}); \\ \quad \begin{array}{l} \forall_{1\leq i\leq n} \\ \forall_{1\leq j\leq n} \end{array} \left\{ \begin{array}{l} s \leftarrow \sum_{1\leq k\leq m}(A_{i,2*k-1} + B_{2*k,j}) * (B_{2*k-1,j} + A_{i,2*k}); \\ C_{ij} \leftarrow s - W_i - V_j; \\ \textbf{if } odd(n) \textbf{ then } C_{ij} \leftarrow C_{ij} + A_{in} * B_{nj} \textbf{ fi} \end{array} \right\} \end{array} \right\}$$

Zapis dla druku, tj. dla człowieka jest zwięzły i czytelny. Jest też czterokrotnie krótszy od kodu.

9. 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 ...

Ćwiczenia

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

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.

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?

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

6.5. Algorytm mnożenia macierzy napisaliśmy przyjmując pewne założenia. Spróbuj ...

ROZDZIAŁ 7

$\mathcal{L}_5$ Programowanie z while

1

1. Programy iteracyjne

Język $\mathcal{L}_5$ jest bogatszy od poprzedniego o jeden tylko rodzaj instrukcji: instrukcję while. Rozszerzenie języka wydaje się niepozorne. Ale pojawiają się nowe zjawiska obliczeniowe.

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.

Przykład 7.1. W latach 30 dwudziestego wieku Ackermann zdefiniował, funkcję, która rośnie 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.

1.1. Składnia. Zbiór programów iteracyjnych różni się od poprzednio opisanego zbioru programów z tablicami, tym, że w ciągu instrukcji takiego programu mogą pojawiać się instrukcje while.

1

$$\mathcal{L}_0 \subsetneq \mathcal{L}_1 \subsetneq \mathcal{L}_2 \subsetneq \mathcal{L}_3 \subsetneq \mathcal{L}_4 \subsetneq \mathcal{L}_5 \subsetneq \mathcal{L}_6 \subsetneq \mathcal{L}_7 \subsetneq \mathcal{L}_8 \subsetneq \mathcal{L}_9 \subsetneq \mathcal{L}_{10}$$

Składnia 7.1. Zbiór instrukcji programów iteracyjnych jest to najmniejszy zbiór P napisów taki, że

i) każda instrukcja atomowa (drukowania, przypisania oraz instrukcja działania na tablicach) należy do zbioru P,
ii) zbiór P jest zamknięty z względu na operacje: złożenia, powtarzania – for oraz rozgałęzienia,
iii) jeśli napis γ jest formułą boolowską i napis K jest instrukcją, to napis

while γ do K od

jest instrukcją iteracji

Przykład 7.2. Dwa przykłady – druga instrukcja operuje na stosach s i t.

```
while x<y do x:=x+1; u:=3*u od
```

```
while ¬ empty(s) do
    e:= top(s);
    s := pop(s);
    t:=push(e,t)
od
```

1.2. Semantyka. Modyfikujemy odpowiednio definicję obliczenia tak by objęła ona programy z języka $\mathcal{L}_5$.

Relacja bezpośredniego następstwa $\mapsto$ konfiguracji (tj. stanów obliczeń) jest nadzbiorem relacji opisanej we poprzednim rozdziale.

Definicja 7.2. Bezpośrednim następnikiem stanu $c : \langle v, \textbf{while } \gamma \textbf{ do } K \textbf{ od}; \mathbf{s}\rangle$ jest konfiguracja określona poniżej

$$c \mapsto \begin{cases} \langle v, \mathbf{s}\rangle & \textbf{gdy } \gamma(v) = false \\ \langle v, K; \textbf{ while } \gamma \textbf{ do } K \textbf{ od}; \mathbf{s}\rangle & \textbf{gdy } \gamma(v) = true \end{cases}$$

Pozostałe przypadki zostały opisane we wcześniejszych rozdziałach.

2. Komputer $\mathcal{K}_5$

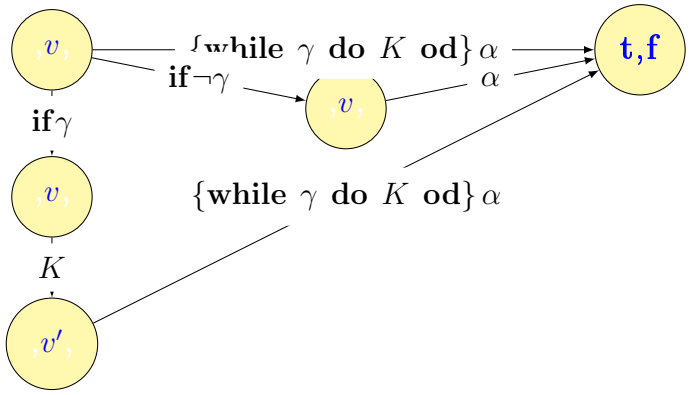
Komputer $\mathcal{K}_5$ potrafi wykonywać wszystkie polecenia z repertuaru komputera $\mathcal{K}_4$ i ponadto potrafi wykonać polecenie while. Zapewniając przy tym, że realizacja polecenia while będzie zgodna z definicją bezpośredniego następstwa konfiguracji, zob. powyżej.

3. Aksjomat instrukcji while i reguła wnioskowania

Znaczenie instrukcji while opisuje aksjomat i reguła wnioskowania.
Aksjomat while

($\mathbf{Ax}_{21}$)
$\{\mathbf{while}\,\gamma\,\mathbf{do}\,K\,\mathbf{od}\}\,\alpha \Leftrightarrow \big((\neg\gamma \wedge \alpha) \vee (\gamma \wedge \{K;\,\mathbf{while}\,\gamma\,\mathbf{do}\,K\,\mathbf{od}\}\alpha)\big)$

Zwróć uwagę – w odróżnieniu od aksjomatów $\mathbf{Ax}_{18}$ –



Rysunek 1. Aksjomat instrukcji while $\mathbf{Ax}_{21}$ (schemat)

$\mathbf{Ax}_{20}$, ten aksjomat nie eliminuje instrukcji while. Nie daje też żadnej wskazówki pozwalającej ograniczyć liczbę powtórzeń iterowanej instrukcji K. Stąd wynika konieczność wzbogacenia rachunku programów o następującą regułę wnioskowania.
Reguła wnioskowania R_3– wprowadzanie while w poprzedniku implikacji.

(R3) $$\frac{\{M(\textbf{if } \gamma \textbf{ then } K \textbf{ fi})^i\, \alpha \Rightarrow \beta\}_{i\in N}}{\{M; \textbf{while } \gamma \textbf{ do } K \textbf{ od}\}\, \alpha \Rightarrow \beta}$$

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:
- przykłady: program PF (Fermat) i program CollatzForAllN. Te programy nie mają obliczenia skończonego.
  Zastanów się: to nie jest takie oczywiste. W przypadku Fermata zajęło to ponad 300 lat pracy wielu matematyków, w przypadku Collatza dziesiatki tysięcy informatyków i matematyków pracuja i ... nic.
  Instrukcja while (odp. operator minimum efektywnego) to narzędzie trudne do opanowania.

4.1. Czy to wystarczy? 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 (ale innymi słowami) wypowiedział i udowodnił Stephen C. Kleene w roku 1936 [Kle36].

Twierdzenie 7.1. Prawdziwe są inkluzje:

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

real) jest obliczalna w dziedzinie liczb całkowitych.

Okazuje się, że w języku $\mathcal{L}_5$ i dziedzinie liczb całkowitych (nieujemnych) potrafimy zaprogramować każdą funkcję obliczalną (porównaj []) i każda funkcja obliczalna może być zaprogramowana w tym języku. Jest to fragment łańcucha twierdzeń uzasadniających tezę Churcha-Turinga. []. Języki programowania zawierają ponadto inne narzędzia: procedury, funkcje, klasy, współprogramy i procesy. Narzędzia te ułatwiaja pracę ludzi tworzących oprogramowanie, pozwalają na oszczędności w gospodarce czasem i zasobami komputera. Zajmiemy się nimi w dalszej części tej książki.

Pytania. Sformułujemy tu kilka pytań jakie nasuwają się człowiekowi myślącemu:

- Jeśli programy while pozwalają zaprogramować obliczenie każdej funkcji obliczalnej, to po co wprowadzać deklaracje modułów procedury, funkcji, klasy, współprogramu, procesu? Czy nie sa one zbędnym dodatkiem?
- Programy while pozwalają zaprogramować obliczenia funkcji obliczalnych, które nie są pierwotnie rekurencyjne. To dobrze.
  Ale czy na pewno? Przykłady skonstruowane przez Ackermanna, Sudana, Rozsę Peter i innych wskazują na to, że te funkcje są niepraktyczne. Rośną one tak szybko, że już dla niewielkich argumentów np. n=6 ich obliczenie mogłoby trwać tak długo, że końca obliczeń nie doczekałaby się ludzkość. A więc poco instrukcja while? Czy gra jest warta świeczki?
- Jak to zobaczymy w praktyce programowania mamy często (najczęściej) z przetwarzaniem skończonych zbiorów informacji. W zastosowaniach w bankowości, administracji, wojsku, wywiadzie i policji, medycynie , etc. programy przetwarzaja zbiory informacji zmagazynowane w pamięci komputerów. Programiści odczuwają potrzebe konstrukcji foreach.

Instrukcja taka powinna być podobna do instrukcji for.
Niech S będzie zbiorem elementów, obiektów, informacji. Niech e będzie zmienną typu element. Wtedy instrukcja

foreach e in S do I od

zapewniałaby powtórzenie ciągu instrukcji I dla każdego elementu występującego w skończonym zbiorze S. Niewiele języków programowania oferuje taka instrukcję. Więcej na ten temat napiszemy w drugiej częsci tej książki.

5. Własności instrukcji while

Poniżej przytaczamy kilka własności tej instrukcji. Zachęcamy do poszukiwania dalsych.

(1) Instrukcja while jest idempotentna.

$$\{\textbf{while } \gamma \textbf{ do } M \textbf{ od}\}\alpha \Leftrightarrow \left\{ \begin{array}{l} \textbf{while } \gamma \textbf{ do } M \textbf{ od}; \\ \textbf{while } \gamma \textbf{ do } M \textbf{ od} \end{array} \right\} \alpha$$

A więc nie warto powtarzać tej instrukcji dwukrotnie[2].

(2) 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

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

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

[2]Czasami jednak powtórzenie takiej instrukcji może być pomocne w zrozumieniu co nasz program robi.

(3) Przydatną tautologią jest następująca równoważność

$$\left\{\begin{array}{l}\textbf{while } \gamma \textbf{ do}\\ \quad \textbf{if } \delta\\ \qquad \textbf{then } K\\ \qquad \textbf{else } M\\ \quad \textbf{fi}\\ \textbf{od}\end{array}\right\}\alpha \Leftrightarrow \left\{\begin{array}{l}\textbf{while } (\gamma \wedge \delta) \textbf{ do } K \textbf{ od};\\ \textbf{while } \delta \textbf{ do}\\ \quad \textbf{while } (\gamma \wedge \delta) \textbf{ do } K \textbf{ od};\\ \quad \textbf{while } (\neg\gamma \wedge \delta) \textbf{ do } M \textbf{ od};\\ \textbf{od};\end{array}\right\}\alpha$$

(4) Myslimy , że warto wprowadzić następującą definicję porządku w zbiorze programów. Niech symbole K oraz M oznaczają dwa programy. Niech α będzie oznaczeniem formułą algorytmiczną.

Definicja 7.3.

$$K \overset{\alpha}{\leq} M \overset{df}{=} \mathcal{T} \vdash (K\alpha \Rightarrow M\alpha)$$

i zbadać tę relację.
Powinniśmy łatwo udowodnić

$$\textbf{while}\,\gamma\,\textbf{do}\,K\,\textbf{od} \overset{\alpha}{=} \underset{i\in N}{g.l.b.}\{\textbf{if}\,\gamma\,\textbf{then}\,K\,\textbf{fi}\}^i$$

Pomocnicze reguły wnioskowania. W wielu dowodach korzystamy z reguł wnioskowania przytoczonych poniżej.

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

Jeszcze inna reguła

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

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.

6. Algorytmiczne teorie typów pierwotnych

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

6.1. Aksjomatyczna teoria $\mathcal{ATI}$. 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 tego aksjomatu należy(!) użyć następującej formuły algorytmicznej

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

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

Twierdzenie 7.2. (prawo Archimedesa jest twierdzeniem algorytmicznej teorii typu integer $\mathcal{ATI}$)
(ArchI)
$\mathcal{ATI} \vdash ((0 < y < x) \Rightarrow \{t \leftarrow y;\ \textbf{while}\ t \leq x\ \textbf{do}\ t \leftarrow t+y\ \textbf{od}\}(t > x))$

Twierdzenie 7.3. W algorytmicznej teorii typu integer można udowodnić poprawność algorytmu Euklidesa.

$$\mathcal{ATI} \vdash \forall n, m > 0 \left\{ \begin{array}{l} \textbf{while}\ n \neq m\ \textbf{do} \\ \quad \textbf{if}\ n > m\ \textbf{then}\ n := n - m \\ \quad\quad \textbf{else}\ m := m - n\ \textbf{fi} \\ \textbf{od} \end{array} \right\} (n = m)$$

Można też udowodnić kategoryczność teorii $\mathcal{ATI}$

Twierdzenie 7.4. Każde dwa modele teorii $\mathcal{ATI}$ są izomorficzne.

6.2. Aksjomaty typu real. Do poprzednio wymienionych aksjomatów ciała możemy teraz dodać aksjomat (prawo?) Archimedesa
(Arch)
$((0 < y \wedge y < x) \Rightarrow \{t \leftarrow y;$ **while** $t \leq x$ **do** $t \leftarrow t + y$ **od**$\}(t > x))$

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 7.5. 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 $\Re\mathfrak{D}$ liczb rzeczywistych wtedy i tylko wtedy, gdy istnieje dowód tej formuły z aksjomatów zbioru Z.

$$\Re\mathfrak{D} \models \psi \text{ wtedy i tylko wtedy, gdy } Z \vdash \psi$$

Sens tego twierdzenia sprowadza się do następującej obserwacji, każda semantyczna własność w_ψ programu(-ów) z języka $\mathcal{L}_5$ wykonywanego w ciele liczb rzeczywistych z porządkiem, jest prawdziwa w ciele $\Re\mathfrak{D}$ wtedy i tylko wtedy gdy wyrażająca tę własność formuła algorytmiczna ψ posiada dowód z aksjomatów zbioru Z. Jest tak, ponieważ widzieliśmy, że własności semantyczne takie jak stop programu, poprawność programu, własność fault, równoważność, etc. są wyrażalne formułami będącymi boolowskimi kombinacjami formuł postaci $K\alpha$.

7. Przykłady

W tym podrozdziale przytoczymy kilka dowodów ...

7.1. Eudoksosa zasada wyczerpywania. Eudoksos w III wieku p.n.e, a później Archimedes, dokonywali

obliczeń wielkości pola (lub objętości bryły) posługując się następującą zasadą wyczerpywania:

> Załóżmy, że x jest poszukiwaną wielkością pola figury F. Skonstruujmy ciąg figur $F_1, F_2, \ldots$, o polach $p_1, p_2, \ldots$ i taki, że dla każdego i, zachodzi nierówność $x - p_i < x/2^i$. Możemy więc obliczyć pole figury F z błędem dowolnie małym $x - p^i < \epsilon$.

Zasada wyczerpywania jest równowazna prawu Archimedesa. Antoni Kreczmar [Kre77b] zapisał tę zasadę w postaci schematu formuł algorytmicznych $H(K)$. Niech K będzie programem, niech x będzie jedyną zmienną wejścia-wyjścia tego programu. Wybierzmy trzy zmienne t, u, z nie występujące w programie K. Każda formuła $H(K)$ o podanym poniżej schemacie jest instancją zasady wyczerpywania.

$$\underset{t,u,y>0}{\forall}\left(((x = t \wedge z = y) \Rightarrow \bigcap\{K;\, z \leftarrow \frac{z}{2}\}(0 \leq x \leq z)) \Rightarrow \{x \leftarrow t\}\bigcup K(x \leq u)\right)$$

Formułę tę można wyprowadzić z aksjomatów pierwotnego typu real. Przypomnijmy jednym z aksjomatów jest prawo Archimedesa.
Zastosowane zasady wyczerpywania pokazujemy poniżej, podczas analizyy algorytmu obliczającego przybliżoną wartość $\sqrt{a}$.

Możemy także stosować następującą regułę wnioskowania przydatną w rozumowaniach o własnościach algorytmów działających w typie real.

$$\mathcal{ATR} \vdash \frac{(x = t) \Rightarrow \{K(x)\}(x < t/2)}{\{\textbf{while}\, x \geq \epsilon\, \textbf{do}\, K(x)\, \textbf{od}\}\,(x < \epsilon)}$$

Nietrudno przekonać się o jej poprawności. Ta reguła wnioskowania znajdzie swe zastosowanie poniżej podczas analizy kolejnych algorytmów.

7.2. Obliczanie $\sqrt{a}$ z żądaną dokładnością.

Twierdzenie 7.6. Program

$$\left\{\begin{array}{l} x := (a+1)/2; \\ \textbf{while}\ (x - a/x) \geq \epsilon\ \textbf{do}\ x := (x + a/x)/2\ \textbf{od} \end{array}\right\} \tag{16}$$

oblicza pierwiastek z liczby dodatniej $a > 0$ z dokładnością do ϵ.

Dowód. Rozważmy program

(17)
$$\left\{\begin{array}{l} d := \epsilon; \\ \textbf{while } d \geq \epsilon \textbf{ do} \\ \quad I : \left\{\begin{array}{l} \textbf{if } d = \epsilon \\ \textbf{then } x := (a+1)/2 \\ \textbf{else } x := (x + a/x)/2 \\ \textbf{fi}; \\ d := (x - a/x) \end{array}\right\} \\ \textbf{od} \end{array}\right\}$$

Ten program oblicza wartość zmiennej x w taki sam sposób jak program poprzedni, lub oba programy mają obliczenie nieskończone. Wystarczy wykluczyć tę drugą możliwość, tzn. wykazać, własnośc stopu programu 17. Wartość zmiennej d jest obliczoną dokładnością. Wykażemy, że dla każdej pary liczb a i ϵ takiej, że $a > 0$ i $\epsilon > 0$ program ten ma obliczenie skończone.

Zastosujemy zasadę wyczerpywania. Najpierw wykażemy, że prawdziwa jest formuła

(18)
$$\{d := \epsilon;\ z := a + \epsilon + 1\} \bigcap \{I; z := z/2\}(0 \leq d \leq z)).$$

Należy więc wykazać, że dla każdej liczby naturalnej $i \in N$ zachodzi formuła

(19)
$$\{d := \epsilon;\ z := a + \epsilon + 1\}\{I; z := z/2\}^i(0 \leq d \leq z)).$$

Dla $i = 0$ rzeczywiście zachodzi formuła

(20)
$$\{d := \epsilon;\ z := a + \epsilon + 1\}(0 \leq d \leq z)).$$

Dla sprawdzenia wystarczy zastosować aksjomat instrukcji przypisania i fakt, że

(21)
$$0 \leq \epsilon \leq a + \epsilon + 1.$$

Dla $i = 1$ skorzystamy z następującego faktu: poniższa formuła

(22)
$$\{x := (a+1)/2;\ d := x - a/x\}(0 \leq d \leq (a + \epsilon + 1)/1)$$

jest równowazna następnej formule

(23)
$$((a+1)/2)^2 \geq a \wedge 0 \leq (a-1)^2/2 * (a+1) \leq (a + \epsilon + 1)/2$$

a więc

(24)
$$\left\{\begin{array}{l} d := \epsilon; \\ z := a + \epsilon + 1; \\ I; \\ z := z/2 \end{array}\right\} (x^2 \geq a \wedge 0 \leq d \leq z \wedge d = x - a/x)).$$

Dla kroku indukcyjnego, najpierw sprawdzamy, że zachodzi formuła

(25)
$$\begin{array}{l} (x^2 \geq a \wedge 0 \leq (x - a/x) \leq z) \Rightarrow \\ \quad (((x + a/x)/2)^2 \geq a \wedge 0 \leq (x + a/x)/2 - a/(x + a/x)/2 \leq z/2) \end{array}$$

Stąd i z własności programu I otrzymujemy

$$\underbrace{(x^2 \geq a \wedge 0 \leq d \leq z \wedge d = x - a/x)} \Rightarrow \tag{26}$$
$$(\{I; z := z/2\}\underbrace{(x^2 \geq a \wedge 0 \ \ \leq d \leq z \wedge d = x - a/x)})$$

Wykazaliśmy, że formuła $(x^2 \geq a \wedge 0 \leq d \leq z \wedge d = x - a/x)$ jest niezmiennikiem programu $\{I; z := z/2\}$. Łącząc ten fakt i implikacje 20 oraz 24 dowodzimy, że
dla każdej liczby naturalnej $i \in N$

$$\{d := \epsilon;\ z := a + \epsilon + 1\};\ \{I;\ z := z/2\}^i (0 \leq d \leq z) \tag{27}$$

Wynika stąd, że stosując regułę R_5, (zob. następny rozdział, strona 155) możemy wprowadzić ogólny kwantyfikator iteracji by otrzymać poprzednik w zasadzie wyczerpywania

$$\{d := \epsilon;\ z := a + \epsilon + 1\} \bigcap \{I; z := z/2\}(0 \leq d \leq z). \tag{28}$$

A więc, stosując zasadę wyczerpywania, wykazaliśmy, że prawdziwa jest formuła

$$a > 0 \wedge \epsilon > 0 \Rightarrow \{d := \epsilon\} \bigcup I(d < \epsilon) \tag{29}$$

Wykorzystamy następujące twierdzenie rachunku programów (zob. formuła (2) str. 62 [MS87])

$$\textbf{while } \gamma \textbf{ do } M \textbf{ od}\, \alpha \Leftrightarrow \bigcup \{\textbf{if } \gamma \textbf{ then } M \textbf{ fi}\}(\neg\gamma \wedge \alpha) \tag{30}$$

i z jego pomocą wprowadzimy operator while

$$a > 0 \wedge \epsilon > 0 \Rightarrow \{d := \epsilon; \textbf{while } d \geq \epsilon \textbf{ do } I \textbf{ od}\}\,(d < \epsilon) \tag{31}$$

Udowodniliśmy więc, że nasz program 17 kończy obliczenia i wobec tego program 16 też kończy obliczenia. □

7.3. Bisekcja. Zadanie, które teraz omówimy, polega na znalezieniu przybliżonej wartości zera funkcji $f(x) = x^5 + 7x^3 - 7x + 12$ w przedziale [-10,10].
Przypatrzmy się własnościom funkcji f. Łatwo zauważyć, że f jest funkcją jednej zmiennej rzeczywistej o wartościach rzeczywistych, że f jest funkcją ciągłą na odcinku $[-10, 10]$, że na końcach przedziału przyjmuje różne znaki $f(-10) * f(10) < 0$ i że funkcja f jest obliczana przez poniższy program Mf.

$$\{y := x^5 + 7x^3 - 7x + 12\} \tag{Mf}$$

Wybiegając w przyszłość sformułujemy algorytm o szerszym zastosowaniu.

Twierdzenie 7.7. Założenia.

(i) Funkcja f jest określona na przedziale $[a, b]$ i ciągła,
(ii) f jest obliczana przez pewien program $\boxed{M}$ tzn. jest prawdą, że
$\forall_{a \leq l \leq b}\{x := l;\ \boxed{M}\}(y = f(l))$,
(iii) znaki funkcji f na końcach przedziału $[a, b]$ są różne, tj. $f(a) * f(b) < 0$.

Teza. Następująca formuła jest twierdzeniem algorytmicznej teorii liczb rzeczywistych.

(32)
$$\left\{\begin{array}{l} x := a;\ \boxed{M};\ fa := y; \\ x := b;\ \boxed{M};\ fb := y; \\ \textbf{while}\ (b-a) \geq \epsilon\ \textbf{do} \\ \quad x := (a+b)/2;\ \boxed{M}; \\ \quad \textbf{if}\ y = 0 \\ \quad \textbf{then}\ b := a;\ \textbf{exit} \\ \quad \textbf{else} \\ \quad\quad \textbf{if}\ y * fa < 0 \\ \quad\quad \textbf{then}\ b := x;\ fb := y \\ \quad\quad \textbf{else}\ a := x; fa := y \\ \quad\quad \textbf{fi} \\ \quad \textbf{fi} \\ \textbf{od} \end{array}\right\} (f(x) = 0 \vee ((b-a) < \epsilon) \wedge f(a) * f(b) < 0)$$

Zauważyłeś, że ten program jest rozwlekły i trzykrotnie powtarza program M. Wybiegając trochę naprzód wprowadzimy pewne rozszerzenie języka. Jeżeli potrafimy udowodnić prawdziwość formuły

$$\mathcal{ATR} \vdash (x = l) \Rightarrow \{M\}(y = f(l))$$

to wolno nam wprowadzić deklarację tj. definicję nowego funktora

i program przyjmie prostszą postać

(33)
$$\left\{\begin{array}{l} \quad \text{unit}\ f : \textbf{function}(x : real) : real \\ \quad \textbf{begin} \\ \quad\quad \boxed{M} \\ \quad \textbf{end}f \\ \hline fa := f(a); \\ fb := f(b); \\ \textbf{while}\ (b-a) \geq \epsilon\ \textbf{do} \\ \quad x := (a+b)/2;\ \ y := f(x); \\ \quad \textbf{if}\ y = 0\ \textbf{then}\ a, b := x \\ \quad \textbf{else} \\ \quad\quad \textbf{if}\ y * fa < 0 \\ \quad\quad \textbf{then}\ b := x;\ fb := y \\ \quad\quad \textbf{else}\ a := x; fa := y \\ \quad\quad \textbf{fi} \\ \quad \textbf{fi} \\ \textbf{od} \end{array}\right\}$$

Naszym celem jest analiza poprawności tego programu. Zwróc uwagę: w programie nie widać założeń twierdzenia jakie chcemy udowodnić. Deklarację funkcji należy, w umyśle czytającego(!) uzupełnić twierdzeniem o określoności funkcji.

Dowód. W dowodzie wykorzystujemy prawo Archimedesa i inne aksjomaty ciała liczb rzeczywistych. Dowód sprowadza się do wyprowadzenia naszej formuły z aksjomatu $Arch$. Najpierw jednak udowodnimy, że z aksjomatu $Arch$ wynika inna, wygodniejsza w zastosowaniu formuła:

Lemat 7.8.

$$R_{arch} \vdash ((0 < a \wedge a < b) \Rightarrow \{\textbf{while } a < b \textbf{ do } b := b/2 \textbf{ od}\}(b < a))$$

Udowodnimy, że każda formuła zbudowana według poniższego schematu

$$(\{t := a; \textbf{ if } t < b \textbf{ then } t := 2{*}t \textbf{ fi}^n\}(b < t)) \Rightarrow \{\textbf{if } a < b \textbf{ then } b := b/2 \textbf{ fi}\}^n(b < a))$$

gdzie $n \in N$ jest dowolną liczbą naturalną

posiada dowód z aksjomatów ciała uporządkowanego. Tzn., że dla każdej liczby naturalnej n można udowodnić, że

$$R_< \vdash (\{t := a; \textbf{ if } t < b \textbf{ then } t := 2{*}t \textbf{ fi}^n\}(b < t)) \Rightarrow \{\textbf{if } a < b \textbf{ then } b := b/2 \textbf{ fi}\}^n(b < a))$$

Gdy to zostanie udowodnione to najpierw korzystamy z aksjomatu instrukcji while i mamy dla każdej liczby naturalnej n można udowodnić, że

$$R_< \vdash (\{t := a; \textbf{ if } t < b \textbf{ then } t := 2{*}t \textbf{ fi}^n\}(b < t)) \Rightarrow \{\textbf{while } a < b \textbf{ do } b := b/2 \textbf{ od}\}(b < a))$$

Teraz można zastosować regułę R_3 (zob. strona 155) i uzyskamy

$$R_< \vdash (\{t := a; \textbf{ while } t < b \textbf{ do } t := 2{*}t \textbf{ od}\}(b < t)) \Rightarrow \{\textbf{while } a < b \textbf{ do } b := b/2 \textbf{ od}\}(b < a))$$

Poprzednik implikacji to aksjomat Archimedesa. Udowodniliśmy więc, że

$$R_{Arch} \vdash \{\textbf{while } a < b \textbf{ do } \underline{:} = b/2 \textbf{ od}\}(b < a)).$$

Co kończy dowód lematu.

Wracamy do dowodu twierdzenia. Dla $n = 0$ bez trudu stwierdzamy, że formuła

$$(\{t := a\}(b < t)) \equiv (b < a))$$

jest tautologią, wystarczy zastosować aksjomat instrukcji przypisania
a więc

$$R \vdash (\{t := a\}(b < t)) \Rightarrow (b < a)).$$

Załóżmy, że dla $k \leq n$ jest prawdą, że

$$R \vdash (\{t := a; \textbf{ if } t < b \textbf{ then } t := 2{*}t \textbf{ fi}^k\}(b < t)) \Rightarrow \{\textbf{if } a < b \textbf{ then } b := b/2 \textbf{ fi}\}^k(b < a))$$

Zbadajmy implikację

$$(\{t := a; \textbf{ if } t < b \textbf{ then } t := 2{*}t \textbf{ fi}^{n+1}\}(b < t)) \Rightarrow \{\textbf{if } a < b \textbf{ then } b := b/2 \textbf{ fi}\}^{n+1}(b < a))$$

Z definicji iteracji K^n programu K mamy

$$\begin{aligned} \vdash (\{t := a; \textbf{ if } t < b \textbf{ then } t := 2 * t \textbf{ fi}^{n+1}\}(b < t)) \equiv \\ (\{t := a; \textbf{ if} t < b \textbf{then} t := 2 * t \textbf{fi}^n; \\ \textbf{if} t < b \textbf{then} t := 2 * t \textbf{fi}\}(b < t)) \end{aligned} \tag{34}$$

Z aksjomatu instrukcji warunkowej if otrzymujemy

$$\begin{aligned} &\vdash (\{t := a;\ \textbf{if}\ t < b\ \textbf{then}\ t := 2 * t\ \textbf{fi}^{n+1}\}(b < t)) \equiv \\ &\big(\{t := a;\ \textbf{if}\ t < b\ \textbf{then}\ t := 2 * t\ \textbf{fi}^{n}\} \\ &((t < b) \wedge \{t := 2 * t\}(b < t) \vee \neg(t < b) \wedge (b < t)\big) \end{aligned} \tag{35}$$

Dwie łatwe do udowodnienia obserwacje będą przydatne.

Lemat 7.9.

(i) $$R_< \vdash ((f(a) * f(b) < 0) \Rightarrow \left\{ \begin{array}{l} c := (a+b)/2; \\ \textbf{if} f(a) * f(c) < 0 \\ \quad \textbf{then}\ b := c \\ \quad \textbf{else}\ a := c \\ \textbf{fi} \end{array} \right\} (f(a) * f(b) < 0)$$

(ii) $$R_< \vdash ((b-a) = k) \Rightarrow \left\{ \begin{array}{l} c := (a+b)/2; \\ \textbf{if} f(a) * f(c) < 0 \\ \quad \textbf{then}\ b := c \\ \quad \textbf{else}\ a := c \\ \textbf{fi} \end{array} \right\} ((b-a) = \frac{k}{2})$$

Dowody własnośći (i) oraz (ii) są bardzo łatwe. Spróbuj sam udowodnić te własności.

Z aksjomatu Archimedesa - w jego zmienionej postaci wynika, że poniższy program się zatrzymuje. Z własności (ii) wynika

Lemat 7.10.

$$R_{Arch} \vdash ((b-a) > \delta) \Rightarrow \left\{ \begin{array}{l} \textbf{while}\ (b-a) > \delta \textbf{do} \\ \quad c := (a+b)/2; \\ \quad \textbf{if} f(a) * f(c) < 0 \\ \quad\quad \textbf{then}\ b := c \\ \quad\quad \textbf{else}\ a := c \\ \quad \textbf{fi} \\ \textbf{od} \end{array} \right\} ((b-a) < \delta)$$

Wykorzystamy własność (i) lematu 7.9 by otrzymać pożądaną tezę

$$R_{Arch} \vdash \left(\begin{array}{l} (b-a) > \delta\ \wedge \\ (f(a) * f(b) < 0 \end{array} \right) \Rightarrow \left\{ \begin{array}{l} \textbf{while}\ (b-a) > \delta \textbf{do} \\ \quad c := (a+b)/2; \\ \quad \textbf{if} f(a) * f(c) < 0 \\ \quad\quad \textbf{then}\ b := c \\ \quad\quad \textbf{else}\ a := c \\ \quad \textbf{fi} \\ \textbf{od} \end{array} \right\} \left(\begin{array}{l} (b-a) < \delta\ \wedge \\ (f(a) * f(b) < 0 \end{array} \right)$$

□

7.3.1. Uwagi. Czytelnik może zapytać, po co tak się męczyć? Przecież jest oczywiste, że ten program jest poprawny. Naprawdę? Wyobraźmy sobie, że obliczenia programu bisekcja wykonywane są w ciele niearchimedesowskim. Co wtedy?

Przypomnijmy, aż do połowy XIX wieku wierzono, że każde ciało liczbowe jest archimedesowskie. Odkrycie, że może być inaczej, było wstrząsem tylko nieco słabszym od odkrycia geometrii nieeuklidesowej. Pytanie: czy można zaprogramować strukturę ciała niearchimedesowskiego?
Wykonaliśmy pewne doświadczenie myślowe(por. niem. gedankenexperiment). Jakie wnioski można wyciągnąć przypatrując się drodze jaką przebyliśmy?

- Skąd wiemy, że znaleźliśmy (przybliżoną) wartość miejsca zerowego funkcji f? Aha, potrzebne dodatkowe założenie, że funkcja f jest ciągła.
- Wydaje się, że wiedzy o ciągłości funkcji nie da się włączyć do kompilatora.
- Mamy jednak nadzieję, że przyszłe systemy wspomagania programisty znajdą sposób by wesprzeć go w tym zadaniu.
- Z drugiej strony zawsze pewna praca człowieczego mózgu będzie niezbędnym składnikiem zawodu programisty. Dzięki temu zawód ten nie zostanie wyeliminowany przez komputery.
- W tym konkretnym przypadku tekst programu należało uzupełnić o założenia twierdzenia. Minimum wymagań to oczekiwanie, że programista wprowadzając deklarację funkcji f dołączy dowód istnienia wyniku. Dla wielomianów takie twierdzenie jest oczywiste, ale w innych przypadkach może to być trudne.
- Najważniejszy wniosek jaki należy wyciągnąć z tego przykładu to: jesli używasz funkcji to zadbaj by jej wynik był wartością określoną!

7.4. Algorytm Euklidesa. Algorytm Euklidesa, w jego najprostszej postaci, zapisujemy stosując instrukcję powtarzania while.

(E) $$\underbrace{\left\{\begin{array}{l}\textbf{while } n \neq m \textbf{ do}\\ \quad \textbf{if } n > m \textbf{ then } n := n - m \textbf{ else } m := m - n \textbf{ fi}\\ \textbf{od}\end{array}\right\}}_{algorytm\ \ Euklidesa}$$

Własność stopu tego programu nie jest oczywista. Udowodnimy, że dla kazdej pary liczb naturalnych różnych od zera algorytm Euklidesa ma obliczenie skończone. W dowodzie wykorzystamy prawo Archimedesa.

(Arch) $\forall_{n,m>0}\, n > m \Rightarrow \{k := m;\ \textbf{while } n > k \textbf{ do } k := k + m \textbf{ od}\}(n \leq k)$

Dowód. Zacznijmy od następującej obserwacji:
każde dwie formuły algorytmiczne o schematach wskazanych

poniżej, są równoważne

$$(36) \qquad \left\{\begin{array}{l}\textbf{while } \alpha \textbf{ do}\\ \quad \textbf{if } \gamma\\ \quad \textbf{then } K\\ \quad \textbf{else } M\\ \quad \textbf{fi}\\ \textbf{od}\end{array}\right\}\beta \equiv \left\{\begin{array}{l}\textbf{while } \alpha \textbf{ do}\\ \quad \textbf{while } \alpha \wedge \gamma \textbf{ do } K \textbf{ od};\\ \quad \textbf{while } \alpha \wedge \neg\gamma \textbf{ do } M \textbf{ od};\\ \textbf{od}\end{array}\right\}\beta$$

Symbole K i M można zastąpić przez dowolne programy, a symbole α, β, γ przez dowolne wyrażenia boolowskie. Równoważność 36 jest przykładem tautologii rachunku schematów programów PAL (tj. zdaniowej logiki algorytmicznej)(por. [MS87] rozdz. V, str. 206-269).

Wstawiamy:

$$\alpha : n \neq m, \qquad \beta : n = m, \qquad \gamma : n \geq m$$
$$K : n := n - m, \qquad M : m := m - n$$

naszym celem jest teraz udowodnienie formuły

$$\left\{\begin{array}{l}\textbf{while } n \neq m \textbf{ do}\\ \quad \textbf{while } n \neq m \wedge n \geq m \textbf{ do } n := n - m \textbf{ od};\\ \quad \textbf{while } n \neq m \wedge \neg(n \geq m) \textbf{ do } m := m - n \textbf{ od};\\ \textbf{od}\end{array}\right\}(n = m)$$

Uprośćmy trochę

$$\left\{\begin{array}{l}\textbf{while } n \neq m \textbf{ do}\\ \quad \textbf{while } n > m \textbf{ do } n := n - m \textbf{ od};\\ \quad \textbf{while } m > n \textbf{ do } m := m - n \textbf{ od};\\ \textbf{od}\end{array}\right\}(n = m)$$

Teraz trzykrotnie można zastosować prawo Archimedesa, w jego trochę innej postaci.

$$\forall_{n,m>0}\{\textbf{while } n > m \textbf{ do } n := n - m \textbf{ od}\}(n \leq m)$$
$$\forall_{n,m>0}\{\textbf{while } m > n \textbf{ do } m := m - n \textbf{ od}\}(m \leq n)$$

Zauważmy też, że

$$|n - m| = k \Rightarrow \left\{\begin{array}{l}\textbf{while } n > m \textbf{ do } n := n - m \textbf{ od};\\ \textbf{while } m > n \textbf{ do } m := m - n \textbf{ od};\end{array}\right\}(|n - m| < k)$$

Skąd stosując prawo Archimedesa po raz trzeci otrzymujemy

$$\forall_{n,m>0}\left\{\begin{array}{l}\textbf{while } n \neq m \textbf{ do}\\ \quad \textbf{while } n > m \textbf{ do } n := n - m \textbf{ od};\\ \quad \textbf{while } m > n \textbf{ do } m := m - n \textbf{ od};\\ \textbf{od}\end{array}\right\}|n - m| = 0$$

co kończy dowód własności stopu.

Poprawność algorytmu łatwo udowodnić posługując się trzema faktami

$$gcd(n, n) = n$$
$$gcd(n, m) = gcd(n - m, m) \quad \textbf{gdy } n > m$$
$$gcd(n, m) = gcd(n, m - n) \quad \textbf{gdy } n < m$$

□

Prawo Archimedesa posiada dowód z pozostałych aksjomatów liczb naturalnych, piszemy o tym w innym miejscu.

7.5. Obliczanie potęgi x^n. Przyjmijmy, że dwie zmienne są zadeklarowane w następujący sposób

var x: real, n: integer

i ponadto przyjmijmy, że n>0. Bardzo szybko możemy napisać program obliczający n-tą potęgę liczby x

$$P : \{\mathsf{p} := 1; \textbf{ for } \mathsf{i} := 1 \textbf{ to } \mathsf{n} \textbf{ do } \mathsf{p} := \mathsf{p} * \mathsf{x} \textbf{ od}\}.$$

Z twierdzenia 5.6 wynika, że

$$\{P\}(p = x^n)$$

Ten algorytm wykonuje n mnożeń. Można jednak znacznie zmniejszyć liczbę działań.

$$BP : \left\{ \begin{array}{l} z := x; \; y := 1; \; m := n; \\ \textbf{while } m \neq 0 \textbf{ do} \\ \quad \textbf{if } odd(m) \textbf{ then } y := y * z \textbf{ fi}; \\ \quad m := m \,\textbf{div}\, 2; \\ \quad z := z * z \\ \textbf{od} \end{array} \right\}$$

Udowodnimy, że

$$\{BP\}(p = x^n)$$

W dowodzie wykorzystamy własności liczb naturalnych oraz fakt, że wartości zmiennych x, y, z należą do pierścienia. Właściwie skorzystamy tylko z łączności mnożenia. Najpierw zauważmy, z aksjomatów algorytmicznej teorii liczb całkowitych $\mathcal{ATI}$ wynika, że zatrzymuje się program

Lemat 7.11.

$$\mathcal{ATI} \vdash \{n \geq 0 \Rightarrow \{m := n; \textbf{ while } m \neq 0 \textbf{ do } m := m \,\textbf{div}\, 2 \textbf{ od}\}\}(m = 0)$$

Dowód. Następująca formuła jest twierdzeniem algorytmicznej teorii liczb naturalnych $\mathcal{ATI}$, zob. prawo Archimedesa

$$\mathcal{ATI} \vdash \forall_{n,m}(0 < n < m) \Rightarrow \{p := n; \textbf{ while } p < m \textbf{ do } p := p + n \textbf{ od }\}(p > m)$$

Stąd łatwo wyprowadzić inne twierdzenie teorii $\mathcal{ATI}$

$$\mathcal{ATI} \vdash \forall_{n,m}(0 < n < m) \Rightarrow \{p := n; \textbf{ while } p < m \textbf{ do } p := 2 * p \textbf{ od }\}(p > m)$$

Dla każdej liczby naturalnej $i \in N$ formuła o następującej postaci jest twierdzeniem teorii $\mathcal{ATI}$

$$\mathcal{ATI} \vdash \left(\begin{array}{l} \{p := n; \textbf{ if } p < m \textbf{ then } p := p + n \textbf{ fi}\}^i(p > m) \\ \Rightarrow \{p := n; \textbf{ if } p < m \textbf{ then } p := p + p \textbf{ fi}\}^i(p > m) \end{array} \right)$$

Stosując aksjomat Ax23 dowodzimy, że każda z następujących formuł jest twierdzeniem teorii $\mathcal{ATI}$ (dla każdej liczby naturalnej $i \in N$).

$$\mathcal{ATI} \vdash \left(\begin{array}{l} \{p := n; \textbf{ if } p < m \textbf{ then } p := p + n \textbf{ fi}\}^i(p > m) \\ \Rightarrow \{p := n; \textbf{ while } p < m \textbf{ do } p := 2 * p \textbf{ od}\}(p > m) \end{array} \right)$$

Możemy teraz zastosować regułę wnioskowania R3 i uzyskujemy

$$\left(\begin{array}{l} \{p := n;\ \textbf{while}\ p < m\ \textbf{do}\ p := p + n\ \textbf{od}\}(p > m) \\ \Rightarrow \{p := n;\ \textbf{while}\ p < m\ \textbf{do}\ p := 2 * p\ \textbf{od}\}(p > m) \end{array} \right)$$

□

Kolejny lemat potrzebny do udowodnienia twierdzenia

Lemat 7.12.

$$(37) \qquad (z^m * y = x^n) \Rightarrow \left\{ \begin{array}{l} \textbf{if}\ odd(m)\ \textbf{then}\ y := y * z\ \textbf{fi}; \\ m := m\,\textbf{div}\,2; \\ z := z * z \end{array} \right\} (z^m * y = x^n)$$

stwierdza, że formuła $z^m * y = x^n$ jest niezmiennikiem programu występującego w powyższym lemacie.

Dowód. Dowód przebiega w trzech łatwych krokach.
1) Formuła wymieniona w lemacie jest równoważna następującej formule

$$(38) \quad (z^m * y = x^n) \Rightarrow \left\{ \begin{array}{l} \textbf{if}\ odd(m)\ \textbf{then}\ y := y * z\ \textbf{fi}; \\ m := m\,\textbf{div}\,2; \end{array} \right\} ((z * z)^m * y = x^n)$$

2) Jeszcze raz stosujemy aksjomat instrukcji przypisania

$$(39) \quad (z^m * y = x^n) \Rightarrow \{\ \textbf{if}\ odd(m)\ \textbf{then}\ y := y * z\ \textbf{fi};\ \}\,((z * z)^{m\,\textbf{div}\,2} * y = x^n)$$

czyli

$$(40) \qquad (z^m * y = x^n) \Rightarrow \{\ \textbf{if}\ odd(m)\ \textbf{then}\ y := y * z\ \textbf{fi};\ \}\,((z)^m * y = x^n)$$

3)

$$(41) \qquad (z^m * y = x^n) \Rightarrow \left(\begin{array}{l} odd(m) \wedge \{y := y * z\}((z * z)^{m\textbf{div}2} * y = x^n) \\ \vee \neg odd(m) \wedge (z^m * y = x^n) \end{array} \right)$$

czyli

$$(42) \qquad (z^m * y = x^n) \Rightarrow (z^m * y = x^n)$$

□

Na podstawie tych dwu lematów możemy przeprowadzić dowód twierdzenia o poprawności programu BP względem warunku początkowego $n > 0$ i warunku końcowego $y = x^n$. Wykorzystamy następującą regułę wnioskowania

$$\frac{\delta \Rightarrow M\delta}{\delta \wedge \{\textbf{while}\,\gamma\,\textbf{do}\,M\,\textbf{od}\}\textbf{true} \Rightarrow \{\textbf{while}\,\gamma\,\textbf{do}\,M\,\textbf{od}\}(\neg\gamma \wedge \delta)}$$

7.6. Najmocniejszy następnik programu M.

Definicja 7.4. Najmocniejszym następnikiem warunku wstępnego α i programu K w strukturze algebraicznej $\mathfrak{A}$ jest taka formuła β, która spełnia następujące warunki

(1) jeśli dane początkowe v spełniają warunek wstępny α i określony jest wynik $v' = K_{\mathfrak{A}}(v)$ to końcowy stan pamięci v' spełnia warunek β.Mówimy krótko, formuła β jest następnikiem pary: formuła α i program K.

(2) Dla dowolnej formuły δ jeśli δ jest następnikiem pary α i K, to implikacja $(\beta \Rightarrow \delta)$ jest formułą prawdziwą w strukturze $\mathfrak{A}$.

Przykład

Przykład 7.3. Rozważmy następujący program M

```
begin
    z:=x; y:=1;
    while z − y ≥ 0
    do
        z:=z-y;
        y:=y+2
    done
end
```

in the structure of the real numbers $\mathbb{R}$ with the usual interpretation of the symbols $+, -, 0, 1, 2, =, \geq$.
The variable y takes on the values of the consecutive odd numbers. As a result, after execution of the ith iteration, the values of variables z, y are, respectively,
$v(x) - 1 - 3 - 5 - ... - (2i-1)$ and $(2i+1)$.
Since $\sum_{0 \leq j \leq i} (2j-1) = i^2$,
the value of variable z after the ith iteration is $v(x) - i^2$. The instruction "while" is executed until the difference $v(x) - i2 - (2i+1)$ is less than zero, i.e. when we find a natural number n, such that
$(n+1)^2 > v(x)$ and $n^2 < v(x)$.
The value of the variable z is equal to $v(x) - \left\lfloor \sqrt{v(x)} \right\rfloor^2$, and the value of y is $2\left\lfloor \sqrt{(x)} \right\rfloor + 1$. Briefly, if the condition $v(x) > 0$ holds for the initial data v, then the program M has a finite, successful computation, and the value of variable z is the distance from the greatest integer square number less than $v(x)$.
The formula
$\beta \equiv (z = x - \lfloor \sqrt{x} \rfloor^2) \wedge (y = 2\lfloor \sqrt{x} \rfloor + 1) \wedge x > 0$
is the strongest postcondition of the formula a ≡ x>0 with respect to program M in the data structure considered.
Indeed, if $\mathbb{R}, v \models x > 0$, then, after execution of the program M, the valuation $M_{\mathbb{R}}(v)$ satisfies the formula β from the above analysis. The condition (1) of the definition is thus satisfied.
Let δ be a formula such that for any valuation v,
$\mathbb{R}, v \models a$ and $v \in Dom(M_R)$ implies $\mathbb{R}, M_{\mathbb{R}}(v) \models d$.
Consider any valuation v' and let R,v'$\models$ b. Then
$v'(y) = 2\left\lfloor \sqrt{v'(x)} \right\rfloor^2 + 1$,
$v'(z) = v'(x) - \left\lfloor \sqrt{v'(x)} \right\rfloor^2$,
$v'(x) > 0$.

Hence $\mathbb{R}, v' \models \alpha$, and therefore the program M has a finite computation at the initial valuation v', i.e. $v' \in Dom(M_{\mathbb{R}})$. Using our assumption, we then have $\mathbb{R}, M_{\mathbb{R}}(v') \models (\delta \wedge \beta)$. Since the behaviour of the program M depends only on the value of the variable x, which, in fact, does not change during the computation, the resulting valuation $M_{\mathbb{R}}(v')$ may differ from the initial valuation only on variables y or z. According to the above analysis, the values of variables z, y after execution of the program M satisfy condition β. As a consequence, $M_{\mathbb{R}}(v') = v'$ and $\mathbb{R}, v' \models \delta$ and finally $\mathbb{R}, v' \models (\beta \Longrightarrow \delta)$ for all valuations v', i.e. $\mathbb{R} \models (\beta \Longrightarrow \delta)$.

7.7. Pewna własność drzew binarnych poszukiwań. W algorytmicznej teorii kolejek priorytetowych należy udowodnić, że ...

Ćwiczenia

7.1. Udowodnij, dla każdej pary liczb $a > 0$ i $\epsilon > 0$ wartości zmiennej x obliczone przez programy 16 i 17 są rôwne.

7.2. Udowodnij, że w ciele liczb rzeczywistych prawdziwa jest następująca implikacja

$$\left\{\begin{array}{l} t := x; \\ \textbf{while } t \leq y \textbf{ do} \\ \quad t := t + x \\ \textbf{od} \end{array}\right\} (x > y) \Rightarrow \left\{\begin{array}{l} u := z; \\ \textbf{while } u \geq \epsilon \textbf{ do} \\ \quad u := u/2 \\ \textbf{od} \end{array}\right\} (u < \epsilon) \tag{43}$$

ROZDZIAŁ 8

Rachunek programów

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

- czy podane wcześniej aksjomaty i reguły wnioskowania o programach stanowią kompletny zestaw narzędzi? Problem ten zajmował nas od poczatku. Kilka prac zawiera twierdzenie o pełności rachunku programów [Mir71, Mir77, BKR91, Kre77a]
- czy aksjomaty rachunku programów opisują semantykę programów w sposób jednoznaczny?
- jakie wnioski można wyciągnąć z twierdzenia o pełności rachunku programów tj. logiki algorytmicznej?

Programy, algorytmy stanowią ważny składnik języka jakim posługują się programiści i matematycy. Od tysiącleci algorytmy są użytecznymi narzędziami. By nie rozwodzić się długo, zauważmy, że Urząd Patentowy USA rejestruje zgłoszenia patentowe algorytmów. Ma to swój wymiar finansowy. Algorytmy są częścią wypowiedzi, np. są cytowane. Algorytmy są także przedmiotem wypowiedzi, np. w zdaniu oznajmującym o poprawności danego programu. Matematycy mają doczynienia z algorytmami od tysięcy lat. Nie odczuwano potrzeby traktowania algorytów w sposób równorzędny z wyrażeniami zdaniowymi i/lub nazwowymi. Mamy do czynienia z pewną niekonsekwencją ... Rola algorytmów nie była dostatecznie uwypuklana.

Programiści pojawili się nie tak dawno. Zdążyli jednak zgromadzić olbrzymi zasób doświadczeń.

Obie partie traktują algorytmy w sposób intuicyjny. Mieszając język i metajęzyk. W szczególności, dostrzegamy brak precyzji w przechodzeniu pomiędzy wyrażeniami i semantyką.

1. Rachunek programów

Rachunek programów jest rachunkiem logicznym (zob. Rys. 1 str. xviii) będącym rozszerzeniem rachunku predykatów. Będziemy używać zamiennie nazw: logika algorytmiczna i rachunek programów.

Rachunek programów AL to zbiór systemów formalnych cechujących się wspólnym schematem. Na każdy taki system $\mathcal{S}$ składają się język $\mathcal{L}$ i operacja konsekwencji syntaktycznej (logicznej) $\mathcal{C}$. System $\mathcal{S} = \langle \mathcal{L}, \mathcal{C} \rangle$ nazywać będziemy systemem dedukcyjnym. Wszystkie języki rachunku programów (mówimy też języki algorytmiczne) mają podobną strukturę. Na zbiór $\mathcal{WFF}$ wyrażeń poprawnie zbudowanych (ang. well formed formulas) składają się zbiór termów $\mathcal{T}$, zbiór formuł $\mathcal{F}$ i zbiór programów $\mathcal{P}$.

$$\mathcal{WFF} = \mathcal{T} \cup \mathcal{F} \cup \mathcal{P}$$

Każdy język rachunku ma tę samą gramatykę. Różnica pomiędzy dwoma językami algorytmicznymi może wynikać z przyjęcia innego alfabetu. Alfabet $\mathcal{A}$ czyli zbiór symboli atomowych dzieli się na podzbiory według wspólnego schematu: zbiór V zmiennych, zbiór Φ funktorów, zbiór R predykatów, zbiór spójników logicznych, zbiór operatorów programotwórczych, zbiór symboli pomocniczych Aux.

$$\mathcal{A} = \{V \cup \Phi \cup R \cup Aux\}$$

Oprócz systemów dedukcyjnych rozpatrywane są też teorie. Teoria $\mathcal{T} = \langle \mathcal{L}, \mathcal{C}, \mathcal{A} \rangle$ to trójka, na którą składają się język $\mathcal{L}$, operacja konsekwencji $\mathcal{C}$ oraz aksjomaty specyficzne dla danej teorii $\mathcal{A}$. Ostatni element trójki $\mathcal{A}$ to zbiór formuł innych niż aksjomaty logiki. Mówimy aksjomaty specyficzne teorii $\mathcal{T}$.

1.1. Język rachunku programów. W poprzednich rozdziałach wprowadzaliśmy wiele elementów języka. napisz Teraz trzeba dodać kwantyfikatory iteracji...

1.2. Semantyka. Przypominamy znaczenie formuł $K\alpha$ i wprowadzamy znaczenie kwantyfikatów iteracji Wartość formuły algorytmicznej postaci $K\alpha$ gdzie napis K jest oznaczeniem programu, a napis α oznacza formułę wyznacza się według następującego wzoru

$$(K\alpha)_{\mathbb{A}}(v) = \begin{cases} \alpha_{\mathbb{A}}(K_{\mathbb{A}}(v)) & \text{gdy wynik } K_{\mathbb{A}}(v) \text{ programu } K \text{ jest określony,} \\ \mathbf{0} & \text{w przeciwnym przypadku wartością formuły jest fałsz.} \end{cases}$$

Wartości formuł zawierających kwantyfikatory iteracji określa się następującymi wzorami

$$\left(\bigcup K\alpha\right)_{\mathbb{A}}(v) = \sup_{i \in N}\{(K^i\,\alpha)_{\mathbb{A}}(v)\}$$

tj. formuła $\bigcup K\alpha$ przyjmuje wartość prawda wtedy i tylko wtedy, gdy istnieje iteracja K^i programu K taka, że po wykonaniu programu K^i zachodzi formuła α,
oraz

$$\left(\bigcap K\alpha\right)_{\mathbb{A}}(v) = \inf_{i \in N}\{(K^i\,\alpha)_{\mathbb{A}}(v)\}$$

tj. formuła $\bigcap K\alpha$ przyjmuje wartość prawda wtedy i tylko wtedy, gdy dla każdej iteracji K^i programu K, po wykonaniu programu

K^i zachodzi formuła α.
Zauważ, tautologią rachunku programów jest formuła

(WhIt) $\quad \{\textbf{while } \gamma \textbf{ do } K \textbf{ od}\}\alpha \Leftrightarrow \bigcup\{\textbf{if } \gamma \textbf{ then } K \textbf{ fi}\}(\neg\gamma \wedge \alpha)$

1.3. Aksjomaty rachunku programów.

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

Aksjomaty rachunku programów są tautologiami, tzn. formuły te przyjmują wartość true w każdej strukturze danych i dla każdego wartościowania zmiennych.

Przykład 8.1. Jest aksjomatem formuła

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

Wskaż, który to aksjomat.

1.3.1. Aksjomaty rachunku programów. Aksjomatem rachunku programów jest każda formuła zbudowana według jednego z poniższych schematów.

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 8.2. Reguła wnioskowania (ang. inference rule) to operacja przyporządkowująca pewnemu zbiorowi formuł Z, formułę α ...
Formuły β_i ze zbioru Z są nazywane przesłankami reguły wnioskowania. Formuła α jest wnioskiem, albo konkluzją, reguły wnioskowania. □

Regułę wnioskowania przedstawiamy w postaci formalnego ułamka.

$$\frac{\{\beta_i\}_{i \in I}}{\alpha}$$

Reguły wnioskowania rachunku programów prowadzą od tautologii w przesłankach do tautologii – wniosku. Poniżej wyliczamy podstawowe reguły wnioskowania rachunku programów.

W dowodach własności algorytmicznych programów napotkasz jeszcze inne, pomocnicze (wtórne), reguły wnioskowania.

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

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

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

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

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

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

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

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 8.3. 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 8.4. 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 8.2. tu przykłady

2. Kilka przykładów

2.1. Wyrażalność własności semantycznych.

2.2. Dowody - wybrane przykłady.

2.3. Algorytmiczne teorie.

3. Pełność AL

W tym miejscu zastanowimy się nad mnastępującymi 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 8.1. (o pełności rachunku programów)
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 - poprawność nie jest tautologią. (Szok Pitagorejcyków) Ale alg. Euklidesa zatrzymuje sie w strukturze liczb całkowitych i potrafimy to udowodnić.

4. 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 8.5. 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 binarną w zbiorze wartościowań zmiennych.

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

Lemat 8.2. 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. Niech v, v' będa dwoma różnymi wartościowaniami w strukturze $\mathbb{A}$, $v \neq v'$. Istnieje wtedy zmienna z taka, że $v(z) \neq v'(z)$. Jeśli zmienna z jest zmienną zdaniową, to kładąc $\alpha = z$ (lub $\alpha = \neg z$), uzyskujemy $\mathbb{A}, v \models \alpha$ i $\mathbb{A}, v' \models \neg\alpha$.
Jeśli z jest zmienną indywiduową and $v(z) = a$, to kładąc $\alpha = (\tau_a = z)$ (przypomnijmy, struktura danych (A) jest ekspresywna – zob. założenie (1)), otrzymujemy, że $\mathbb{A}, v \models (\tau_a = z)$ i $non\ \mathbb{A}, v' \models (\tau_a = z)$. Łatwiej udowodnić implikację odwrotną. Jesli $v = v'$, tj. gdy dla każdej zmiennej z, $v(z) = v'(z)$, to dla każdej formuły α, $\mathbb{A}, v \models \alpha$ wttw gdy $\mathbb{A}, v' \models \alpha$. □

Definicja 8.6. Struktura semantyczna $\langle A, I, \models \rangle$ jest standardowa jeśli spełnione są następujące warunki dla każdej pary programów $K, M \in P$, dla każdej formuły otwartej $\gamma \in F_o$ i dla każdej zmiennej $x \in V$ i dla dowolnego wyrażenia ω, where $id_{\mathbb{A}}(\gamma) = \{(v, v) : \mathbb{A}, v \models \gamma\}$

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

Lemat 8.3. Jesli $\langle \mathbb{A}, I, \models \rangle$ jest standardową strukturą semantyczną, to dla dowolnego programu M i dla dowolnych wartościowań $v,\ v'$ w strukturze $\mathbb{A}$,

$$(v, v') \in I(M) \text{ wtedy i tylko wtedy gdy } (v, v') \in M_{\mathbb{A}}.$$

tj. relacja $I(M)$ jest równa relacji wyznaczonej przez standardowe obliczenia programu M w strukturze danych $\mathbb{A}$.

Dowód wynika wprostt z przyjętych definicji.

Definicja 8.7. Struktura semantyczna $\langle \mathbb{A}, I, \models \rangle$ jest modelem systemu formalnego $AL(\Pi)$ jeśli

a) dla każdego aksjomatu α z systemu $AL(\Pi)$,zachodzi $\mathbb{A} \models \alpha$,
b) dla każdej reguły r wnioskowania z systemu $AL(\Pi)$, prawdziwość wszystkich przesłanek reguły r pociąga prawdziwość konkluzji w tej regule.

Następujące twierdzenie wynika bezpośrednio z lematu 8.3 i z twierdzenia o pełności ??.

Twierdzenie 8.4. Każda standardowa struktura semantyczna jest modelem systemu formalnego $AL(\Pi)$ rachunku programów (tj. logiki algorytmicznej).

W pozostałej części tego podrozdziału będziemy dowodzić implikacji odwrotnej badając jak spełnianie aksjomatów rachunku programów wymusza okreslone własnośći interpretacji I.

Lemat 8.5. Jeśli struktura semantyczna $\langle \mathbb{A}, I, \models \rangle$ jest modelem wszystkich formuł postaci wymienionej w aksjomacie Ax16

$$\text{Ax16}: \quad M(\alpha \wedge \beta) \;\equiv\; (M\alpha \wedge M\beta)$$

gdzie wyrażenia α, β są dowolnymi formułami algorytmicznymi, to dla kazdego programu $M, I(M)$ jest funkcją częściową na zbiorze W wszystkich wartościowań w ten sam zbiór.

Dowód. Załóżmy, że dla pewnych wartościowań v, v', v'' mamy $(v, v'') \in I(M)$ i $(v, v') \in I(M)$ oraz $v'' \neq v'$, . Wtedy z lematu 8.2 wynika, że istnieje formuła α_0 taka, że $A, v' \models \alpha_0$ i $A, v'' \models \neg\alpha_0$. A więc $A, v \models M\alpha_0$ i $A, v \models M\neg\alpha_0$. Stosując aksjomat Ax16 otrzymujemy $A, v \models M(\alpha_0 \wedge \neg\alpha_0)$, sprzeczność. W ten sposób wykazaliśmy, że dla każdego warościowania v, istnieje conajwyżej jedno wartościowanie v' takie, że $(v, v') \in I(M)$, oznacza to, ze dla każdego programu M, jego interpretacja $I(M)$ jest funkcją częściową. □

Lemat 8.6. Jeśli struktura semantyczna $\langle \mathbb{A}, I, \models \rangle$ jest modelem dla aksjomatów Ax16 i Ax19

$$\text{Ax19} \quad \textbf{begin } K; M \textbf{ end } \alpha \iff K(M\alpha),$$

to dla każdej pary programów K i M,

$$I(begin\ K; M\ end) = I(K) \circ I(M),$$

tj. interpretacja złozenia programów K i M, jest superpozycją interpretacji $I(K)$ oraz $I(M)$.

Dowód. Niech $(v, v') \in I(\textbf{begin } K; M \textbf{ end})$ i $A, v' \models \alpha$. Wtedy

$$A, v \models \textbf{begin } K; M \textbf{ end } \alpha\,,$$

i na mocy aksjomatu Ax19, $A, v \models K(M\alpha)$. Z definicji struktury semantycznej wynika, że istnieją wartościowania v_2, v_3 takie, że $(v, v_2) \in I(K), (v_2, v_3) \in I(M)$ oraz $A, v_3 \models \alpha$. Na mocy lematu 8.5, wartościowania v_2 i v_3 są wyznaczone jednoznacznie. niezależnie od formuły α. Możemy powtórzyc ten argument dla dowolnej formuły β, i dowieść, że $A, v' \models \beta$ implikuje $A, v_3 \models \beta$. Tak więc, z lematu 8.2, otrzymamy $(v, v_2) \in I(K), (v_2, v') \in I(M)$ i $v_3 = v'$. W konsekwencji $(v, v') \in I(K) \circ I(M)$. Zawieranie w drugą stronę dowodzimy w podobny sposób. □

Lemat 8.7. Jeśli struktura semantyczna $\langle \mathbb{A}, I, \models \rangle$ jest modelem aksjomatów Ax19, Ax16 i Ax20

$$\text{Ax20}: \quad \textbf{if } \gamma \textbf{ then } K \textbf{ else } M \textbf{ fi}\alpha \iff (\gamma \wedge K\alpha) \;\vee\; (\neg\gamma \wedge M\,\alpha),$$

to dla dowolnych programów K, M i dla dowolnej formuły otwartej γ,

$$I(\textbf{if } \gamma \textbf{ then } K \textbf{ else } M \textbf{ fi}) \;=\; id_{\mathbb{A}}(\gamma) \circ I(K) \;\cup\; id_{\mathbb{A}}(\neg\gamma) \circ I(M).$$

Lemat 8.8. Jeśli struktura semantyczna $\langle \mathbb{A}, I, \models \rangle$ jest modelem aksjomatów Ax16, Ax19, Ax20 i

$$\text{Ax21}: \quad \textbf{while } \gamma \textbf{ do } K \textbf{ od}\,\alpha \Longleftrightarrow ((\neg\gamma \wedge \alpha) \vee (\gamma \wedge K(\textbf{while } \gamma \textbf{ do } K \textbf{ od}\,\alpha))),$$

to dla dowolnej formuły otwartej γ i dla dowolnego programu M zachodzi

$$\bigcup_{i\in N} I((\textbf{if } \gamma \textbf{ then } M \textbf{ fi})^i) \circ \; id_{\mathbb{A}}(\neg\gamma) \;\subset\; I(\textbf{while } \gamma \textbf{ do } M \textbf{ od}).$$

.

Dowód. Załóżmy, że $(v, v') \in \bigcup_{i\in N} I((\textbf{if } \gamma \textbf{ then } M \textbf{ fi})^i) \;\circ\; id_{\mathbb{A}}(\neg\gamma)$. Wynika stąd, że istnieje liczba naturalna m taka, że $A, v' \models \neg\gamma$ i $(v, v') \in I((\textbf{if } \gamma \textbf{ then } M \textbf{ fi})^m)$.

Jeśli, dla pewnej formuły α, zachodzi $A, v' \models \alpha$, to zachodzi też $A, v \models (\textbf{if } \gamma \textbf{ then } M \textbf{ fi})^m(\neg\gamma \wedge \alpha)$ i z prawdziwości aksjomatu Ax21 w naszej strukturze semantycznej, mamy

$$A, v \models \textbf{while } \gamma \textbf{ do } M \textbf{ od } \alpha.$$

Wynika stąd, że istnieje wartościowanie v'', które jest wyznaczone jednoznacznie (zob. lemat 8.5) i takie, że $(v, v'') \in I(\textbf{while } \gamma \textbf{ do } M \textbf{ od})$ and $A, v'' \models \alpha$. Ponieważ powyższy argument może byc powtórzony dla dowolnej formuły α, to na mocy lematu 8.2 otrzymujemy równość $v' = v''$. A więc

$$(v, v') \in I(\textbf{while } \gamma \textbf{ do } M \textbf{ od}).$$

□

Lemat 8.9. Jeśli struktura semantyczna $\langle \mathbb{A}, I, \models \rangle$ jest modelem aksjomatów Ax16, Ax19, Ax20, Ax21 i jeśli reguła wnioskowania R3 jest poprawna w tej strukturze, to

$$I(\textbf{while } \gamma \textbf{ do } K \textbf{ od}) = \bigcup_{i\in N} I((\textbf{if } \gamma \textbf{ then } K \textbf{ fi})^i) \circ \; id_{\mathbb{A}}(\neg\gamma).$$

.

Dowód. Niech $(v, v') \in I(\textbf{ while } \gamma \textbf{ do } M \textbf{ od})$ i niech $A, v' \models \alpha$. Załóżmy, że $x_1, x_2, ..., x_k$ są wszystkimi zmiennymi indywiduowymi i $q_1, ..., q_l$ są wszystkimi zmiennymi zdaniowymi, które występują w formule $\textbf{while } \gamma \textbf{ do } M \textbf{ od}\,\alpha$. Przyjmijmy ponadto, że $v(x_j) = a_j$ dla $1 \leq j \leq k$. Niech β będzie formułą opisującą wartościowanie v zmiennych $x_1, x_2, ..., x_k, q_1, ..., q_l$, np.

$$\beta = (x_1 = a_1) \wedge (x_2 = a_2) \wedge ... \wedge (x_k = a_k) \wedge q_1' \wedge ... \wedge q_l',$$

gdzie podformuła q_i' oznacza q_i jeśli $v(q_i) = \mathbf{1}$ a podformuła q_i' jest postaci $\neg q_i$ jeśli $v(q_i) = \mathbf{0}$. A więc mamy $A, v \models \beta$ i $A, v \models \textbf{while } \gamma \textbf{ do } M \textbf{ od}\,\alpha$ i, wobec tego, $A, v \not\models \textbf{while } \gamma \textbf{ do } M \textbf{ od}\,\alpha \Rightarrow \neg\beta)$. Stosując regułę R3, stwierdzamy istnienie takiej liczby

naturalnej m, że $nonA \models$ (**if** γ **then** M **fi**)$^m(\alpha \wedge \neg\gamma)\ \neg\beta$. A więc istnieje wartościowanie v'' takie, że $A, v'' \models$ (**if** γ **then** M **fi**)$^m(\alpha \wedge \neg\gamma)$ i $A, v\text{“} \models \beta$, skąd wynika, na mocy przyjętej definicji formuły β, że $A, v \models$ (**if** γ **then** M **fi**)$^m(\alpha \wedge \neg\gamma)$. Przyjmijmy, że m jest najmniejszą liczbą naturalną o tej własności. A więc istnieje wartościowanie v'', takie, że $(v, v\text{“}) \in I$(**if** γ **then** M **fi**)^m i $A, v\text{“} \models (a \wedge \neg g)$.

Udowodnimy teraz, że dla dowolnehj formuły δ, $A, v' \models d$ implies $A, v\text{“} \models (\delta \wedge \neg\gamma)$. Powtarzając powyższe rozumowanie uzyskujemy $A, v \models$ **if** γ **then** M **fi**)$^j(\delta \wedge \neg\gamma)$ dla pewnej liczby naturalnej j. Z założenia o liczbie m i z aksjomatu Ax20 wynika, że $m \leq j$. A więc $A, v \models$ **if** γ **then** M **fi**)$^m(\delta \wedge \gamma)$. Na mocy lematu 8.5 i definicji wartościowania v'', otrzymujemy $A, v\text{“} \models (\delta \wedge \neg\gamma)$.

Udowodniliśmy więc, że istnieje liczba naturalna m i wartościowanie v'' takie, że $(v, v'') \in I$**if** γ **then** M **fi**)^m i dla dowolnej formuły δ, jeśli $A, v' \models \delta$ to $A, v'' \models (\delta \wedge \neg\gamma)$. Z lematu 8.2 $v'' = v'$ a więc

$$(v, v') \in I(\textbf{if } \gamma \textbf{ then } M \textbf{ fi})^m \ \circ id_{\mathbb{A}}(\neg\gamma).$$

□

Lemat 8.10. Jeśli struktura semantyczna $\langle \mathbb{A}, I, \models \rangle$ jest modelem dla aksjomatu Ax18

$$\text{Ax18}: \quad (x := \tau)\gamma(x) \ \equiv \ \gamma(x/\tau) \wedge (\tau = \tau)(q := \gamma)\gamma(q) \ \equiv \ \gamma(q/\gamma),$$

, to dla każdej zmiennej indywiduowej x i dla dowolnego termu τ określony jest wynik $I(x := \tau) = \{(v, v') : \tau_{\mathbb{A}}(v)$ i $v'(x) = \tau_{\mathbb{A}}(v)$, oraz $v'(z) = v(z)$ for $z \neq x\}$. pamietaj o termach czesciowych, czy je tu mamy?? i dla każdej zmiennej zdaniowej q oraz dowolnej formuły otwartej γ' zachodzi $I(q := g') = \{(v, v') : \ v'(q) = g_A(v)$ i $v'(z) = v(z)$ dla $z \neq q\}$.

Dowód. Niech $(v, v') \in I(x := t), v'(x) = a$ i $v'(z) = b$ 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 $v\text{“}$ such that

$(v, v\text{“}) \in I(x := t)$ and$A, v\text{“} \models (x = ta) \ \wedge \ (z = tb)$,

i.e.
$v\text{“}(x) = tav\text{“}(z) = tb$.
From our assumptions, we have $v\text{“}(x) = v'(x)$ and $v\text{“}(z) = v'(z)$. Since z was an arbitrary variable such that $z \neq x$, we can deduce that $v\text{“} = v'$, 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. □

Z powyższych faktów wynika natychmiast główny wynik tego podrozdziału.

Twierdzenie 8.11. Każda struktura semantyczna, która jest modelem formalnego systemu rachunku programów jest strukturą standardową.

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 8.2 axioms of algorithmic logic AL(Π) uniquely determine 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!!!!!

5. Definicje

Kolejne narzędzia programowania to bloki, funkcje, procedury i klasy. Zanim przejdziemy do omawiania tych pojęć, konieczne jest zrozumienie czym one są. Przyda się nam analiza pojęcia definicji w językach pierwszego rzędu.

- W języku $\mathcal{L}_5$ można zaprogramować każdą funkcję obliczalną (teza Turinga-Churcha). Dlaczego więc w językach programowania występuja deklaracje funkcji, procedur, klas? Jaką rolę odgrywają, np. deklaracje funkcji? Niewątpliwie są użyteczne. Czy jednak właściwie pojmujemy jak się nimi posługiwać? Pragniemy zwrôcić Twoją uwagę na potrzebę dyscypliny we wprowadzaniu deklaracji.
- W logice matematycznej omawia się pojęcie definicji i ustala sie kilka ważnych faktôw. O czym piszemy poniżej. Przyjmuje się, że definicją jest para wyrażeń

  $$\langle definiendum\rangle\{\overline{\overline{\Leftrightarrow}}\}\langle definiens\rangle$$

  oddzielonych znakiem rôwnoważnosci lub znakiem rôwności.
  Definiendum czyli wyrażenie definiowane, ma postać albo formuły atomowej $\rho(x_1, \ldots, x_n)$ albo termu elementarnego $\phi(x_1, \ldots, x_m)$. Symbol ρ jest nowym predykatem,

tj. oznaczeniem nowo wprowadzanej relacji. Podobnie, symbol ϕ jest nowym, nie występującym do tej pory w języku teorii $\mathcal{T}$ oznaczeniem nowej operacji.

Jawna, lub nieuwikłana, definicja relacji. Jawna, lub nieuwikłana definicja relacji jest równoważnością

$$\rho(x_1, \ldots, x_n) \Leftrightarrow \alpha(x_1, \ldots, x_n)$$

Nowy predykat ρ definiowany jest przez formułę $\alpha(x_1, \ldots, x_n)$, która ma dokładnie te same zmienne wolne $x_1, \ldots, x_n$, jakie występują po lewej stronie definicji i ponadto, w formule $\alpha(x_1, \ldots, x_n)$ nie występuje symbol definiowany ρ.

Niejawna, lub uwikłana, definicja relacji.

Jawna definicja funktora. Napis postaci

$$\varphi(x_1, \ldots, x_n) = \tau(x_1, \ldots, x_n)$$

jest definicją nowego funktora jeśli 1) symbol φ nie występował dotąd w języku rozważanej teorii, 2) wyrażenie $\tau(x_1, dots, x_n)$ jest termem (wyrażeniem nazwowym) i występują w nim dokładnie tesame zmienne $x_1, \ldots, x_n$ co po lewej stronie równości.

Niejawna definicja funktora. Definiendum czyli wyrażenie definiujące (jest to odpowiednio, formuła lub term), jest wyrażeniem o takim samym zbiorze zmiennych wolnych.$x_1, \ldots, x_n$ jest wyrażeniem w jednej z trzech postaci:

(i) formuła $\alpha(x_1, \ldots, x_n)$
(ii) formuła $\forall_{x_1} \ldots \forall_{x_n} \exists_y \beta(x_1, \ldots, x_n, y)$ jest twierdzeniem teorii $\mathcal{T}$. ?
(iii) term $\tau(x_1, \ldots, x_m)$

- W rachunku programów jest trochę inaczej ponieważ definiens może być formułą algorytmiczną tj. w definicji może pojawić się algorytm. Pomiędzy jawną i niejawną definicją funktora można umieścić wiele różnych postaci definicji

$\phi(x_1, ..., x_n) \stackrel{df}{=} K\tau$, w termie $K\tau$ nie występuje symbol definiowany ϕ.
... j.w. ale dopuszczalne jest by po prawej stronie definicji występował symbol definiowany
dla relacji $\rho(x_1, ..., x_n) \stackrel{df}{\equiv} M\alpha$ podobnie
jawna definicja predykatu ρ wymaga by po prawej stronie nie występował ten symbol,
niejawna definicja ...
Sformułować warunki konieczne by zachodziły twierdzenia o istnieniu modelu i o nieistotności definicji. Podać kontrprzykłady.

- xxx ...

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 (tzn. 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 $\mathcal{C}$ zachowuje wszystkie schematy aksjomatów logicznych 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 nieistotności 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 [Kle36].

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 [Wan78] i M. O'Donnell[O'D82].
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 [Kal55], ...

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 własnością 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.

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

tu przenieś z sekcji funkcje I

Ćwiczenia

8.1. Udowodnij powyższy lemat 8.7.

8.2. Czy potrafisz wyprowadzič to twierdzenie z aksjomatów AL podanych poniżej?

8.3. Narysuj diagramy formuły $\{\textbf{while } \gamma \textbf{ do } K \textbf{ od}\}\alpha$ oraz formuły $\bigcup\{\textbf{if } \gamma \textbf{ then } K \textbf{ fi}\}(\neg\gamma \wedge \alpha)$. Porównaj.

8.4. Porównaj formułę WhIt z wzorem podanym przez Dijkstrę w [Dij78] str. 36 (oraz 40) i zechciej skomentowač.

TODO
Połączyć trzy następne rozdziały w jeden: Funkcje, procedury i bloki ?

ROZDZIAŁ 9

$\mathcal{L}_6$ Bloki

$$\mathcal{L}_0 \subsetneq \mathcal{L}_1 \subsetneq \mathcal{L}_2 \subsetneq \mathcal{L}_3 \subsetneq \mathcal{L}_4 \subsetneq \mathcal{L}_5 \subsetneq \mathcal{L}_6 \subsetneq \mathcal{L}_7 \subsetneq \mathcal{L}_8 \subsetneq \mathcal{L}_9 \subsetneq \mathcal{L}_{10} \subsetneq \mathcal{L}_{11} \subsetneq \mathcal{L}_{12}$$

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 pewne instrukcje $\mathbb{I}$ wykorzystują dodatkowe zasoby (wprowadzane przez lokalne deklaracje) $\mathbb{D}$,

  block $\mathbb{D}$ begin $\mathbb{I}$ end
- 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).
- Czasami trzeba wykonać pewien algorytm w środowisku zadeklarowanym przez klasę o nazwie K. Stosujemy wtedy instrukcję bloku prefiksowanego

  pref K block $\mathbb{D}$ begin $\mathbb{I}$ end.

  O tej instrukcji napiszemy obszerniej w następnej części Programuj z klasą.

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 ciągu 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 rozszerzają wspólne środowisko w inny sposób.

Przykład 9.1. W poniższym programie zawarty jest moduł bloku

```
program SWAPzBlokiem;
   const k= -12 , l=35 ;
   var x = 0
   var y = 0
   var t = 0
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
```

Zauważ, wewnątrz bloku występują, zmienna t – lokalna i dwie zmienne x i y ź zewnątrz-- nielokalne.
Nielokalna zmienna t nie jest dostępna wewnątrz bloku.

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

Przykład 9.2.

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

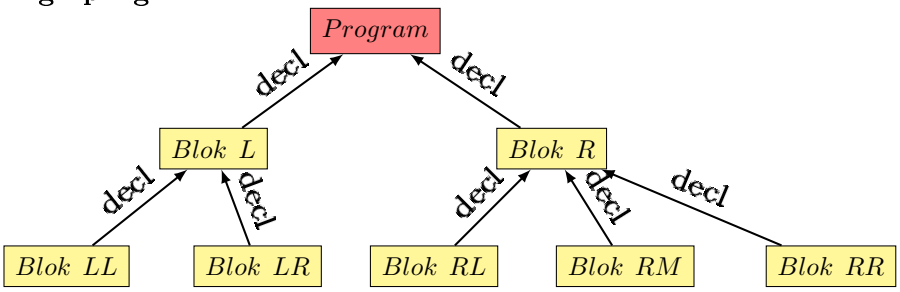
W niektórych językach programowania każda instrukcja for jest takim blokiem.

Uwaga. Oprócz zmiennych, deklaracja bloku może wprowadzać lokalne deklaracje funkcji, procedur i klas.

Struktura modułów programu. Programy w języku $\mathcal{L}_6$ mają strukturę drzewa. Węzłami są bloki. Relacja bezpośredniego

zawierania się jednego bloku w drugim będzie oznaczana strzałką $\xleftarrow{decl}$. Blok programu Main jest korzeniem drzewa.

Przykład 9.3. Rysunek przedstawia strukturę bloków pewnego programu P.



W następnych rozdziałach, zwłaszcza w rozdziale ?? Dziedziczenie, struktura modułów okaże się bardziej skomplikowana.

2. Komputer $\mathcal{K}_6$

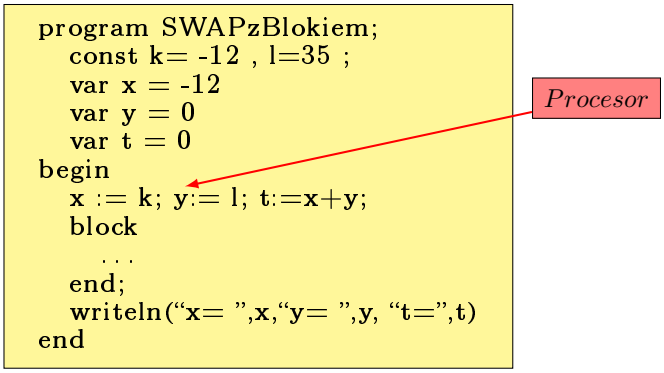
Wykonanie instrukcji bloku polega tym, że do zbioru jednostek dynamicznych dodawana jest nowa jednostka o. Struktura jednostko o jest podobna do struktury rekordu aktywacji programu głównego Main.
Ale taka jednostka jest wyposażona w strzałkę SL, która przydaje się gdy wykonanie instrukcji napotyka na zmienną nielokalną. Rekord aktywacji bloku wyposażony jest w jeszcze jedną strzałkę DL.

Przykłady
Popatrzmy jak przebiega wykonanie tego programu. Stan początkowy, to rekord aktywacji programu.

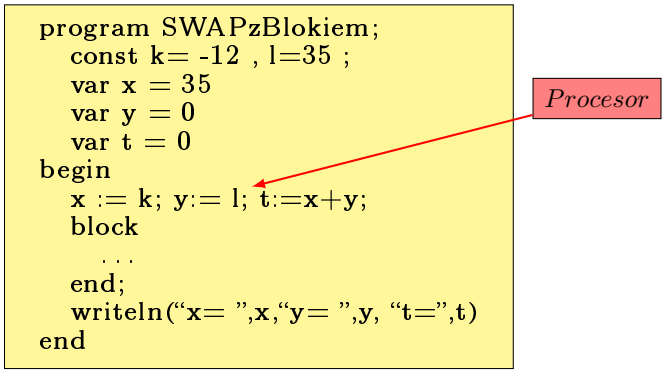
```
program SWAPzBlokiem;
   const k= -12 , l=35 ;
   var x = 0
   var y = 0
   var t = 0
begin
   x := k; y:= l; t:=x+y;
   block
      ...
   end;
   writeln("x= ",x,"y= ",y, "t=",t)
end
```

Procesor

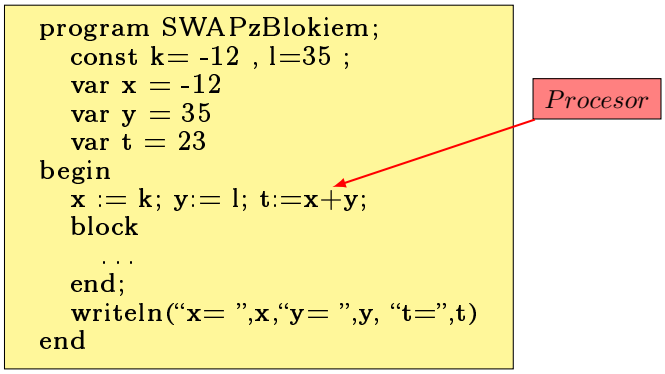
Następny stan, po wykonaniu polecenia, x:=k



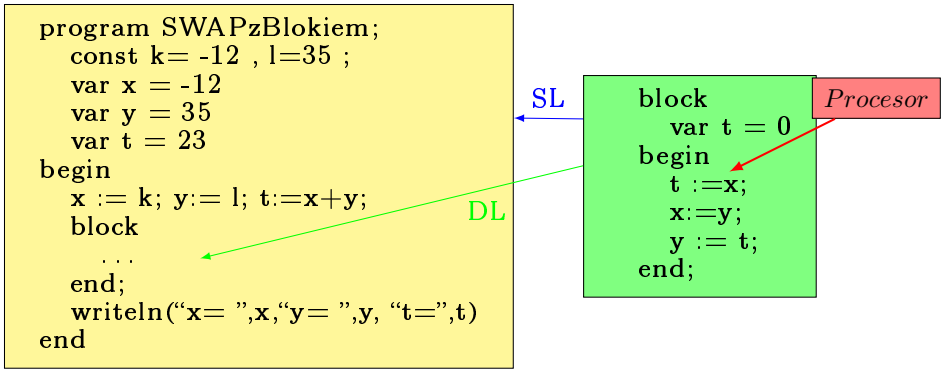
Następny stan, po wykonaniu polecenia y:=l



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

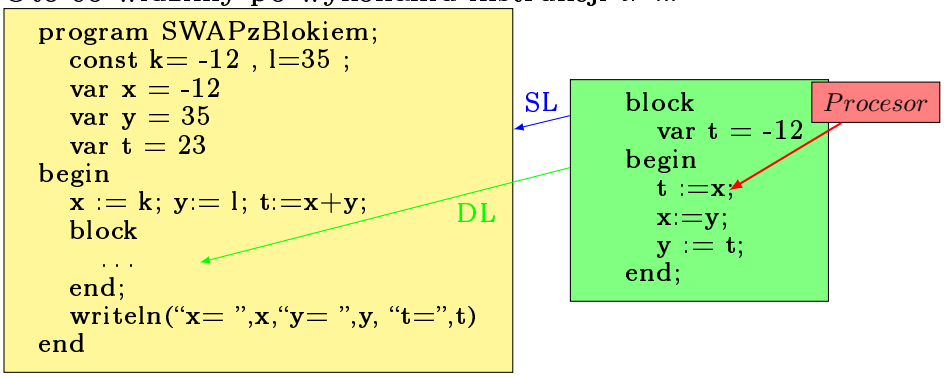


Następny stan, rozpoczynamy wykonywanie instrukcji bloku

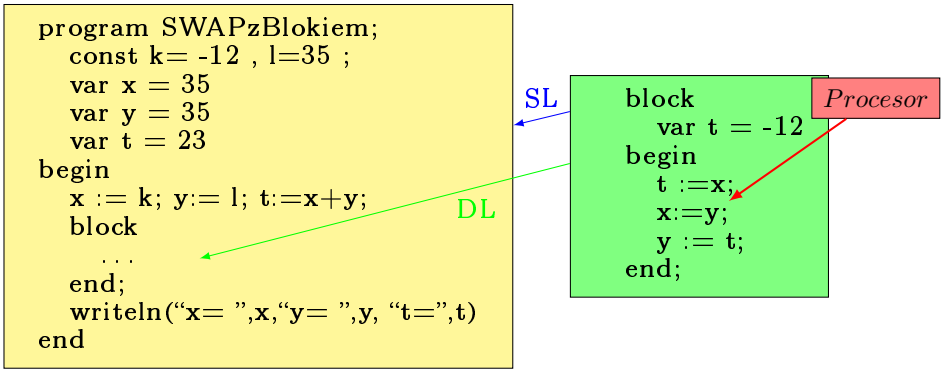


Na rysunku widać, że komputer utworzył nową jednostkę dynamiczną (jej struktura jest podobna do rekordu aktywacji programu Main). Rekord aktywacji bloku jest wyposażony w dwie strzałki SL i DL. Ich znaczenie okaże się za chwilę. Procesor zaczyna wykonywanie instrukcji bloku.

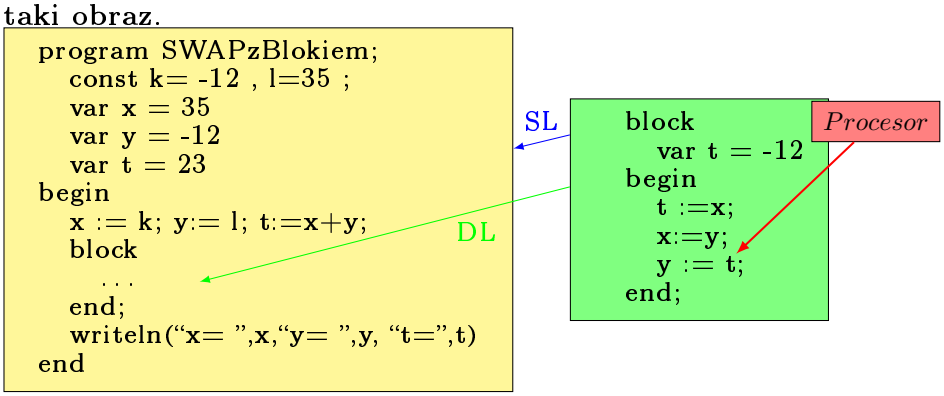
Przystępujemy do wykonania pierwszej instrukcji. Gdzie jest zmienna t? Procesor szuka zmiennej t w bloku i znajduje. Zmienna x? Nie ma jej wśród zmiennych zadeklarowanych w bloku – szukamy dalej przechodząc po strzałce SL. Znaleziono. Oto co widzimy po wykonaniu instrukcji t:=x.



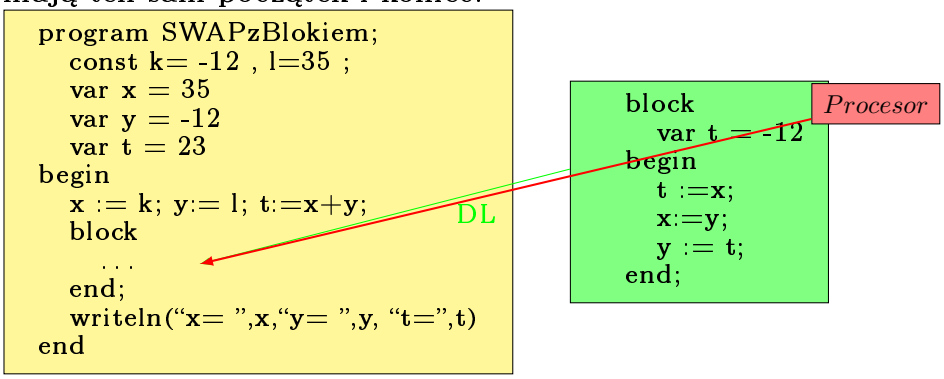
Teraz druga instrukcja. Łatwo się domyślić co się dzieje. Zmiana dokonuje się w rekordzie aktywacji Main. Oto co widzimy po wykonaniu instrukcji x:=y.



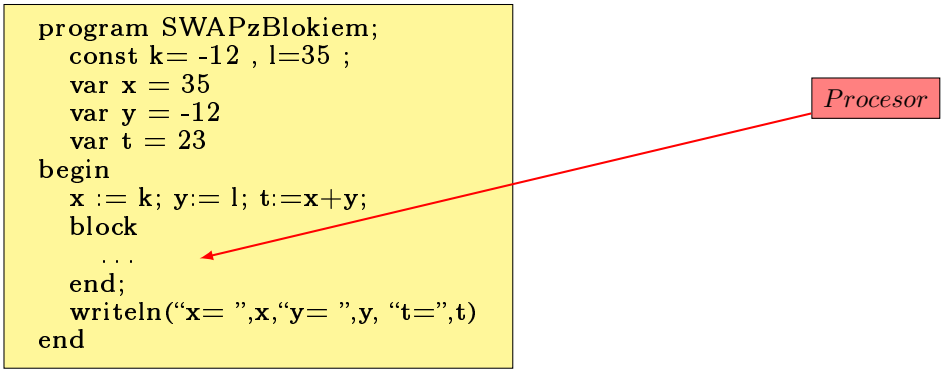
Ostania instrukcja bloku. Po wykonaniu polecenia y:=t widzimy taki obraz.



Zakończono wykonywanie bloku. Strzałka SL staje się zbędna. Gdzie należy kontynuować? Procesor przesuwamy wzdłuż strzałki DL. Można usunąć rekord aktywacji bloku. 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.



Teraz można wydrukować wyniki.



Zapamiętajmy:

- Wszystkie moduły zawarte w programie to moduły bloku. (Program też jest blokiem.)
  Rekordy aktywacji to rekordy aktywacji bloku.
- procesor wskazuje na jeden rekord aktywacji: rekord aktywacji programu głównego Main lub rekord aktywacji jakiegoś bloku.
- SL – ta strzałka wskazuje gdzie jest rozszerzenie pamięci.
- DL – ta strzałka wskazuje procesorowi gdzie należy kontynuować wykonywanie poleceń po wypełnieniu wszystkich zadań z bieżącego rekordu aktywacji.
- bind – funkcja przyporządkowująca wystąpieniu zmiennej x w instrukcji I zawartej w module m programu taki moduł m' tego programu, który zawiera deklarację zmiennej x i jest przy tym najmniejszym modułem zawierającym moduł m.

$$bind(x,m) \stackrel{df}{=} \begin{cases} m' = Decl^i(m) & \text{przy czym } i \text{ jest najmniejszą liczbą taką,} \\ & \text{że moduł } Decl^i(m), \text{zawiera deklarację} \\ & \text{zmiennej } x, \\ \text{BŁĄD} & \text{tj. program jest niepoprawny} \\ & \text{zmienna } x \text{ niezadeklarowana w żadnym} \\ & \text{bloku obejmującym blok } m. \end{cases}$$

  W naszym przykładzie ... Wystarczy zapamiętać ile razy należy przejść do modułu obejmującego dany moduł. Liczba ta zostanie wykorzystana przez komputer $\mathcal{K}_6$ podczas wykonywania programu.
- find – 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.

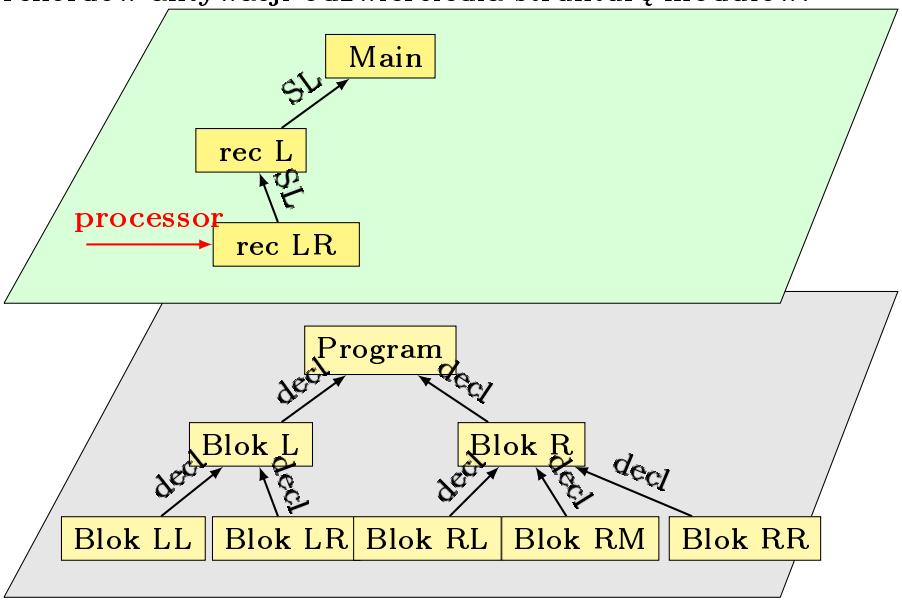
$$find(x, o) \stackrel{df}{=} SL^i(o)$$

Dokładniejszy opis tej funkcji podamy w następnych rozdziałach.

3. Semantyka

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.

Przykład 9.4. Wtrakcie wykonywania programu struktura rekordów aktywacji odzwierciedla strukturę modułów.



Na dolnej (szarej) płaszczyznie umieszczono moduły. Strzałki decl wskazują moduł m' obejmujący dany moduł m. Oznacza to tyle, że zmienna nielokalna w bloku m będzie poszukiwana w bloku $decl(m)$.
Na górnej(zielonej) płaszczyźnie umieszczono rekordy aktywacji. Zauważ, podczas wykonania programu istnieją rekordy aktywacji tylko pewnych modułów.
Main jest rekordem aktywacji bloku głównego tj. programu, rec L jest rekordem aktywacji bloku L, rec LR jest rekordem

aktywacji bloku LR. Na rysunku ujęto chwilę, gdy ten właśnie rekord jest aktywny tzn. procesor wykonuje instrukcje (niewidoczne na rysunku) z rekordu rec LR. □

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

$$\left\{\begin{array}{l}\textbf{block}\\ \quad\textbf{var } x:T\\ \textbf{begin}\\ \quad I(x,z)\\ \textbf{end}\end{array}\right\}\alpha(\mathrm{z}) \Leftrightarrow \left\{\begin{array}{l}\textbf{block}\\ \quad\textbf{var } y:T\\ \textbf{begin}\\ \quad I(x/y,z)\\ \textbf{end}\end{array}\right\}\alpha(\mathrm{z})$$

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.

Analizowanie instrukcji bloku. Podczas analizy instrukcji bloku należy pamiętać o paru zasadach:

- Blok jest instrukcją złożoną.
- Instrukcje bloku posługują się zasobami lokalnymi – zadeklarowanymi w tym bloku, a także zasobami zastanymi – opisanymi i zadeklarowanymi wcześniej. Przyjmijmy, że do analizy otoczenia bloku należy stosować teorię $\mathcal{T}_G = \langle \mathcal{L}, \mathcal{C}, \mathcal{A}\rangle$ Teoria $\mathcal{T}_B = \langle \mathcal{L}_B, \mathcal{C}_B, \mathcal{A}_B\rangle$, która od teorii $\mathcal{T}_B$ różni się w sposób następujący:...
- Globalny efekt instrukcji bloku można opisywać implikacją

$$\mathcal{T} \vdash \alpha \Rightarrow \{\mathbb{I}\}\beta$$

gdzie

-

3.1. Eliminacja bloku. Instrukcja bloku zawarta w zewnętrznym (tj. obejmujacym ją) bloku może być wyeliminowana w ten sposób:

```
block
   const k= -12 , l=35 ;
   var x = 0
   var y = 0
   var t = 0
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
```

```
block
   const k= -12 , l=35 ;
   var x = 0
   var y = 0
   var t = 0
   var t2 = 0
begin
   x := k; y:= l; t:=x+y;
   t2 :=x;
   x:=y;
   y := t2;
   writeln("x= ",x,"y= ",y, "t=",t)
end
```

W ten sam sposób mozna usunąć instrukcję bloku zawartego w każdym innym module,procedury, funkcji, klasy, współprogramu, procesu.
Nie jest to jednak najlepsza metoda. Po pierwsze jesteśmy zobowiązani dokonać kłopotliwej dla naszej pamięci transformacji tekstu. Należy kazdą zmienną zadeklarowaną w bloku wewnętrznym zastąpić nową zmienną jaka nie występuje w bloku zewnętrznym. W naszym przypadku niech to będzie t2. Do deklaracji bloku zewnętrznego należy dodać deklarację nowej zmiennej. Usunąć deklarację zmiennej t oraz słowa block, begin oraz end.
Jeśli nasz program ma więcej bloków wewnętrznych to łatwo popełnić błąd.
Po drugie ta metoda nie daje się zastosować w przypadku gdy mamy do czynienia z instrukcją procedury, zob. następny rozdział.

3.2. Udowodnij lemat. Udowodnij lemat o następującym schemacie

$$\alpha \Rightarrow \{I(x/z)\}\beta$$

gdzie formuła α ...
Pozwoli Ci to na udowodnienie nowej formuły zawierającej instrukcję bloku

$$\delta \Rightarrow \left\{ \begin{array}{l} \textbf{block} \\ \quad \mathbb{D} \\ \textbf{begin} \\ \quad I_1; \\ \quad \textbf{block } var\ x \textbf{ begin } I(x) \textbf{ end}; \\ \quad I_2; \\ \textbf{end} \end{array} \right\} \gamma$$

Przykład...

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

4. Teoria $\mathcal{T}_6$

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? Każdy moduł programu należy rozpatrywać w innej teorii. Jeżeli do tej pory program wprowadził n teorii to nowa deklaracja wprowadza teorię $n+1$ pierwszą. Mamy tu ze zjawiskiem niespotykanym w matematyce: Jeden program może wprowadzać setki teorii. Przy czym wprowadzanie to nie ma miejsca!. Programiści nie są swiadomi tego robią. Pracują intuicyjnie. Na pociechę możemy zauważyć, że teorie zawierają się w sobie tak jak moduły zawierają się w sobie. Z jednej strony struktura modułów programu jest strukturą drzewa.
Rysunek?
Z drugiej strony w każdym momencie (w każdym module) do jego analizy wystarcza nam znajomośc tej gałęzi drzewa modułów która prowadzi od danego modułu (na razie, bloku – ale to się zaraz zmieni) do modułu korzenia tzn. do bloku MAIN.

?? Możemy formalizm podzielić na warstwy: wprowadzając definicje (przez indukcję) kolejnych warstw [trochę to może przypominać 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 hierarchią teorii. W każdym module obowiązuje inny alfabet etc.

Informacja o grafie modułów i relacjach $decl$ oraz inh jest bardzo ważna.

Stosując instrukcję bloku:

- programista zmienia teorię – rozszerza zbiór zmiennych, dodaje nowe definicje , etc.
- można także potraktować tę instrukcję bloku I jako właśnie nową instrukcję

$$\alpha \Rightarrow I\beta$$

jesli to może pomóc to warto udowodnić taki lemat $(\alpha \Rightarrow I\beta)$

- instrukcja bloku umożliwia łatwe i dokładne określenie zbioru zmiennych nieistotnych dla pewnej części programu. Mianowicie instrukcja bloku

```
block
   var x,y,z:T
begin
   M
end
```

pozwala wykonać algorytm M zapewniając przy tym, że wartości tych trzech zmiennych nie wpłyną na działanie innych instrukcji programu, Te trzy zmienne mają wpływ tylko na wykonanie algorytmu M wewnątrz bloku. Powtórzmy, jeśli analiza programu ma odbywać się w teorii $\mathcal{T}$, to analizując działanie bloku pracujemy w nowej bogatszej teorii $\mathcal{T}'$.

ROZDZIAŁ 10

$\mathcal{L}_8$ Funkcje

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 i przejście do nowej teorii. 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. Z drugiej strony różnorodne formy jakie może przyjmować deklaracja funkcji w programowaniu stwarza wiele interesujących możliwości. 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:

$$even(n) \overset{df}{\Leftrightarrow} \exists z(n = z + z)$$

$$n\mathbf{div}2 = y \overset{df}{\Leftrightarrow} (n = y + y) \vee (n = y + y + 1)$$

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: integer
begin
  i:=0;
  while i<n do
    i:=i+2;
  od;
  result := i=n;
end div2
```

Obaj (mniej lub bardziej świadomie) 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 wyrażeniem boolowskim (tj.formułą otwartą (por. 3.2.2), 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 10.1. 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 10.2. 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. 3.2.1), 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 10.2. 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. Jeśli w pewnym module znajdują się dwie deklaracje funkcji o tej samej nazwie to program zawiera błąd sprzeczności. Kompilator wykrywa taki błąd i odrzuca program zawierający podwójną deklarację tej samej funkcji. Inaczej mówiąc, w każdym module poprawnego programu każda zadeklarowana w nim wielkość jest zadeklarowana jeden raz.

1. przykłady

Wykorzystamy poprzednie rozważania by podać kilka przykładów.

Przykład 10.3. Funkcja obliczająca n-tą potęgę liczby x.

```
unit power: function(x,n:integer): integer;
  var z,y,m: integer
begin
  z:=x; y:=1; m:=n;
  while m ≠ 0 do
    if odd(m) then y:= y*z fi;
    m := m div 2;
    z:=z*z
  od ;
  result := y*z;
end power;
```

Uwaga. Z postaci tej deklaracji łatwo widać, że obliczenie zależy tylko od argumentów x i n, nie występują zmienne nielokalne. Taka deklaracja może być przeniesiona w dowolne miejsce programu. Można ją skopiować do dowolnego programu. Deklaracja taka może się bezpiecznie znaleźć w bibliotece procedur.

Wykorzystując rozważania podrozdziału 7.5, str. 146, stwierdzamy: jeżeli obie wartości x i n są dodatnie, to wynikiem jest x^n. Obliczenie wyniku wymaga $\log n$ mnożeń.

Przykład 10.4. Funkcja sqrt obliczająca pierwiatsek kwadratowy liczby x.

```
unit sqrt: function(a, ε: real): real;
  var x: real;
begin
  if a < 0 then raise NegativeArgSqrt fi ;
  x:=(a + 1)/2;
  while (x − a/x) ≥ ε do x:= (x + a/x)/2 od;
  result := x
end sqrt
```

Twierdzenie 16, str. 138, upewnia nas o poprawności wyniku.

Nieco bardziej skomplikowany przykład.

Przykład 10.5. iloczyn macierzy
unit MultMat: function(A,B: arrayof arrayof real): arrayof arrayof real;

end MultMat

2. 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. [RS63] 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 10.6.

```
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 10.7.

```
unit Fc: function(x: integer): Boolean;
begin
  result := Fc(x)
end Fc
```

Dowolna funkcja $Fc(x)$ spełnia równoważność $Fc(x) \Leftrightarrow Fc(x)$.

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.dlaczego? wyjaśnij! 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. 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

Co to znaczy nieuwikłane? objaśń 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 10.3. 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. Tu wstawić parę funkcji wywołujących się nawzajem.

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).
W logice matematycznej definicja funkcji jest albo jawna – i nie zawiera lokalnych definicji, albo jest niejawna tzn. uwikłana – co to oznacza? czy można wtedy wprowadzać wewnętrzne deklaracje? definicje?.

Definicja funkcji jest przyjmowana jako aksjomat definiujący nową operację.
W programowaniu, zwłaszcza w programowaniu obiektowym deklaracja metody w klasie definiuje sposôb obliczania ALE niekoniecznie jest to opis własności metody.

Może okazać się, że znamy deklarcję metody np. push w klasie stosy, ale trudno jest odczytać własności tej metody. Bo np. potrzeba kilku formuł w których wystepuje funkcja push by otrzymać specyfikację czyli aksjomatyzację.

Ponadto, implementacja metody jedna nie wyklucza innego sposobu zaimplementowania metody push.

Podsumujmy. Z tych uwag wynika, że

- W programowaniu z klasami i obiektami konieczne jest stosowanie modułów specification.
  Ponieważ nie można uznać modułu implementującego za pełny opis zamierzenia. Moduł klasy jest implementacją – jedną z wielu. Tak jest w przypadku klas, kolejek FIFO, kontenerôw etc.

- Każdy moduł ma prawo wprowadzenia swoich lokalnych definicji: zmiennych, funkcji, klas ... W konsekwencji struktura programu jest bardziej skomplikowana niż to spotyka się w podstawach matematyki.

  Jest to wyzwanie dla badaczy.

3. Funkcjonały

Proponujemy by deklaracje funkcji w których przynajmniej jeden argument jest funkcją nazywać funkcjonałami i odpowiednio stosować termin functional.
Przykład.
Algorytm bisekcji jest przykładem funkcjonału.

```
unit bisection: functional(a,b,eps:real; function f(x:real):real): real;
  var m:real
begin
  if
...
  fi
end bisection;
```

Ale wartością nie jest funkcja! Pytanie: czy potrafimy stworzyć moduł, którego argumentem jest funkcja f i który zwraca jako wynik nazwę nowej funkcji g? Próbę odpowiedzi przedstawimy w nastepnej części tej książki, w rozdziale 13.

Ćwiczenia

10.1. Ciąg Fibonacciego jest opisany w następujący sposób

$$f_n \stackrel{df}{=} \begin{cases} 0 & \text{gdy } n = 0 \\ 1 & \text{gdy } n = 1 \\ f_{n-2} + f_{n-1} & \text{gdy } n \geq 2 \end{cases}$$

Twoje zadanie polega na porównaniu kilku różnych sposobów obliczania wartości n-tego elementu ciągu Fibonacciego:

- bezpośrednio wykorzystując definicję ciągu – rekursja,
- obliczając kolejno $f_0, f_1, \ldots f_{n-2}, f_{n-1}$ – iteracja,
- wykorzystując wzór Bineta,

Porównaj czasy wykonania Twoich trzech programów dla obliczenia wartości f_{30}.

10.2. Oszacuj koszt każdej z metod.
Czy można obliczyć wartość f_n w czasie logarytmicznym $O(log\, n)$?

10.3. Funkcja Ackermanna. To zadanie jest trudniejsze.
Definicja funkcji Ackermanna

$$Ack(m,n) \stackrel{df}{=} \begin{cases} n+1 & \text{gdy } m = 0 \\ Ack(m-1,1) & \text{gdy } m > 0 \text{ i } n = 0 \\ Ack(m-1, Ack(m,n-1)) & \text{gdy } m > 0 \text{ i } n > 0 \end{cases}$$

daje się łatwo przepisać na deklarację funkcji.

a) Zrób to i oblicz wartość wyrażenia Ack(3,2).
b) Czy potrafisz udowodnić, że Twój program zatrzyma się i zwróci wynik po skończonym czasie?
c) Czy potrafisz oszacować koszt?
d) Napisz program A obliczający wartość $Ack(m,n)$ tej funkcji. Program A ma być zapisany w języku $\mathcal{L}_5$ programów iteracyjnych. Nie wprowadzaj deklaracji funkcji.

10.4. Co się stanie jeśli w przykładzie 10.4 usuniemy deklarację zmiennej x i w treści funkcji sqrt zmienną x zastąpimy przez result?

ROZDZIAŁ 11

$\mathcal{L}_7$ Procedury

W każdym module programu możesz wprowadzić nowe polecenie deklarując odpowiednią procedurę, np. P.

```
unit P: procedure(PARAMETRY);
  DEKLARACJE LOKALNE
begin
  INSTRUKCJE
end P
```

W tym module (i w każdym innym module, który jest w nim zawarty) możesz użyć polecenia call P(Argumenty).

TODO
1. Zauważ, jesli procedura (lub funkcja) nie ma wielkosci nielokalnych, tzn. każda wielkość jest albo parametrem formalnym albo jest zadeklarowana lokalnie w tej procedurze to można ją wywoływać z dowolnego miejsca z którego jest dostępna.
2. Deklaracja procedury z wielkościami nielokalnymi
– może być przestawiona w inne miejsce pod warunkiem ...
– może być wywołana skądinąd pod warunkiem ...
Te warunki mogą się różnić.
3. Reguła kopii – to właściwie jest przyjęcie, schematu nieskończenie wielu aksjomatów. Czyli Definicja!
Ale... żeby definicja mogła być przyjęta powinny być spełnione warunki.
Myślę, że reguła kopii powinna zostać poddana obostrzeniom:
– udowodnij niesprzeczność ,
– udowodnij jednoznaczność ...

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

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
```

Drugi przykład ... mergesort? quicksort? search w BST? obmyśl

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.

```
program sortuj3elementy;
   var a = l1
   var b = 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
```

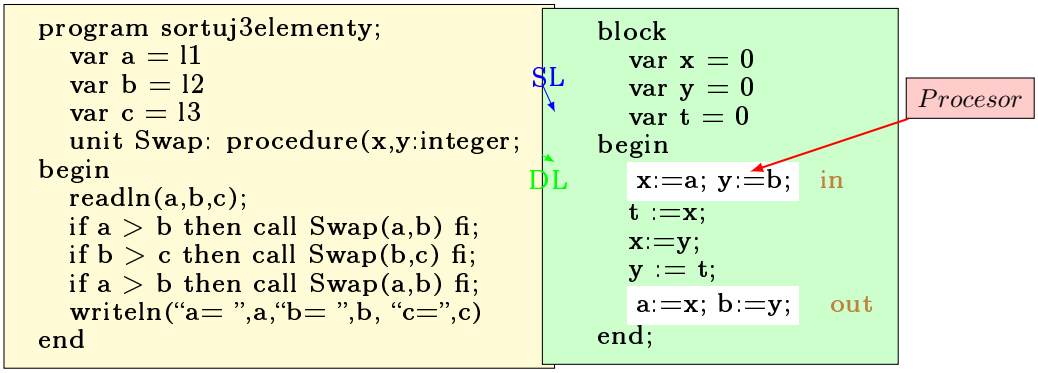
Procesor

Jeśli $l1 \leq l2$ to następny stan wygląda tak.

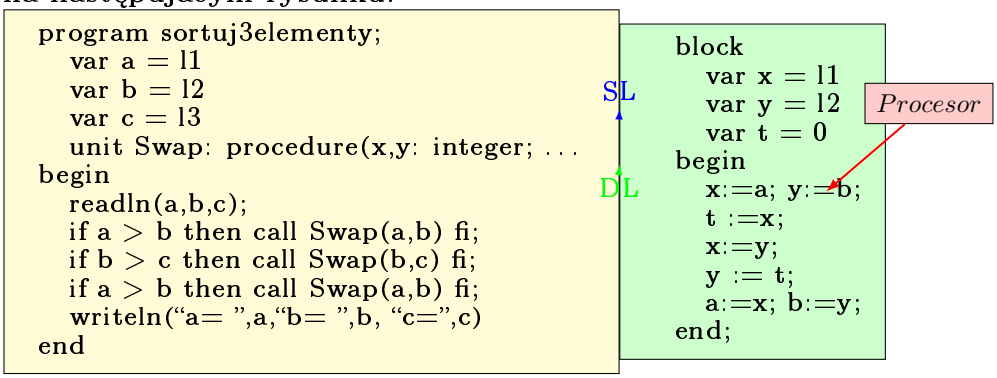
```
program sortuj3elementy;
   var a = l1
   var b = l2
   var c = l3
   unit Swap: procedure(x,y: integer; ...
begin
   readln(a,b,c);
   if a > b then call Swap(a,b) fi;
   if b > c then call Swap(b,c) fi;
   if a > b then call Swap(a,b) fi;
   writeln("a= ",a,"b= ",b, "c=",c)
end
```

Procesor

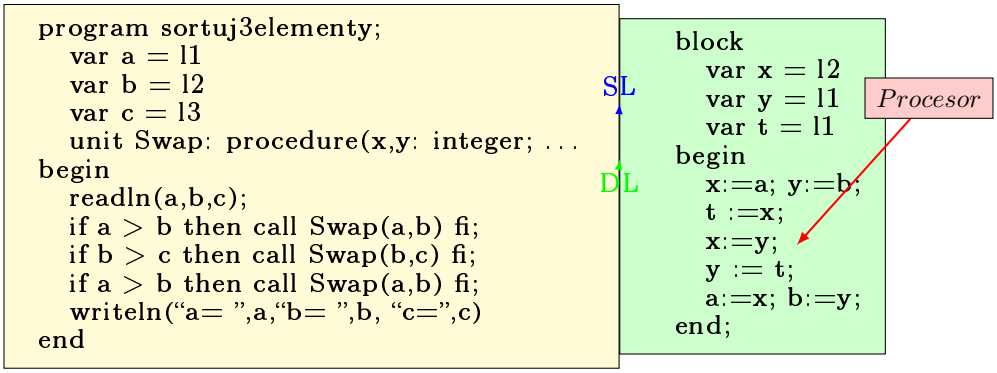
Natomiast jeśli $l1 > l2$ to następny stan wygląda tak.



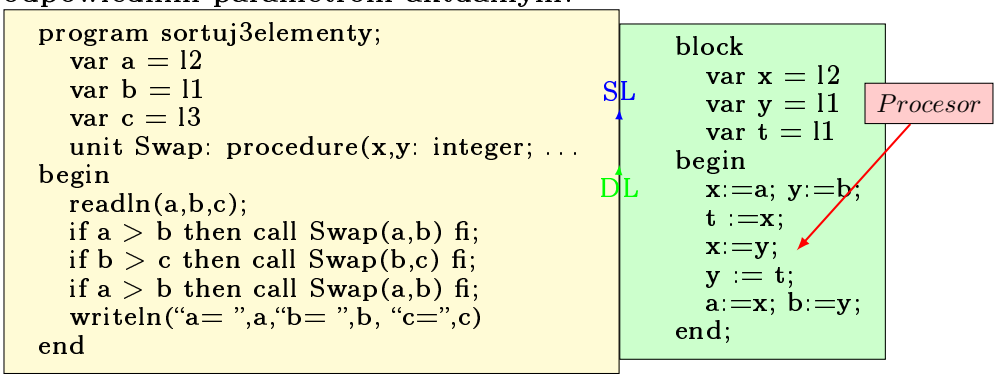
Pojawiła się nowa jednostka dynamiczna: rekord aktywacji procedury Swap. Utworzono ją według następujacej reguły kopii: Deklarację procedury przekształca się w instrukcję bloku. Parametry formalne stają się wielkościami (lokalnymi) zadeklarowanymi wewnątrz bloku. W treści bloku pojawiają się najpierw 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.



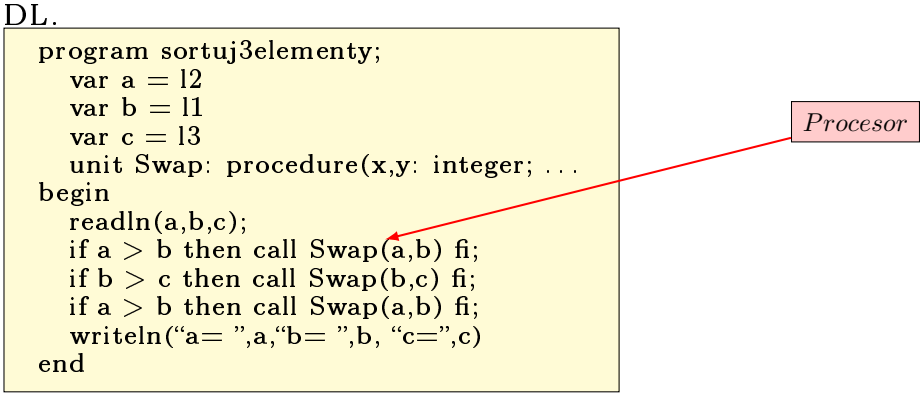
Po wykonaniu trzech kolejnych instrukcji przypisania zawartych w treści procedury Swap, na kolejnym rysunku zobaczymy taki obraz.



Kończymy wykonywanie procedury Swap. Przekazujemy wartości parametrów formalnych oznaczonych modyfikatorem out odpowiednim parametrom aktualnym.

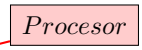


Usuwamy rekord aktywacji procedury Swap. Procesor wznawia działanie w rekordzie aktywacji wskazanym przez strzałkę DL.



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
```

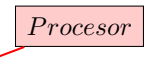


W podobny sposób przekonujemy się, że po wykonaniu drugiej instrukcji warunkowej if b >c ... fi zachodzi rôwność $c = max(max(a,b),c)$. Natomiast nie wiadomo czy $a < b$.

```
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

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 11.1. 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

$$\underbrace{\underbrace{\overbrace{\text{unit } \Psi}^{nazwa}\text{: procedure } \overbrace{(\mu_1\ \kappa_1 : T_1,\ \ldots, \mu_n\ \kappa_n : T_n)}^{\text{lista parametrów formalnych}}}_{\text{nagłówek procedury}}\ ;\underbrace{\mathbb{D}\ \text{begin}\ \mathbb{I}\ \text{end}\ \Psi}_{\text{treść procedury}}}$$

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 11.2.

```
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 11.2. Instrukcja procedury ma następującą budowę

$$\text{call } \Psi \quad \underbrace{(\omega_1, \ldots, \omega_n)}_{lista\ \ par.aktualnych}$$

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.

Zbiór procedur zadeklarowanych w programie wraz z relacja *call* jest grafem wywołań. rys. moduły + call

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.

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 call – 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 11.3. 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 rôwnoważny 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
      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                                                               }  (∗ ≡ callPsi...∗)
   writeln(“z =”, z, “y =”, y)
end
```

Kolejny przykład

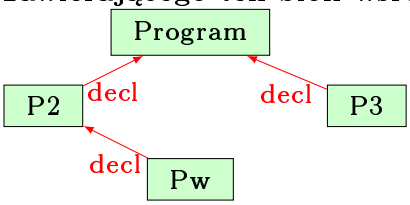
Przykład 11.4. W tym przykładzie pokażemy, że może istnieć konflikt nazw: lokalna nazwa parametru formalnego i nielokalna zmienna występująca w parametrze aktualnym.
Transmisja parametru aktualnego do parametru formalnego budzi w tym przypadku wątpliwości.
a := (y+3)*a
Wartość wyrażenia (y+3)*a powinna 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.

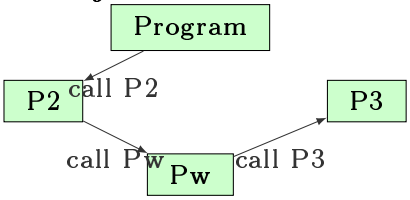
Przykład 11.5.

```
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
```

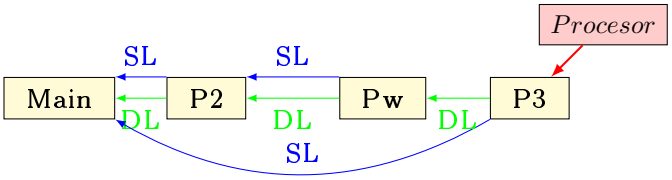
Statyczna struktura modułów tego programu jest widoczna na rysunku poniżej. Zielone prostokąty reprezentują procedury (lub bloki). Strzałka decl prowadzi od modułu A do modułu B gdy ten ostatni jest zadeklarowany w module B. W przypadku gdy moduł A jest blokiem, strzałka decl prowadzi do modułu zawierającego 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.

$$\text{call } P(a,b,c)\alpha \Leftrightarrow \left\{ \begin{array}{l} \text{block} \\ \text{oblicz wartości a, b i c,} \\ \text{złóż je na stos} \\ \text{block} \\ \text{treść tego bloku weź z deklaracji P;} \\ \text{modyfikując ją w ten sposób:} \\ \text{przypisz wartości ze stosu} \\ \text{a, b i c parametrom formalnym} \\ \text{tu wstaw treść P} \\ \text{zakończenie – odebranie wyników} \\ \text{end} \\ \text{end} \end{array} \right\} \alpha$$

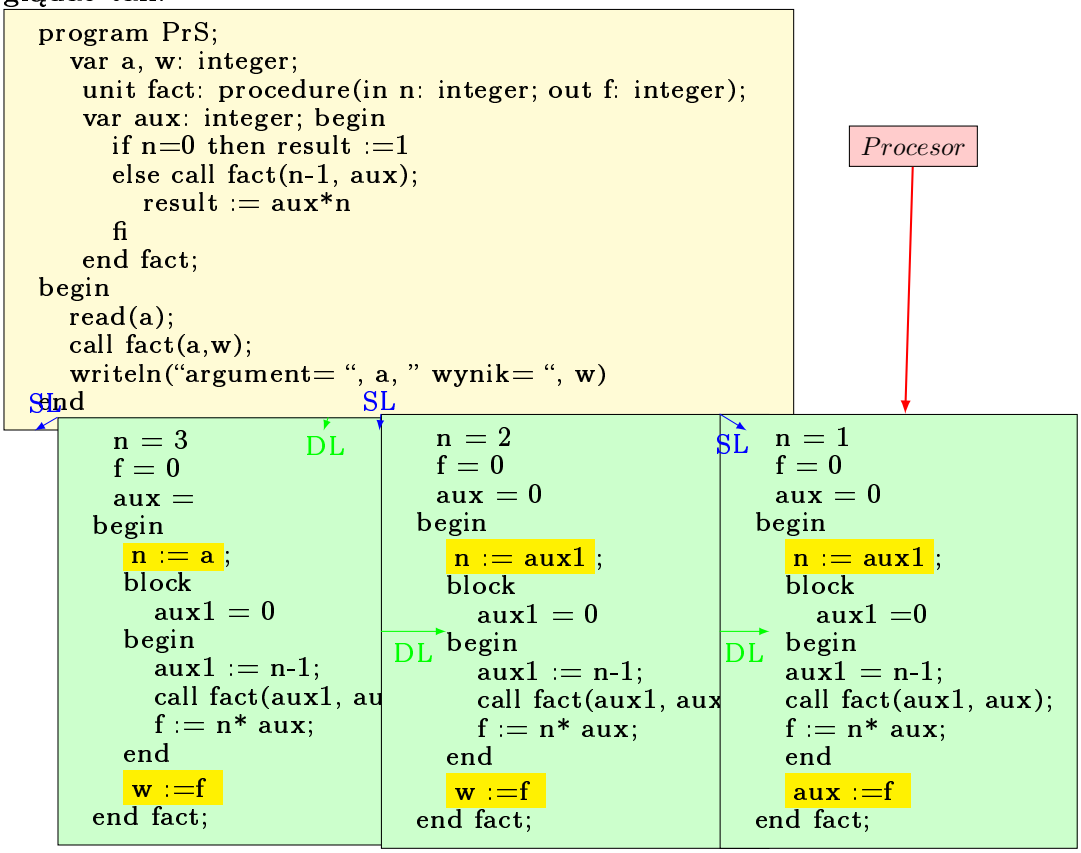
Problemy i pytania
Czy to musi być takie skomplikowane?
Zauważ, obliczenie wartośći parametrów ma miejsce tam gdzie jest instrukcja procedury, ale ciąg dalszy obliczeń ma miejsce w rekordzie aktywacji procedury P. Jeśli parametry aktualne są zmiennymi lub stałymi to reguła kopii jest prostsza:
call P(a, 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) }

4.1. Procedury rekurencyjne. W tym miejscu przedstawimy dwa przykłady: procedurę rekurencyjną obliczania silni oraz procedurę ustawiającą 8 hetmanów na szachownicy.

Przykład 11.6.

```
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:



ta koncepcja reguły kopii jest będna! 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 $P(a_1, \ldots, a_n)$ 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?

4.2. Protokół call - realizacja instrukcji procedury. Poniżej opisujemy protokół realizacji polecenia call. Do zrozumienia jak on działa wystarczy, że opiszemy go dla przypadku gdy procedura P jest zadeklarowana w ten sposób:

```
unit P: procedure(in a: integer, inout b: integer, out c: real);
  var t: integer;
begin
  t :=a;
  c:=-77;
  b := t;
end P
```

(1) Niech J oznacza jednostkę dynamiczną , w której należy wykonać polecenie

call P(t+u, v,w)

wykonujemy ciąg połowicznych instrukcji przypisania

(t+u)!, v!

polega to na wysłaniu wartości tych dwu wyrażeń na stos roboczy,

(2) utwórz rekord aktywacji procedury P i przejdź do niego
- wyznacz jednostkę dynamiczną J', która zawiera deklarację procedury P,
  Niech i= min(j: SL^j(J) zawiera deklarację procedury P).
  J'= SL^i(J)
- utwórz nową jednostkę dynamiczną – rekord aktywacji procedury P, oznaczmy ją K.

K.SL := J'
K.DL:= J

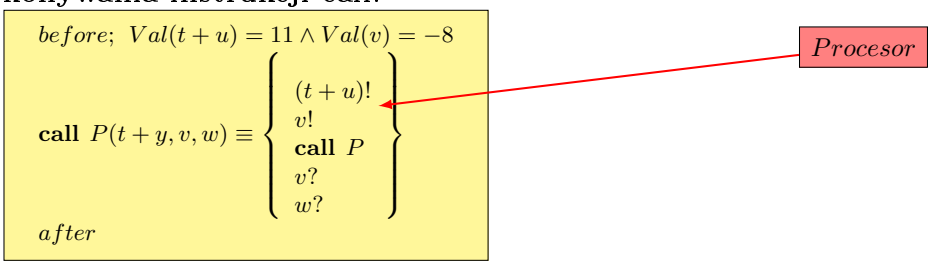
(3) W rekordzie aktywacji procedury P:

- odbierz parametry aktualne
  a? c?
  w ten sposób dopełni się wykonywanie poleceń a:= t+u; c:=v.
  Zwróć uwagę: obliczenie wartości wyrażeń (t+u) oraz v wykonuje się w otoczeniu rekordu aktywacji, w którym znajduje się polecenie call, a dokończenie instrukcji przypisania dokonuje się w rekordzie aktywacji procedury P.
- wykonaj instrukcje treści procedury P,
  podczas wykonywania tych poleceń (treści procedury), może dojść do odczytywania wartości parametrów formalnych a i c (a także b), może tez dojść do przypisywania tym parametrom nowych wartości
- wyślij wyniki b!c! na stos roboczy ,
- powróć do wykonywania poleceń w miejscu wystąpienia instrukcji procedury, tzn. procesor powraca do wykonywania instrukcji w jednostce dynamicznej J.

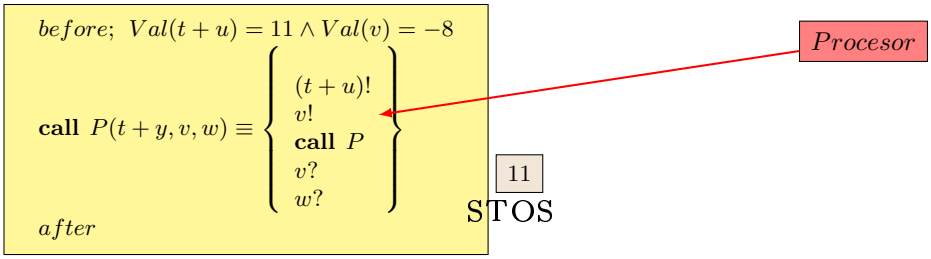
(4) W jednostce dynamicznej J:

- odbierz wyniki v? i w?
  w ten sposób dopełni się wykonywanie poleceń v:=b;w:=c
- kontynuuj obliczenie (rekord aktywacji procedury P będzie usunięty)

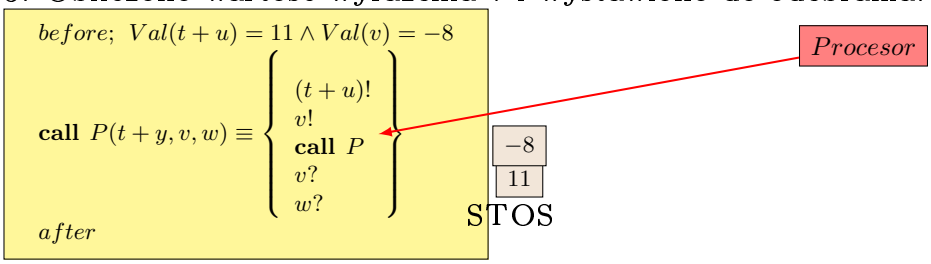
Historia obrazkowa w dziesięciu odsłonach. 1. Początek wykonywania instrukcji call.



2. Obliczono wartość wyrażenie $t = u$ i wystawiono do odebrania.

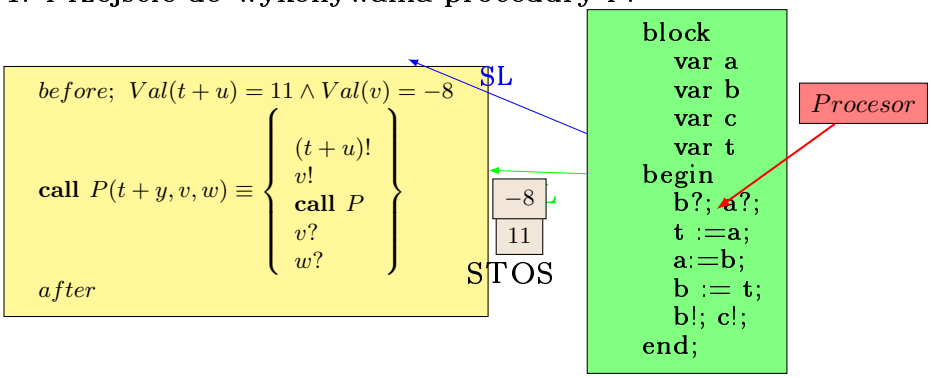


3. Obliczono wartość wyrażenia v i wystawiono do odebrania.



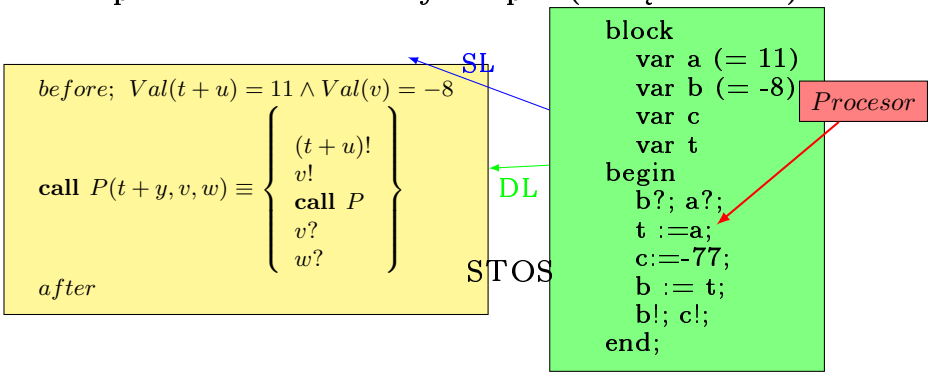
Dwukrotnie wykonano operację wstaw do stosu, raz wartość wyrażenia t+u tj. 11, drugi raz wartość wyrażenia v tj. -8.

4. Przejście do wykonywania procedury P.

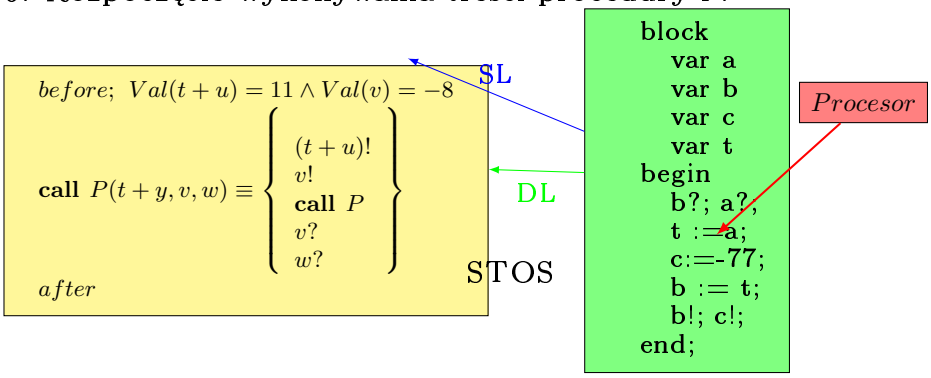


Strzałka SL wskazuje na pewien rekord aktywacji, który zawiera deklarację procedury P.

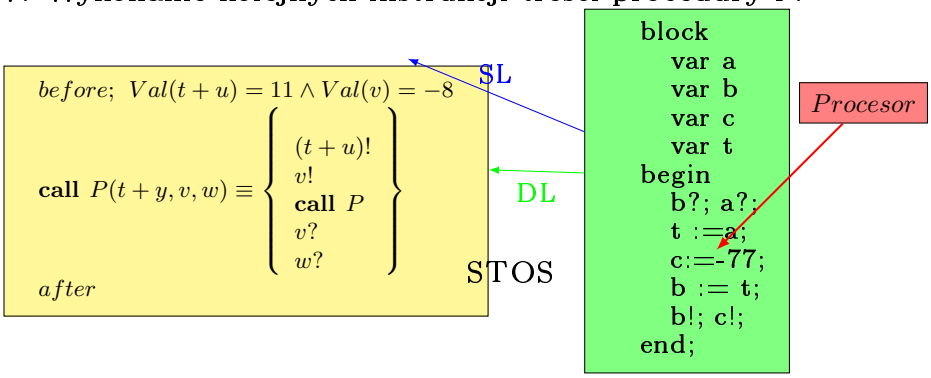
5. Pobranie dwu kolejnych wystawionych wartości i przypisanie ich parametrom formalnym input (to są zmienne) a i b.



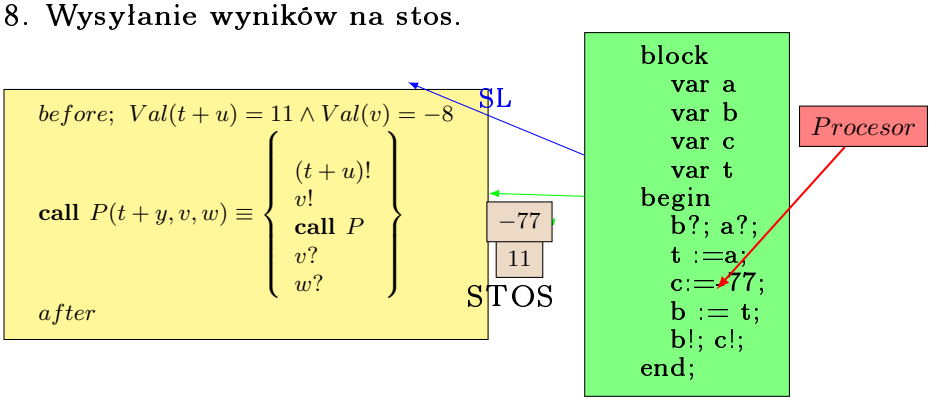
6. Rozpoczęcie wykonywania treści procedury P.



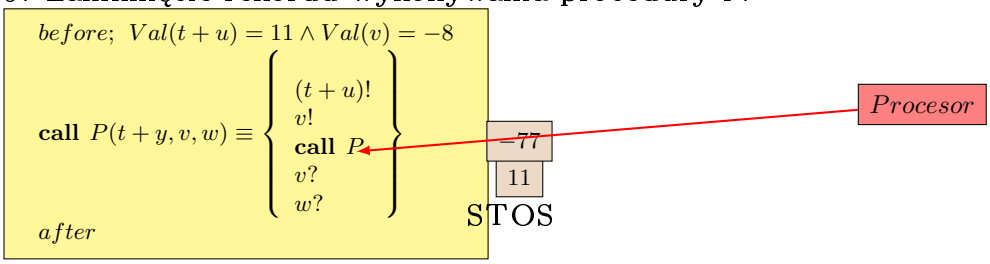
7. Wykonanie kolejnych instrukcji treści procedury P.



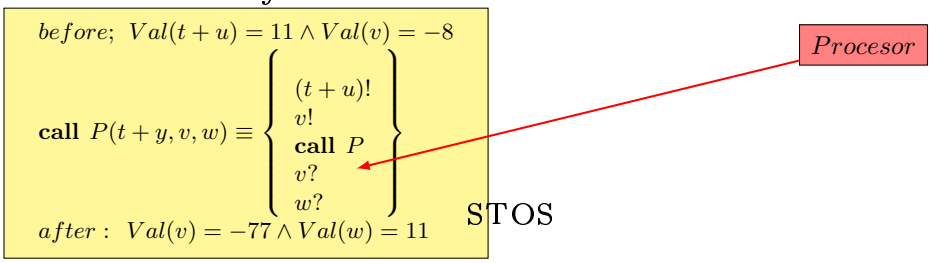
8. Wysyłanie wyników na stos.



9. Zamknięcie rekordu wykonywania procedury P.



10. Odbieranie wyników ze stosu.



Realizacja tego protokołu może byc tańsza. Jeśli wydaje Ci się, że ten protokół wykonywania instrukcji procedury jest skomplikowany i kosztowny w realizacji to masz rację. Kompilator i maszyna wirtualna realizują ten sam cel taniej. W istniejacych instalacjach wystawianie wartości jest realizowane przez wstawianie do stosu. Natomiast odbieranie parametrów aktualnych i przypisywanie wartości parametrom formalnym nie kosztuje NIC!. Jak to możliwe? Spróbuj odgadnąć.

Przekazywanie procedur, funkcji, klas. Jeśli parametrem aktualnym jest funkcja to do rekordu aktywacji (poprzez stos!) przekazywana jest para: <nazwa funkcji, rekord aktywacji w którym zawarta jest deklaracja funkcji>.
Uwaga. Moduł procedury, której (jednym z) parametrem jest funkcja, ... powinno się nazywać funkcjonałem.

4.3. Sprzeczność. W tym miejscu warto zwrócić uwagę na możliwość zadeklarowania procedur wzajemnie sprzecznych.

5. Dowód programu PawelG

Przypominamy program 0.0.1 z przedmowy.

```
program PawelG:
    var A: arrayof integer;
    var n, i, k, l, u: 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
    readln(n);
    array A dim(1:n);
    l:=lower(A); u:=upper(A);
    for i := l to u do A(i) := 0 od;
    k :=l;
    call F;
    writeln("Bywaj")
end PawelG
```

Zamierzamy udowodnić następujące

Twierdzenie 11.1. Dla każdej liczby naturalnej n program Pawel oblicza i drukuje wszystkie permutacje zbioru $\{1, \ldots, n\}$.

5.1. Dowód M – matematyczny, prawie formalny. Zadanie jakie sobie teraz stawiamy to przedstawienie dowodu, w którym nie odnosimy się do pojęć: obliczenia, stanu obliczenia etc. Dowód ma zawierać tylko takie formuły, które są albo aksjomatami algorytmicznej teorii liczb naturalnych, por. podrozdział ??, zobacz też [MS92], albo aksjomatami rachunku programów, zob. [MS87], albo wynikają z zastosowania pewnej reguły wnioskowania rachunku programów do pewnego zbioru formuł wcześniej udowodnionych. Wielokrotnie stosujemy prawa rachunku zdań i wykorzystujemy własność ekstensjonalności. Zastępujemy pewną podformułę ψ danej formuły ϕ, formułą θ wiedząc, że równość $\theta \equiv \psi$ posiada dowód. Nie będziemy takich przypadków rozpatrywać zbyt szczegółowo by nie zmącić głównej nici dowodu. Dopuszczamy zastosowanie reguły ω w ograniczony

sposób: możesz mianowicie udowodnić, pewne metatwierdzenie stwierdzające, że dla każdego $i \in N$ formuła γ_i posiada dowód i zastosować jedną z ω-reguł rachunku programów R_3, R_6, R_7.

Taki dowód quasi-formalny, może więc zawierać nieskończone podzbiory formuł. Tym niemniej, dowody o jakich myślimy mogą być sprawdzane mechanicznie przez proof-checker. Więcej o tym w nieopublikowanym artykule [Sala].

Co mamy udowodnić? Cel nasz zostanie osiągnięty gdy potrafimy wykazać, że 1°) każde wykonanie polecenia call DrukujA spowoduje wydrukowanie pewnej permutacji liczb $1, \ldots, n$, 2°) liczba wykonanych poleceń call DrukujA jest równa $n!$ oraz 3°) wydrukowane permutacje nie powtarzają się.

Jakie narzędzia, jakie aksjomaty i reguły wnioskowania są niezbędne dla przeprowadzenia dowodu?
W analizowanym programie występują zmienne typu integer i operacje na liczbach całkowitych. Czy możemy uznać, że do przeprowadzenia dowodu wystarczy algorytmiczna teoria liczb naturalnych $\mathcal{ATN}$? Zauważ. w programie PawelG występuje atomowa instrukcja procedury call F. Język teorii $\mathcal{ATN}$ nie zawiera takiej instrukcji atomowej, ani (tym bardziej) aksjomatu opisującego działanie tej instrukcji.
Trzeba więc przyjąć, że deklaracja procedury F wprowadzona w programie opisuje sposób wykonania polecenia call F, determinuje jego semantykę. Ale dla nas wprowadzenie deklaracji procedury F oznacza coś więcej, mianowicie, deklaracja procedury F jest schematem nieskończonie wielu dodatkowych aksjomatów postaci

$$(\{\mathsf{call\ F}\}\varphi \Leftrightarrow \{\mathrm{Body}_F\}\varphi)$$

gdzie φ jest dowolną formułą. Skrót $\mathrm{Body_F}$ oznacza treść procedury F. Dokładniej, każda równoważność zbudowana według poniższego schematu jest dodatkowym aksjomatem opisującym

procedurę F.
(44)

$$\{\textbf{call F}\}\varphi \Leftrightarrow \left\{\begin{array}{l}
(*\ \textbf{poniżej znajduje się treść F tzn. Body_F}\ *)\\
\textbf{block}\\
\quad \textbf{var}\ i:\ integer;\\
\textbf{begin}\\
\quad \textsf{if}\ k = n+1\ \textsf{then call}\ DrukujA\\
\quad \textsf{else}\\
\qquad \textsf{for}\ i := 1\ \textsf{to}\ n\ \textsf{do}\\
\qquad\quad \textsf{if}\ A[i] = 0\ \textsf{then}\\
\qquad\qquad A[i] := k;\ k := k+1;\\
\qquad\qquad \textsf{call}\ F;\\
\qquad\qquad A[i] := 0;\ k := k-1;\\
\qquad\quad \textsf{fi}\\
\qquad \textsf{od}\\
\quad \textsf{fi}\\
\textbf{end block}
\end{array}\right\}\varphi$$

Co więcej, w powyższej formule, w treści procedury F można wszystkie wystąpienia identyfikatora i równocześnie zastąpić przez dowolny inny identyfikator, por. formułę (isub). Mówimy, że zmienna i zadeklarowana w bloku jest zmienną związaną.

Dowód twierdzenia 11.1 będziemy przeprowadzać w algorytmicznej teorii $\mathcal{T}'$, która powstaje z algorytmicznej teorii liczb naturalnych $\mathcal{ATN}$ przez dodanie do języka nowej instrukcji atomowej call F, i dodanie do zbioru aksjomatów teorii $\mathcal{ATN}$ nieskończonego zbioru formuł zbudowanych według omówionego powyżej schematu.

Niech $\delta(n,k,i_1,\ldots,i_{k-1})$ bedzie oznaczeniem formuły o następującym schemacie
(45)

$$\delta(n,k,i_1,\ldots,i_{k-1}) \stackrel{df}{\equiv} \left(\begin{array}{l}
(1 \le k \le n+1) \wedge \bigwedge_{j=1}^{k-1}\big((1 \le i_j \le n) \wedge\ (A[i_j] = j)\big) \wedge\\
\left\{\begin{array}{l}
z := 0;\\
\textbf{for}\ j := 1\ \textbf{to}\ n\ \textbf{do}\\
\quad \textbf{if}\ A[j] = 0\ \textbf{then}\ z := z+1\ \textbf{fi}\\
\textbf{od}
\end{array}\right\}(z = n-k+1)
\end{array}\right)$$

Formułę $\delta(n,k,i_1,\ldots,i_{k-1})$ czyta się w następujący sposób (jest to jej znaczenie semantyczne):
spełnione są następujące warunki

(i) liczba naturalna n jest dodatnia,
(ii) wartością zmiennej A jest n-elementowy obiekt tablicowy,
(iii) liczba naturalna k spełnia podwójną nierówność $1 \le k \le n+1$ i liczby $1,\ldots,k-1$ są zapisane w dowolnym układzie na pozycjach $i_1,\ldots,i_{k-1}$ tablicy A, tj. $\forall_{j=1}^{k-1} A[i_j] = j$,

(iv) pozostałe miejsca tablicy A, oznaczmy je $r_k, \ldots, r_n$, są rôwne zero, $\forall_{j=k}^{n} A[r_j] = 0$.

Udowodnimy następujący fakt.

Lemat 11.2. Niech wartością zmiennej A będzie n-elementowa tablica (wektor) liczb naturalnych. Każda formuła o poniższym schemacie jest twierdzeniem teorii $\mathcal{T}'$

(46) $$\mathcal{T}' \vdash \delta(n, k, i_1, \ldots, i_{k-1}) \Rightarrow \{\mathsf{call}\ F\}\, \delta(n, k, i_1, \ldots, i_{k-1}).$$

Dowôd. Dowôd lematu przebiega przez indukcję ze względu na liczbę $n+1-k$, tj. liczbę wolnych miejsc w tablicy A.

B0)(baza) Niech $k = n+1$, tzn. liczba wolnych miejsc jest 0.

Jeśli spełniony jest warunek $(k = n+1) \wedge \delta(n, k, i_1, \ldots, i_n)$ to tablica A zawiera pewną permutację liczb $1, \ldots n$, ponieważ zachodzi $\bigwedge_{j=1}^{n} A[i_j] = j$.
Jest tautologią formuła

(47) $$\vdash \delta(n, n+1, i_1, \ldots, i_n) \Rightarrow \bigwedge_{j=1}^{n} A[i_j] = j$$

Instrukcja write(x) nie zmienia wartości żadnej zmiennej. Własnością tej instrukcji jest

(48) $$(y = k) \equiv \{\mathsf{write(x)}\}(y = k)$$

Posługując sie tym faktem dowodzimy, że jest twierdzeniem teorii $\mathcal{T}'$ implikacja

(49) $$\mathcal{T}' \vdash \delta(n, n+1, i_1, \ldots, i_n) \Rightarrow \{call\ \ DrukujA\}(\bigwedge_{j=1}^{n} A[i_j] = j)$$

Postępując podobnie udowodnimy formułę

(50) $$\big(\delta(n, n+1, i_1, \ldots, i_n) \Rightarrow \{call\ \ DrukujA\}\, \delta(n, n+1, i_1, \ldots, i_n)\big)$$

Można to zapisać nieco inaczej

(51) $$\mathcal{T}' \vdash \big(\delta(n, k, i_1, \ldots, i_n) \wedge k = n{+}1\big) \Rightarrow \{call\ DrukujA\}\, \delta(n, n{+}1, i_1, \ldots, i_n)$$

Korzystamy z reguły wnioskowania (??,zob. stronę ??) o instrukcji if.
(52)

$$\mathcal{T}' \vdash \big(\delta(n, k, i_1, \ldots, i_n) \wedge k = n{+}1\big) \Rightarrow \left\{ \begin{array}{l} \textbf{if } k = n+1 \\ \textbf{then call } DrukujA \\ \textbf{else} \\ \quad \textbf{(* tu wstaw cokolwiek *)} \\ \textbf{fi} \end{array} \right\} \delta(n, n{+}1, i_1, \ldots, i_n)$$

Wstawiamy to co potrzeba, dbając o to by zmienna $i_{\underline{n}}$ występowała tylko w tej instrukcji for.

(53)

$$\mathcal{T}' \vdash (\delta(n,k,i_1,\ldots,i_n) \wedge k = n{+}1) \Rightarrow \left\{ \begin{array}{l} \textbf{if } k = n+1 \\ \textbf{then } \mathsf{call}\ DrukujA; \\ \textbf{else} \\ \quad \mathsf{for}\ i_{\underline{n}} := 1\ \mathsf{to}\ n\ \mathsf{do} \\ \quad\quad \mathsf{if}\ A[i_{\underline{n}}] = 0\ \mathsf{then} \\ \quad\quad\quad A[i_{\underline{n}}] := k;\ k := k+1; \\ \quad\quad\quad \mathsf{call}\ F; \\ \quad\quad\quad A[i_{\underline{n}}] := 0;\ k := k-1; \\ \quad\quad \mathsf{fi} \\ \quad \mathsf{od} \\ \textbf{fi} \end{array} \right\} \delta(n, n{+}1, i_1, \ldots, i_n)$$

Ponieważ zmienna $i_{\underline{n}}$ występuje tylko w tych kilku liniach to możemy zastosować aksjomat instrukcji bloku i otrzymamy

(isub)

$$\mathcal{T}' \vdash \delta(n,n{+}1,i_1,\ldots,i_n) \Rightarrow \left\{ \begin{array}{l} \textbf{block} \\ \quad \textbf{var } i : \ integer \\ \textbf{begin} \\ \quad \textbf{if } k = n+1 \\ \quad \textbf{then } \ \textbf{call } DrukujA; \\ \quad \textbf{else} \\ \quad\quad \textbf{for } i := 1 \textbf{ to } n \textbf{ do} \\ \quad\quad\quad \textbf{if } A[i] = 0 \textbf{ then} \\ \quad\quad\quad\quad A[i] := k;\ k := k+1; \\ \quad\quad\quad\quad \textbf{call } F; \\ \quad\quad\quad\quad A[i] := 0;\ k := k-1; \\ \quad\quad\quad \textbf{fi} \\ \quad\quad \textbf{od} \\ \quad \textbf{fi} \\ \textbf{end block} \end{array} \right\} \delta(n,n{+}1,i_1,\ldots,i_n)$$

Program występujący w powyższej formule (isub) to treść procedury F, możemy więc zastosować aksjomat, czyli deklarację procedury F

(54) $\quad \mathcal{T}' \vdash \delta(n,n+1,i_1,\ldots,i_n) \Rightarrow \{\mathsf{call}\ F\}\ \delta(n,n+1,i_1,\ldots,i_n)$

co kończy dowód przypadku gdy $k = n+1$.

W dalszej części dowodu wykorzystamy dwie niewielkie obserwacje. Zauważmy, że twierdzeniem rachunku programów, a więc i teorii $\mathcal{T}'$ jest poniższa formuła 55

Uwaga 11.3.

(55)

$$\big(\delta(n,k,i_1,\ldots,i_{k-1}) \wedge A[p] = 0\big) \Rightarrow \{A[p] := k;\ k := k{+}1\}\ \big(\delta(n,k{+}1,i_1,\ldots,i_{k-1},i_k) \wedge i_k = p\big)$$

Fakt ten jest łatwą konsekwencją zastosowania aksjomatu instrukcji przypisania ??. Z założenia $\delta(n,k,i_1,\ldots,i_{k-1})$ wynika,

że tablica A zawiera wszystkie liczby $1,\ldots,k-1$ czyli $\bigwedge_{j=1}^{k-1} A[i_j]=j$. Ponadto miejsce $A[p]$ tablicy A jest wolne, a więc $p\neq i_j$ dla $j=1,\ldots,k-1$.

Stąd wynika, że $\{A[p]:=k\}(\bigwedge_{j=1}^{k} A[i_j]=j)\wedge A[p]=k$. Z kolei, $(z=n-k+1)\Rightarrow\{k:=k+1\}(n-k)$. Wynika stąd

(56)
$$\big((\bigwedge_{j=1}^{k-1} A[i_j]=j)\wedge z=n-k+1\big)\Rightarrow\{A[p]:=0;\,k:=k+1\}\big((\bigwedge_{j=1}^{k} A[i_j]=j)\wedge(z=n-k)\big)$$

Podobnie, następująca formuła (57) jest twierdzeniem teorii $\mathcal{T}'$.

Uwaga 11.4.

(57)
$$\big(\delta(n,k+1,i_1,\ldots,i_{k-1},p)\wedge A[p]=k\big)\Rightarrow\{A[p]:=0;\,k:=k-1\}\,\delta(n,k,i_1,\ldots,i_{k-1})$$

I) (krok indukcyjny)
(założenie) zakładamy, że dla każdego układu $p_1,\ldots,p_{k-1}$ liczb naturalnych, następująca implikacja jest twierdzeniem teorii $\mathcal{T}'$

(58) $$\mathcal{T}'\vdash\delta(n,k,p_1,\ldots,p_{k-1})\Rightarrow\{\mathsf{call}\ F\}\,\delta(n,k,p_1,\ldots,p_{k-1}).$$

(teza) Wykażemy, że dla dowolnego układu liczb naturalnych $i_1,\ldots,i_{k-2}$ i $n-k+2$ miejsc zerowych w tablicy A można udowodnić, że

(59) $$\mathcal{T}'\vdash\delta(n,k-1,i_1,\ldots,i_{k-2})\Rightarrow\{\mathsf{call}\ F\}\,\delta(n,k-1,i_1,\ldots,i_{k-2}).$$

Oznaczmy miejsca zerowe w tablicy A przez $r_{k-1},\ldots,r_n$ czyli dla $j=k-1,\ldots,n$ zachodzi $A[r_j]=0$. Rozpatrzymy po kolei te miejsca zerowe. Zauważmy, że dla $j=k-1,\ldots,n$ można udowodnić implikacje

(60)
$$A[r_j]=0\wedge\delta(n,k-1,i_1,\ldots,i_{k-2})\Rightarrow\left\{\begin{array}{l}A[r_j]:=k;\,k:=k+1;\\ call\ F;\\ A[r_j]:=0;k:=k-1\end{array}\right\}\delta(n,k-1,i_1,\ldots,i_{k-2})$$

Wynika to z Faktów (11.3) i (11.4). Dla każdego $j=k-1,\ldots,n$ warunek $A[r_j]=0\wedge\delta(n,k-1,i_1,\ldots,i_{k-2})$ pociąga za sobą $\{A[r_j]:=k;\,k:=k+1\}\delta(n,k,i_1,\ldots,i_{k-2},r_j)$ (Fakt(11.3)). Z założenia indukcyjnego

(61) $$\delta(n,k,i_1,\ldots,i_{k-1},r_j)\Rightarrow\{\mathsf{call}\ F\}\,\delta(n,k,i_1,\ldots,i_{k-2},r_j)$$

Teraz wykorzystamy Fakt (11.4)

(62) $$\delta(n,k,i_1,\ldots,i_{k-2},r_j)\Rightarrow\{A[r_j]:=0;\,k:=k-1\}\delta(n,k-1,i_1,\ldots,i_{k-2}).$$

Z kolei dla $A[i]\neq 0$ zachodzi w oczywisty sposób

(63) $$A[i]\neq 0\wedge\delta(n,k-1,i_1,\ldots,i_{k-2})\Rightarrow\delta(n,k-1,i_1,\ldots,i_{k-2}).$$

Przyjmijmy następujące oznaczenia
(64)

$$\theta_i(n,k{-}1,i_1,\ldots,i_{k-2}) \stackrel{df}{\equiv} \begin{cases} A[i]\neq 0 \wedge \delta(n,k-1,i_1,\ldots,i_{k-2}) & \textbf{gdy} A[i]\neq 0 \\ A[i]=0 \wedge \left\{\begin{array}{l} A[i]:=k;\\ k:=k+1;\\ \mathit{call}\ F;\\ A[i]:=0;\\ k:=k-1 \end{array}\right\} \delta(n,k-1,i_1,\ldots,i_{k-2}) & \textbf{gdy} A[i]=0 \end{cases}$$

W ten sposób, wykorzystując aksjomat instrukcji warunkowej if, wykazaliśmy, że dla każdego $i=1,\ldots,n$ twierdzeniem teorii $\mathcal{T}'$ jest
(65)

$$\mathcal{T}' \vdash \delta(n,k-1,i_1,\ldots,i_{k-2}) \Rightarrow \left\{\begin{array}{l} \textbf{if}\ A[i]=0\ \textbf{then}\\ \quad A[i]:=k;\\ \quad k:=k+1;\\ \quad \mathit{call}\ F;\\ \quad A[i]:=0;\\ \quad k:=k-1\\ \textbf{fi} \end{array}\right\} \delta(n,k-1,i_1,\ldots,i_{k-2})$$

Zastosujemy następującą regułę wnioskowania pozwalającą na wprowadzenie kwantyfikatora ogólnego ograniczonego po instrukcji for
(wprkwog)

$$\frac{\bigwedge\limits_{i=1}^{n} \{P(i)\}\,\vartheta(i),\quad \bigwedge\limits_{1\leq i<j\leq n} \big(\{P(i);\,P(j)\}\vartheta(i) \equiv \{P(i)\}\,\vartheta(i)\big)}{\left\{\begin{array}{l} \textbf{for}\ i\leftarrow 1\ \textbf{to}\ n\\ \textbf{do}\\ \quad P(i)\\ \textbf{od} \end{array}\right\} \bigwedge\limits_{i=1}^{n} \vartheta(i)}$$

by uzyskać
(66)

$$\mathcal{T}' \vdash \delta(n,k-1,i_1,\ldots,i_{k-2}) \Rightarrow \left\{\begin{array}{l} \textbf{for}\ i:=1\ \textbf{to}\ n\ \textbf{do}\\ \quad \textbf{if}\ A[i]=0\ \textbf{then}\\ \quad\quad A[i]:=k;\\ \quad\quad k:=k+1;\\ \quad\quad \mathit{call}\ F;\\ \quad\quad A[i]:=0;\\ \quad\quad k:=k-1\\ \quad \textbf{fi}\\ \textbf{od} \end{array}\right\} \delta(n,k-1,i_1,\ldots,i_{k-2})$$

Wiemy, że $k-1 \neq n+1$ jeszcze raz zastosujemy aksjomat instrukcji if

(67)

$$\mathcal{T}' \vdash \delta(n, k-1, i_1, \ldots, i_{k-2}) \Rightarrow \left\{ \begin{array}{l} \textbf{if } k = n+1 \textbf{ then } \mathit{call}\ \mathit{DrukujA} \textbf{ else} \\ \textbf{for } i := 1 \textbf{ to } n \textbf{ do} \\ \quad \textbf{if } A[i] = 0 \textbf{ then} \\ \quad\quad A[i] := k; \\ \quad\quad k := k+1; \\ \quad\quad \mathit{call}\ F; \\ \quad\quad A[i] := 0; \\ \quad\quad k := k-1 \\ \quad \textbf{fi} \\ \textbf{od} \\ \textbf{fi} \end{array} \right\} \delta(n, k-1, i_1, \ldots, i_{k-2})$$

Zadbaliśmy o to by zmienna i różniła się od wszystkich innych, możemy więc zastosowac aksjomat instrukcji bloku

(68)

$$\mathcal{T}' \vdash \delta(n, k-1, i_1, \ldots, i_{k-2}) \Rightarrow \left\{ \begin{array}{l} \textbf{block} \\ \quad \textbf{var } i : \mathit{integer}; \\ \textbf{begin} \\ \textbf{if } k = n+1 \textbf{ then } \mathit{call}\ \mathit{DrukujA} \textbf{ else} \\ \textbf{for } i := 1 \textbf{ to } n \textbf{ do} \\ \quad \textbf{if } A[i] = 0 \textbf{ then} \\ \quad\quad A[i] := k; \\ \quad\quad k := k+1; \\ \quad\quad \mathit{call}\ F; \\ \quad\quad A[i] := 0; \\ \quad\quad k := k-1 \\ \quad \textbf{fi} \\ \textbf{od} \\ \textbf{fi} \\ \textbf{end block} \end{array} \right\} \delta(n, k-1, i_1, \ldots, i_{k-2})$$

Teraz skorzystamy z deklaracji=aksjomatu instrukcji call F i uzyskamy

(69) $$\mathcal{T}' \vdash \delta(n, k-1, i_1, \ldots, i_{k-2}) \Rightarrow \{\mathsf{call}\ F\}\, \delta(n, k-1, i_1, \ldots, i_{k-2}).$$

co kończy dowód lematu 11.2. □

W ten sposób udowodniliśmy, że dla każdej liczby naturalnej, dodatniej $n > 0$ program PawelG zakończy obliczenie.
Pozostaje do wykazania, że program ten drukuje wszystkie permutacje liczb $1, \ldots, n$.

Zauważyliśmy wcześniej, że każde wykonanie polecenia call DrukujA drukuje pewna permutację liczb $1, \ldots, n$, zob. str. 208.
Wykażemy, że polecenie call DrukujA jest wykonane $n!$ razy.

Ułatwimy sobie zadanie, wprowadzając do programu trzy niewielkie zmiany: 1°) dodajemy deklarację zmiennej licz obok deklaracji zmiennych n,k,j, 2°) obok polecenia call DrukujA dopisujemy ; licz:=licz+1, 3°) w programie głównym, przed instrukcją k:=1 wstawiamy instrukcję licz:=0;.
Definiujemy warunek początkowy

(70) $$\alpha(n,k,i_1,\ldots,i_{k-1},w) \stackrel{df}{\equiv} \big(\delta(n,k,i_1,\ldots,i_{k-1}) \wedge licz = w\big)$$

i warunek końcowy

(71) $$\beta(n,k,i_1,\ldots,i_{k-1},w) \stackrel{df}{\equiv} \big(\delta(n,k,i_1,\ldots,i_{k-1}) \wedge licz = (n-k+1)! + w\big)$$

Nietrudno teraz udowodnić odmieniony wariant lematu 11.2.

Lemat 11.5.

(72) $$\mathcal{T}' \vdash \alpha(n,k,i_1,\ldots,i_{k-1},w) \Rightarrow \{\mathsf{call}\ F\}\,\beta(n,k,i_1,\ldots,i_{k-1},w).$$

Dowód tego lematu naśladuje dowód lematu 11.2.
Gdy $k = n$ to wykonaniu polecenia call DrukujA towarzyszy instrukcja licz:=licz+1.
Jeśli $k < n$ i teza jest udowodniona dla $k+1$ to instrukcja for jest równoważna $(n-k+1)$- krotnemu wykonaniu polecenia złozonego {A[i]:=k; k:=k+1; call F; A[i]:=0; k:=k-1 }, zmienna i przyjmuje wartości $r_k,\ldots,r_n$ o których mówiliśmy wcześniej.
Z założenia indukcyjnego, każde takie polecenia powoduje zwiększenie licznika licz o $(n-k)!$. Razem licznik licz zostaje zwiększony o $(n-k+1)!$.

Koniec dowodu lematu 11.5

Pozostaje upewnić się, że żadna permutacja nie została wydrukowana dwukrotnie.
Rzeczywiście, potrafimy wykazać, że jeśli przyjąć zmieniony warunek końcowy
(73)

$$\beta'(n,k,i_1,\ldots,i_{k-1},w) \stackrel{df}{\equiv} \left(\begin{array}{l}\delta(n,k,i_1,\ldots,i_{k-1}) \wedge licz = (n-k+1)! + w \wedge \\ \text{żadna z tych permutacji nie powtarza się}\end{array}\right)$$

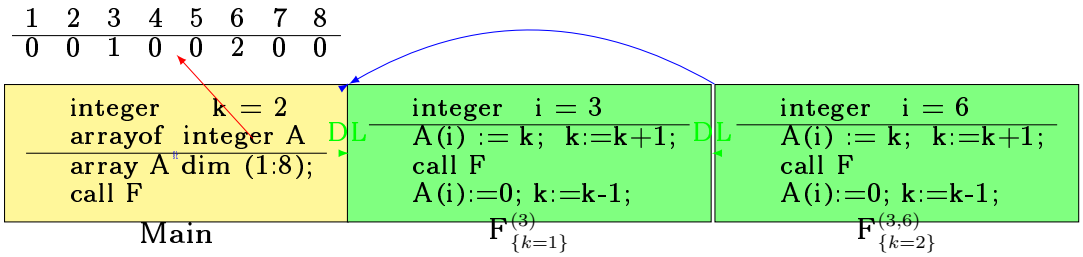
Nietrudno teraz udowodnić odmieniony wariant lematu 11.2.

Lemat 11.6.

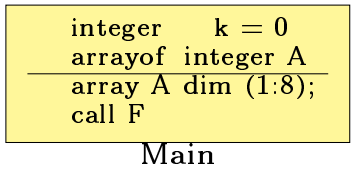
(74) $$\mathcal{T}' \vdash \alpha(n,k,i_1,\ldots,i_{k-1},w) \Rightarrow \{\mathsf{call}\ F\}\,\beta'(n,k,i_1,\ldots,i_{k-1},w).$$

Nie będziemy formalizować tego dowodu. Możesz spróbować zrobić to samodzielnie. Zacznij od napisania formuły wyrażającej własnośc dwie permutacje są różne. Nasz argument jest prosty. Podczas wykonywania instrukcji for powtarzamy $(n-k+1)$ razy czynność następującą: dla $j = k,\ldots,n$, zapisz liczbę k na miejscu $A[r_j]$ i na pozostałych $(n-k)$ wolnych miejscach wytwórz $(n-k)!$ różnych, permutacji. Widzimy, że utworzone w ten sposób $(n-k+1)!$ permutacje są różne, nie ma powtórzeń.
Koniec dowodu lematu 11.6 i koniec dowodu twierdzenia 11.1.

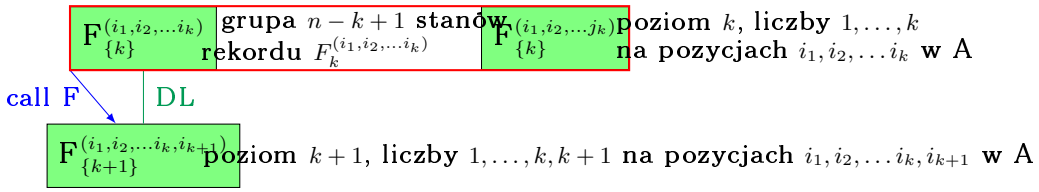
5.2. Dowód G – semantyczny, graficzny – "jak to działa"? W tym rozdziale przedstawiamy argumenty na rzecz twierdzenia 11.1, korzystając z wiedzy o tym jak wykonywany jest program. Konfiguracja (stan) obliczenia programu Pawel jest ciągiem rekordów aktywacji procedury F.



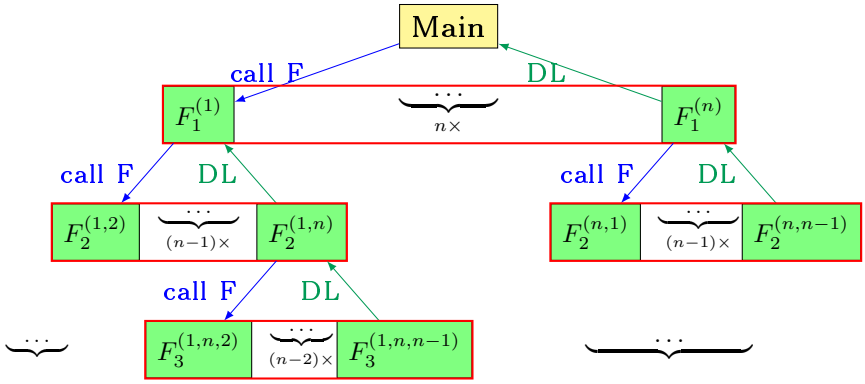
Analiza obliczeń programu Pawel
Drzewo możliwych aktywacji procedury F (tj. historii obliczenia)
Korzeniem drzewa jest rekord aktywacji programu głównego Pawel.



Dla każdego węzła w drzewa H aktywacji procedury F, każde wykonanie instrukcji procedury call F w tym rekordzie aktywacji spowoduje utworzenie nowego rekordu aktywacji procedury F – syna w'



Dla ustalonego n drzewo aktywacji H ma więc taki kształt



Na każdym poziomie $k, k < n$ każdy węzeł ma $n - k$ synów.
Węzeł na poziomie n ma jednego syna, jest nim rekord aktywacji procedury DrukujA. Liczba liści drzewa aktywacji jest równa $n!$. Stan tablicy A podczas wykonywania tej procedury z wartością zmiennej $k = n$ jest taki, że wypełnione jest każde miejsce w tej tablicy. Wartościami zapisanymi w tej tablicy są liczby $1, \ldots, n$ i żadna liczba się nie powtarza.
Po powrocie z rekordu aktywacji poziomu k żadna liczba w tablicy A nie jest większa od k.
Podsumowując, wydrukowane zostaną wszystkie permutacje liczb $1, \ldots, n$. □
Dowód nie odnoszący się do pojęcia obliczenia zostanie przedstawiony poniżej.

Dowód. Dowód twierdzenia sprowadza się do udowodnienia następującego lematu.

Lemat 11.7. Założenia: A jest tablicą liczb całkowitych o rozmiarze n. Liczba n jest dodatnia $n > 0$, Zachodza równości $l = 1$ i $u = n$.
Liczba k spełnia warunek $0 \leq k - 1 \leq n$. Na k różnych miejscach tablicy A znajdują się liczby $1, \ldots, k$.
Teza. Jest twierdzeniem algorytmicznej teorii liczb naturalnych $\mathcal{ATN}$ uzupełnionej deklaracją procedury F - schematem aksjomatów $\{call\ F\}\alpha \Leftrightarrow \{TrescF\}\alpha$ następujące zdanie

$$\delta \Rightarrow \{k := k + 1; \text{call } F\}(\gamma)$$

Baza indukcji. Dla $n = 1$ teza twierdzenia jest oczywiście prawdziwa.
Krok indukcji. Udowodnimy następującą tezę.
Teza. Dana jest tablica A o rozmiarze większym od n. Tablica zawiera dokładnie j zer. Pozostałe miejsca są wypełnione liczbami dodatnimi $1, \ldots, l$. Wartością zmiennej k jest $j + 1$.

Załóżmy, że dla każdej liczby $n \leq j$ jest prawdą, że program oblicza i drukuje wszystkie permutacje zbioru $\{1, \ldots, j\}$. Rozpatrujemy zbiór $\{1, \ldots, j, j+1\}$. Wykonanie instrukcji call F w programie głównym powoduje utworzenie rekordu aktywacji tej procedury i zapoczątkowanie wykonywania instrukcji for. W tablicy A o $j+1$ elementach na miejscu 1 zostanie wstawiona liczba 1 (tzn. wartość zmiennej k).

□

ROZDZIAŁ 12

$\mathcal{L}_e$ Sygnały i ich obsługa

$$\mathcal{L}_0 \subsetneq \mathcal{L}_1 \subsetneq \mathcal{L}_2 \subsetneq \mathcal{L}_3 \subsetneq \mathcal{L}_4 \subsetneq \mathcal{L}_5 \subsetneq \mathcal{L}_6 \subsetneq \mathcal{L}_7 \subsetneq \mathcal{L}_e \subsetneq \mathcal{L}_8 \subsetneq \mathcal{L}_9 \subsetneq \mathcal{L}_{10}$$

Ten rozdział poświęcamy sygnałom, wyjątkom i ich obsłudze.
Podczas wykonywania obliczeń mogą wystąpić dwa niepożądane, wręcz groźne, zjawiska:
zapętlenie – obliczenie nieskończącesię czyli wymagające ingerencji, przerwania pracy programu, lub
zerwanie obliczeń przez komputer lub maszynę wirtualną VLP, wynikające z wykrytego błędu np. dzielenie przez zero, próba dostępu do nieistniejącego obiektu, próba zapisania wartości w nieistniejącym elemencie tablicy, ...
Mechanizm polegający na sygnalizowaniu sytuacji wyjątkowej i odpowiednim jej obsłużeniu jest pomocny w zwiększaniu odporności (ang. robustness) programu. Wiemy, jak groźnym zjawiskiem jest zapętlanie się, czyli wystąpienie obliczenia nieskończonego. Podobnie groźne jest zjawisko obliczenia nieudanego – zerwanego. Przykładem obliczenia zerwanego jest doprowadzenie do dzielenia przez zero.

Przykład 12.1. Wiele programów wykorzystuje strukturę danych stosy i posługuje się opearcjami: push, pop, top, empty na nich. Często się zdarza, że program wykonuje operację pop na pustym stosie lub operację push na stosie pełnym. Hakerzy świadoie wykorzystujatakie sytuacje. Można wymagać by każde wykonanie polecenia pop było poprzedzone sprawdzeniem czy stos jest pusty. Często wymaganie takie jest nierealistyczne. Natomiastprzewidujący programista zadeklaruje sygnał np. signal EmptStk i zapewni, że podczas wykonywania metody pop sygnał taki zostanie zgłoszony raize EmptStk. W przypadku gdy taki bład wystąpi, użytkownik zostanie ostrzeżony i będzie mógł odpowiednio zareagować. Np. wprowadzając deklarację modułu obsługi sygnału. O czym obszerniej opowiemy poniżej.

Nie powinno się pozostawiać takich zdarzeń bez odpowiedniej reakcji. Minimalną reakcją powinno być powiadomienie użytkownika programu o wystąpieniu błędu i zakończeniu pracy programu. (Ale komunikat"ten program popełnił bład i zostanie zakończony" to nie jest odpowiednia reakcja.) Komunikatowi powinna towarzyszyć diagnoza. Maszyna wirtualna

Loglanu dostarcza krôtką diagnozę o rodzaju błędu i miejscu wystąpienia błędu w programie. Programista posługując się mechanizmem sygnałów i ich obsługi może znacznie ulepszyć diagnostykę ewentualnych błędôw, a w wielu przypadkach zaprogramować odpowiednią reakcję na zasygnalizowany błąd.
Wiele językôw programowania nie sygnalizuje grożnych błędôw: próba dostępu do nieistniejącego elementu tablicy, próba dostępu do nieistniejącego obiektu to dwa przykłady z wielu podobnych.
Warto by programista przewidywał możliwość wystąpienia błędu np. dostępu do nieistniejącego elementu tablicy i by wstawił w odpowiednie miejsca moduły reakcji (obsługi) sytuacji wyjątkowych (błędôw).

Przykład 12.2. W podrozdziale 7.3 udowodniliśmy poprawność algorytmu bisekcja. Jeżeli programista wykorzysta ten algorytm i zadba przy tym by spełnione były założenia twierdzenia 7.7, to nie stanie się nic złego. Ale... jak pokazuje wieloletnie doświadczenie, programiści chętnie stosują gotowe algorytmy i rôwnocześnie lekceważą sprawdzanie czy spełnione są wymagania gwarantujące otrzymanie poprawnego wyniku. Popatrzmy jak to może wyglądać.

```
program ApplyBisection;
  unit bisection: function(a,b,eps:real;function f(x:real):real): real;
    ...
  end bisection;
  unit f1: function(y:real): real;
  begin
    result := 8.3*(y-7)*(y+11)*()y-3.3)*(y-2.1)
  end f1;
  unit f2: function(y:real): real;
  begin
    result := (3*y+7)/(y+5)
  end f2;
  var z,t:real;
begin

  t:= bisection(-20, 11, 0.00001, f1);
  writeln(t);
  z:= bisection(-20, 11, 0.00001, f2);
  writeln(z);
end
```

Co tu się może stać?

- Obliczanie miejsca zerowego funkcji $f1$ przebiega bezproblemowo.
- Podczas szukania miejsca zerowego funkcji $f2$ może dojść do dzielenia przez zero.

- Możemy też zaobserwować niekorzystne efekty nieciągłości funkcji $f2$.

Jak temu zaradzić? Poprôbujmy.

Przykład 12.3. Wprowadzamy deklaracje sygnałów i moduły ich obsługi.

```
program SafeBisection;
  signal DivZero, Discontin;
  unit bisection: function(a,b,eps:real;function f(x:real):real): real;
    ...
  end bisection;
  unit f1: function(y:real): real;
  begin
    result := 8.3*((y-7)*(y+11))*((y-3.3)*(y-2.1))
  end f1;
  unit f2: function(y:real): real;
    handlers
      when NumError: raise DivZero;
    end handlers;
  begin
    if abs(y+5 )<0.001 then raise Discontin fi;
    result :=(3*y+7)/(y+5)
  end f2;
  var z,t:real;
  handlers
    when DivZero: writeln(" naruszono warunki");
    when Discontin: writeln(" w pobliżu punktu nieciągłości");
  end handlers;
begin

  t:= bisection(-20,11,0.00001,f1);
  writeln(t);
  t:= bisection(-20,11,0.00001,f2);
  writeln(t);
end
```

W programie SafeBisection, jeśli dojdzie do dzielenia przez zero to sygnał o tym zdarzeniu będzie wygenerowny przez komputer (tj. hardware) i zostanie przejęty przez moduł obsługi w rekordzie aktywacji funkcji $f2$, w tym module zostanie podniesieny sygnał DivZero. Ten sygnał zostanie obsłużony w module obsługi sygnałów programu głównego. Natomiast sytuacja w której podczas obliczeń zbliżamy się do punktu nieciągłości funkcji $f2$ zostanie zasygnalizowana przez program raise Discontin.

Poniżej przytaczamy trzy przykłady w których maszyna wirtualna VLP sygnalizuje błąd

Przykład 12.4. Podajemy trzy z kilku możliwych sygnalizacji błędu zgłaszanych przez maszynę wirtualną

- Reakcja na dzielenie przez zero.
  ```
  division by zero error
  line 234
  ```
- Reakcja na polecenie utworzenia tablicy o ujemnej długości.
  ```
  array error
  line 234
  ```
- Reakcja na próbę odczytania/zmodyfikowania elementu tablicy spoza jej zasięgu.
  ```
  array index error
  line 234
  ```

Jeżeli w programie nie umieścimy modułu handlers przechwytującego i obsługującego sygnał o takim błędzie, to obliczenie programu zostanie zerwane z krótka informacją:
w wierszu XXX programu wystąpił błąd
Czasami jednak można wykorzystać informację o błędzie w konstruktywny sposób.

Przypomnijmy dwa algorytmy: obliczanie pierwiastka kwadratowego i rozwiązywanie układu równań liniowych. Udowodniliśmy, że gdy argument funkcji sqrt jest nieujemny to obliczenie algorytmu będzie skończone i otrzymany wynik będzie przybliżał dokładną wartość $\sqrt{a}$ z żądaną dokładnością ϵ. A co się stanie gdy argument a jest liczbą ujemną? Nietrudno się przekonać, że w tym przypadku obliczenie może być dowolnie długie (nieskończone) i oczywiście nie przynosi odpowiedzi. Jak się zabezpieczyć przed taką ewentualnością? Można każde wystąpienie instrukcji x:=sqrt(b) zastąpić instrukcją warunkową

if b>0 **then** x:=sqrt(b) **else** writeln("obliczenie pierwiastka liczby ujemnej") **fi**

. Ten sposób zabezpieczenia programu jest jednak męczący dla programisty, zmniejszający efektywność poprawnych obliczeń (sprawdzanie warunku to dodatkowy koszt), i zwiększający koszt wytworzenia programu (choćby przez to, że zmuszamy się do napisania dłuższego tekstu). Ponadto metoda ta nie zapobiega błędowi przypadkowego pominięcia sprawdzania warunku.
W drugim przykładzie sytuacja jest bardziej złożona. Podczas rozwiązywania układu równań może dojść do dzielenia przez zero. Oznacza to, że dany układ równań jest singularny (tzn. macierz współczynników przy niewiadomych jest niewłaściwa). Nie mamy jednak, żadnego łatwego testu pozwalającego orzec przed rozpoczęciem obliczeń, że układ równań nie posiada rozwiązania. Pozostaje więc tylko jedna droga postępowania: rozpocząć obliczenia np. według metody Gaussa i gdy dojdzie bo

zgłoszenia błędu dzielenia przez zero należy błąd taki zinterpretować w odpowiedni sposób. Np. zgłaszając błąd "macierz A jest niewłaściwa".
Te dwa przykłady pokazują niebezpieczeństwo obu błędów: zarówno błędu obliczenia nieskończonego jak i błędu zerwania obliczenia. Najlepiej by było udowodnić o każdym wystąpieniu nazewnika funkcyjnego $sqrt(\omega)$, że wartość wyrażenia ω jest nieujemna. W takim wypadku nie grozi wystąpienie błędu. Jeśli jednak nie potrafimy tego dokonać to można wysłać odpowiedni sygnał i przejąć go w stosownym module obsługi sygnału.
Przykład W pewnym bloku programu zawarto deklarację funkcji f, sygnału S

```
program Odporny;
  var x:real
begin
  block
    signal UjemnyArgument;
    unit sqrt: function(a:real):real;
      var x,d:real;
      const epsilon=0.00001;
    begin
      if a<0 then raise UjemnyArgument fi;
      d:=epsilon;
      while d >= epsilon do
        if d=epsilon
        then x:=(a+1)/2
        else x:=(x+a/x)/2
        fi;
        d:=(x-a/x)
      od
      result:=x
    end sqrt
  begin
    block
      handlers
          when UjemnyArgument: writeln("Alarm")
      end handlers
    begin
      x:= ...
      writeln(sqrt(x))
    end (* block *)
  end (* block *)
end
```

Mówiąc najkrócej moduł obsługi sygnału ma strukturę podobną do treści procedury. Deklaracja sygnału jest podobna

do nagłówka deklarowanej procedury.
Polecenie raise uruchamia protokół o takiej nazwie. Protokół raise różni się od protokołu call realizującego instrukcję procedury tym, że moduł obsługi sygnału wyszukiwany jest wzdłuż ścieżki strzałek DL.
Ma to znaczenie dla inżynierii programowania – pozwala bowiem reagować na błędy jakie mogą pojawiać się w trakcie wykonywania programu.

Uruchamianie protokołu raise jest raczej kosztowne i nie zalecamy stosowania instrukcji raise bez istotnej przyczyny. Czasami jednak użycie mechanizmu obsługi sygnałów gdy nie występuje żaden błąd może przynieść ciekawe rezultaty.

Przykład 12.5.

```
program MAZE;
  var A : arrayof arrayof boolean,
    i,n : integer,
    there_is_a_path : boolean;
  signal Found;

  unit PATH : procedure (i,j : integer);
  (* the procedure makes one move from(i,j) *)
  begin
    if A(i,j)
    then (* we can go through (i,j) field *)
      if i=n and j=n then raise Found fi;
      if i< n then call PATH(i+1,j) fi;
      if j< n then call PATH(i,j+1) fi;
    fi;
    last_will : write(i,j) ;
    (* the path from A[n,n] to A[1,1] will be printed *)
  end PATH;

  handlers
    when Found : there_is_a_path :=true; wind
  end handlers;
begin (* main program *)
  .... (* create a maze A etc. *)
    call PATH(1,1);
    if there_is_a_path then ...
    ...
end MAZE
```

1. Składnia

Elementy obsługi sygnałów są rozłożone w kilku miejscach programu: W programie muszą pojawić się:

a) deklaracja sygnału(-ów),
b) deklaracja modułu(-ów) obsługi sygnału handlers,

c) instrukcja raise wysłania sygnału,
d) instrukcja kończąca obsługę sygnału,
e) ostatnia wola lastwill, przygotowana na wypadek usunięcia jednostki dynamicznej (rekordu procedury, obiektu itp.).

Uprzedzamy Czytelnika, że praca z sygnałami i ich obsługa przebiega w sposób odmienny od dotychczas nabytych doświadczeń.
Sygnał S i obsługujący go moduł obsługi H, razem tworzą parę podobną do modułu procedury. Są jednak różnice: (i) sygnał S i handler H mogą znajdować się w różnych, odległych od siebie modułach programu, (ii) jeden sygnał S może być obsługiwany w wielu, różnych modułach programu.
Dokładniej, obsługa podniesionego sygnału odbywa się podczas obliczenia programu, może istnieć więcej jednostek dynamicznych zawierających moduły obsługi (handlery) niż deklaracji tych handlerów w tekście programu.
Kolejna odmiana jaką spotykamy, ucząc się sygnałów i ich obsługi, to nieoczywisty sposób odszukiwania miejsca w którym sygnał będzie obsłużony. W odróżnieniu od statycznego wiązania instrukcji I_P procedury z deklaracją procedury P, mamy teraz do czynienia z dynamicznym wiązaniem. Od miejsca w którym wystąpiło podniesienie sygnału rozpoczyna się poszukiwanie modułu H obsługi wzdłuż ścieżki DL.

Deklaracja sygnału. Sygnał S musi być zadeklarowany w programie i widoczny z miejsc w których będzie zgłaszany, a także w tych miejscach gdzie ulokowane zostaną moduły obsługi tego sygnału.
Wymaganie to nie odnosi się do sygnałów błędów wykrywanych i zgłaszanych automatycznie przez maszynę wirtualną VLP.
Deklaracja sygnału przypomina nagłówek deklaracji procedury.

Przykład:

```
signal Err12, Tablica_o_ujemnej_długości;
signal Err11(x:real, c:char);
```

Jak widać kilka deklaracji sygnału można poprzedzić jednym słowem signal. Zadeklarowany sygnał może mieć parametry formalne.

Moduł obsługi sygnałów. Moduł obsługi może znaleźć się w każdym module: bloku, procedurze, klasie, współprogramie.

```
handlers
 when Err12: writeln("Pomyłka");
 when Err13: ...
end handlers
```

W powyższym przykładzie pomiędzy wierszami handlers i end handlers zawarto dwa moduły obsługi: dla sygnałów Err12 i Err13. Trochę kłopotu może sprawiać zapamiętanie, że moduł obsługi sygnałów musi być ostatnią deklaracją modułu, tzn. występuje tuż przed słowem **begin**. Nie należy się tym przejmować, kompilator przypomni te zasady.

Zgłaszanie sygnału. Zgłaszanie (albo wysyłanie, albo podnoszenie) sygnału realizuje program wykonując polecenie **raise**.

```
raise Err13;
```

Kończenie obsługi. Ostatnią instrukcją wykonywaną podczas obsługi sygnału jest domyślnie polecenie **terminate**. Programista ma cztery sposoby zakończenia obsługi sygnału, są to

- **terminate**,
  - domyślny, najczęściej spotykany sposób zakończenia obsługi sygnału
- **wind**,
- **return**,
- **call** endrun
  - w ostateczności zakończ program

Ostatnia wola. Poniżej przeczytasz o zamykaniu rekordów aktywacji, jednostek dynamicznych w efekcie poleceń terminate i wind. Programista może przygotować polecenia poprzedzające zamknięcie takich jednostek dynamicznych. Polecenia ostatniej woli, (lub destruktora) umieszczamy jako ostatnie w tekście modułu (przed end), poprzedzając je etykietą lastwill:. Zob. przykład 12.5.

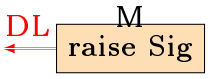
Przykład

2. Semantyka

Omówienie semantyki zaczynamy u źródła. Sygnał może być zgłoszony przez program (polecenie raise) lub przez maszynę wirtualną VLP wykonującą nasz program. Sygnałom zgłaszanym przez maszynę VLP nie towarzyszą parametry aktualne. Natomiast polecenie raise S może mieć formę niemal identyczną z instrukcją procedury, np. `raise S(5.4, 'x')`.

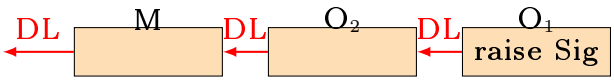
Protokół raise. Sygnał S i paczka zgromadzonych parametrów aktualnych, wędruje wzdłuż ścieżki DL w poszukiwaniu jednostki dynamicznej zawierającej moduł obsługi podniesionego sygnału

Sygnał może być podniesiony podczas wykonywania programu albo przez maszynę wirtualną VLP (np. MEMERROR) lub przez program wykonujący polecenie raise.
Przyjmijmy, że zgłoszenie sygnału S nastąpiło w trakcie wykonywania poleceń jednostki dynamicznej M (tzn. M jest albo rekordem aktywacji bloku, funkcji lub procedury) lub jest obiektem klasy w trakcie inicjalizacji (konstruktor).

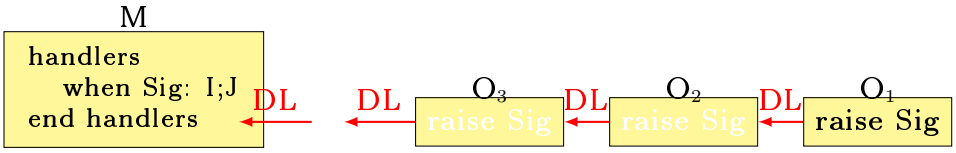


Jeżeli jednostka dynamiczna M zawiera moduł obsługi H dla sygnału S to ten moduł zostanie wykonany.
Dokładniej, tworzona jest instancje O_H tego modułu obsługi H, jej statycznym ojcem jest M, ojcem dynamicznym jest jednostka dynamiczna, w której zgłoszono sygnał S, w tym konkretnym przypadku SL=DL.
Jeśli jednostka dynamiczna M nie zawiera modułu obsługi dla sygnału S to kładziemy $M = M.DL$ czyli przechodzimy do dynamicznego ojca jednostki M. Można powiedzieć, że sygnał S jest propagowany do dynamicznego ojca M.

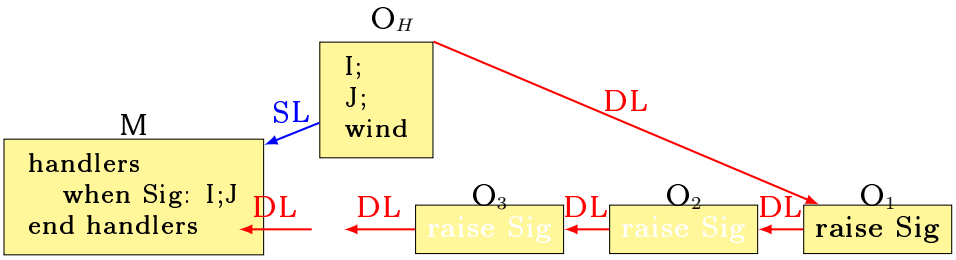
Są następujące modyfikacje tego schematu postępowania:
– jeśli jednostka M jest instancją modułu obsługi H, to to sygnał jest propagowany do jednostki zawierającej deklarację tego modułu obsługi. W tym przypadku w poszukiwaniu odpowiedniego modułu obsługi wykonywane jest przejście wzdłuż odnośnika SL.
– jeśli jednostka dynamiczna M jest obiektem współprogramu lub rekordem aktywacji programu głównego Main, to obliczenie programu jest kończone,
– jeśli M jest obiektem procesu, to to obiekt ten jest kończony i usuwany.



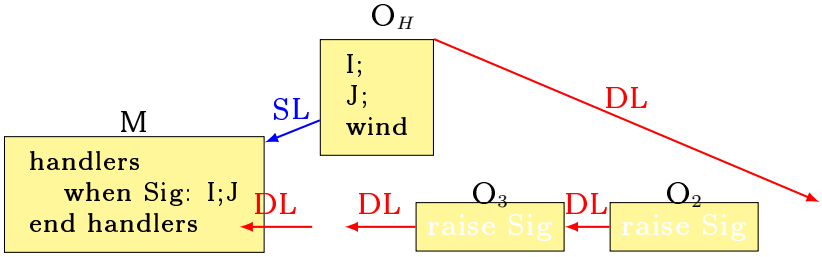
Gdy postępowanie to kończy się sukcesem tzn. gdy znaleziono jednostkę dynamiczną zawierającą moduł obsługi podniesionego sygnału, to tworzony jest rekord aktywacji tego modułu, jego dynamicznym ojcem zostaje jednostka w której zgłoszono sygnał, jego statycznym ojcem jest jednostka dynamiczna w której znaleziono moduł obsługi sygnału.



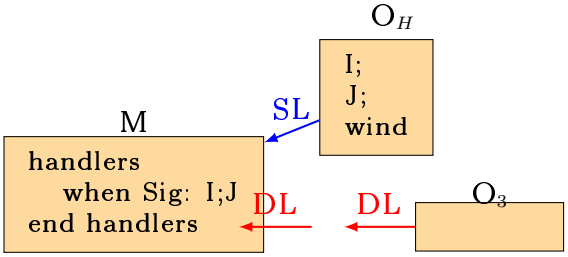
Następuje przekazanie parametrów aktualnych.
Wykonywane są po kolei instrukcje modułu obsługi.



Na koniec wykonywane jest jedno z poleceń:
wind – zamykane i usuwane są wszystkie jednostki dynamiczne poczynając od tej w której podniesiono sygnał S, raise S, a kończąc na rekordzie aktywacji O_H modułu obsługi błędu S. Wykonywanie programu jest kontynuowane w jednostce dynamicznej, która zawiera moduł obsługi błędu S.
terminate – zamykane i usuwane są wszystkie jednostki dynamiczne poczynając od tej w której podniesiono sygnał S, raise S, a kończąc na jednostce dynamicznej M, która zawiera moduł obsługi błędu S. Wykonywanie programu kontynuowane jest w jednostce dynamicznej wskazanej przez M.DL.



Usuwane są jednostki dynamiczne jedna po drugiej. Przed zlikwidowaniem jednostki wykonywana jest ostatnia wola, o ile o to zadbałeś.



return – to polecenie pozwala powrôcić do instrukcji następującej po poleceniu raise S. To polecenie może być zastosowane wyłącznie do obsługi sygnału zadeklarowanego przez program. call endrun – to polecenie kończy wykonywanie programu.

Jak zakończyć obsługę sygnału? Programista ma do dyspozycji cztery sposoby zakończenia obsługi sygnału lub błedu zgłaszanego przez system maszyny wirtualnej Loglanu:

- return – wykonanie tego polecenia wznawia obliczenie w miejscu następującym po instrukcji raise. Zwróć uwagę, polecenie return nie jest odpowiednim sposobem zakończenia obsługi błędu zgłoszonego prze zmaszyne wirtualna.
- terminate – wszystkie jednostki dynamiczne w których poszukiwno modułu handlera, łącznie z ta jednostka dynamiczna w której znaleziono moduł obsługi zostaną zakończone i usunięte. Obliczenie będzie kontynuowane w jednostce dynamicznej wskazanej przez odnośnik DL w jednostce, która zawierała moduł obsługi sygnału.
  Przy tej okazji wykonane zostana polecenia ostatniej woli lastwill.
  Polecenie terminate jest domyślnym sposobem kończenia obliczeń wewnatrz modułu obsługi sygnału – podobnie jka polecenia return domyślnie konczy obliczenie wewnatrz modułu procedury lub funkcji.
- wind – to polecenie ma efekt podny do polecenia terminate, z tą różnicą, że obliczenie jest kontynuowane w jednostce dynamicznej zawierającej moduł obsługi sygnału.
- call endrun – w ostateczności programista może zakończyć obliczenie programu.

lastwill. Gdy jednostka dynamicna jest kończone w sposób nienormalny przez wykonanie polecenia terminate lub wind, wykonywane sa polecenia, jakie programista przygotował na taką ewentualność.

dziedziczenie vs. obsługa sygnałów. Handlers (modules handling signals) behave similarly to virtual procedures, i.e. the new handler from the prefixed module replaces the module from the prefixing module. Example 6

Przykład 12.6.

```
unit STACKS: class (type :telem);
signal empty_stack(s:stack), stack_overflow(s:stack);
unit stack : class (size:integer);
hidden place, top;
var place : arrayof token,
top: integer;
unit pop: function:telem;
handlers
when conerror: raise empty_stack(this stack)
end handlers;
begin
result :=place(top);
(** here con_error signal can be raised by run_time_system **)
top:=top-1
end pop;
unit push : procedure (e:telem);
begin
if top> size
then
raise stack_overflow(this stack)
fi;
top:=top+1;
place(top):=e
end push;
unit empty: function : boolean;
begin
result:=top< 1
end empty;
unit increase : procedure (addition: integer);
var i : integer,
x : arrayof telem;
begin
array X dim (size+addition);
size:=size+addition;
for i:=1 to upper(place)
do
x(i):=place(i)
od;
kill(place);
place:=X
end increase.
begin
array place dim(1:size);
end stack;
handlers
when empty_stack: write(ęmpty stack"); terminate;
when stack_overflow: write(śtack overflow"); terminate;
when conerror : write(ęrror in stack increasing");
call endrun;
end handlers;

end STACKS;
```

Applications
Example 7

```
program APPLICATION_1;
unit STACKS: ...
unit element: ...
...
pref STACKS(element) block
var s1,s2 : stack
...
handlers
when stack_overflow : ...
call s.increase(70);
return;
end handler;
begin (*** block ***)
...
s1:= new stack(c1);
s2:= new stack(c2);
...
call s1.push(e);
...
y:=s2.pop;
...
end (*block*);
```

end APPLICATION_1

In this example the handler for stack_overflow from the prefixed block overrides the handler given in the class STACKS.

Example 8

```
program ReversePolishNotation; (* Application 2 *)
unit STACKS: class; ... end STACKS;
...
unit element : class(sign:char);
end element;
...
pref STACKS(element) block;
...
handlers
when empty_stack: write("
error in expression – too many ( closing brackets ");
terminate;
end handlers;

begin (*block *)
```

```
while not eof
do
read(x);
if x= ')' then
(* take operators from stack until ( is met *)
else
....
fi
od
end (*block*)
end ReversePolishNotation;
```

This example shows that there are situations in which the user of a class STACKS knows how to handle the signal empty_stack and solves the problem gracefully since in this case empty_stack means that the data were not a well formed expression.

3. Przykłady

Przykład sygnał dzielenie przez zero może pojawić się podczas wykonywania różnych fragmentów programu.
– podczas obliczania wartości pewnej funkcji wymiernej i oznacza że argument funkcji nie należy do dziedziny funkcji,
– podczas rozwiązywania układu równań metodą Gaussa i oznacza, że macierz A jest osobliwa.

4. Porady i wnioski

Gdzie umieścić moduł obsługi sygnału?
Najlepiej wpisać go w najmniejszy moduł zawierający ciąg C instrukcji, taki, że pojawienie się wyjątku S podczas wykonywania którejkolwiek instrukcji I z ciągu C ma być obsłużone w ten sam sposób. Modułem takim może być procedura P (jej deklaracja) lub blok.

Gdzie umieścić deklarację sygnału?
Pamiętajmy deklaracja sygnału odpowiada (jest podobna do) nagłówkowi deklaracji procedury. Moduły obsługi tego sygnału (handlery) to rozmaite treści realizujące wezwanie do obsługi sygnału. Sygnał przypomina deklarację procedury z wieloma treściami tej procedury ulokowanymi w różnych miejscach programu. Jest to wygodne i racjonalne jeśli pamiętamy, ze szukanie odpowiedniej reakcji na sytuację wyjątkową (błąd) odbywa się wzdłuż ścieżki DL.
Zawsze bezpiecznie można zadeklarować sygnał w głównym bloku programu tj. po słowie program. Warto też w bloku głównym umieścić moduł obsługi tego sygnału. Taki moduł jest ostatnią deską ratunku. Jeśli go zabraknie to możemy zostać z przerwanym obliczeniemi raczej kiepska diagnostyką że w programie nie znaleziono modułu obsługi sygnału. Ale nie

jest to konieczne.
Obsługa sygnałów jest dość kosztowna – trzeba odszukać odpowiedni moduł obsługi sygnału. W tej sytuacji pożądane jest udowodnienie, że błąd np. "array index error " nie wystąpi. Zauważ, wykorzystanie funkcji, lower() i upper(), w poniższej instrukcji

for i :=lower(A) to upper(A) do A(i):= ω od

może pomôc w przeprowadzeniu dowodu, że nie wystąpi ten sygnał błędu.

Porównanie protokołów call i raise. Jak najkrôcej opisać rôżnicę pomiędzy instrukcją procedury (i wywołania funkcji) call, a instrukcją podniesienia sygnału raise?
Mozemy powiedzieć call odszukuje potrzebną deklaracjęprocedury (lub funkcji) wzdłuż ścieżki modułów zawierających instrukcję procedury. Protokół raise odszukuje potrzebny moduł obsługi sygnału wzdłuż ścieżki DL ... Statyczne dowiązania mogą byż wstępnie wyznaczone podczas kompilacji programu i cały mechanizm się uprszcza. Natomiast przed wykonaniem programu nie sposób określič dokąd trafimy poszukując modułu obsługi sygnału.

O naśladowaniu instrukcji try ... catch C++ i Javy. napisać

Część 2

Programuj z klasą
lub
programowanie obiektowe