Fundusze Europejskie Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Politechnika Warszawska
Unia Europejska Europejski Fundusz Społeczny

Metody Translacji

Jan Bródka

Wydział Matematyki i Nauk Informacyjnych

Materiały pomocnicze do wykładów

Projekt „NERW 2 PW. Nauka – Edukacja – Rozwój – Współpraca” współfinansowany jest ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego.

Zadanie 10 pn. „Modyfikacja programów studiów na kierunkach prowadzonych przez Wydział Matematyki i Nauk Informacyjnych”, realizowane w ramach projektu „NERW 2 PW. Nauka – Edukacja – Rozwój – Współpraca”, współfinansowanego ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego.

Spis treści

1. Wprowadzenie

Rozwój języków programowania przebiegał od języków bezpośrednio związanych ze sprzętem (i niezbyt wygodnych w użyciu) do języków wysokiego poziomu, które są niezależne od konkretnego typu komputera i zawierają abstrakcyjne konstrukcje niemające bezpośrednich odpowiedników w rozkazach maszynowych ani typach danych, które mogą być bezpośrednio przetwarzane przez rozkazy maszynowe. Ogólnie języki wysokiego poziomu mają umożliwiać programiście formułowanie algorytmów za pomocą abstrakcyjnych konstrukcji bez konieczności wnikania, w jaki dokładnie sposób te konstrukcje są realizowane przez sprzęt.
Procesu tłumaczenia programu z języka źródłowego, wygodnego dla człowieka, na język wynikowy, który może być bezpośrednio wykonywany przez sprzęt, nie można jednak pominąć. Ale na szczęście można go całkowicie zautomatyzować.

Proces ten, nazywany translacją, może przebiegać na dwa podstawowe sposoby.

A) Proces tłumaczenia przeplata się z wykonaniem programu.
Tłumaczymy fragment programu (zwykle pojedynczą jednostkę syntaktyczną, np. instrukcję), a następnie ją wykonujemy. Ten proces tłumaczenia i wykonania powtarzany jest dla kolejnych fragmentów. Translator działający w ten sposób to **interpreter**.

B) Najpierw tłumaczymy całość programu z języka źródłowego na docelowy, a następnie wykonujemy program w całości. W tym przypadku faza tłumaczenia nazywa się kompilacją, a wykonujący ją translator to **kompilator**.

Oczywiście istnieje również wiele modeli mieszanych. Np. w interpreterach zwykle język źródłowy nie jest tłumaczony wprost na rozkazy maszynowe, a na kod pośredni, który następnie wykonywany jest przez maszynę wykonawczą stanowiącą część interpretera. Pomysł specjalnych maszyn wykonawczych, nazywanych w tym przypadku maszynami wirtualnymi (np. wirtualna maszyna Javy, wirtualna maszyna .NET), dotyczy również kompilacji. W tym przypadku zachowana jest podstawowa zasada kompilacji, czyli przetworzenie całego kodu źródłowego przed rozpoczęciem wykonania, ale efektem tego przetwarzania nie są rozkazy maszyny rzeczywistej, a rozkazy maszyny wirtualnej.

Poniższa tabela zawiera porównanie cech interpreterów i kompilatorów

interpretery	**kompilatory**
większa elastyczność (w tym możliwość modyfikacji programu źródłowego podczas jego wykonywania), ceną za to jest wolniejsze wykonanie programu	większa szybkość wykonywania programu
słaba kontrola poprawności programu - wiele błędów ujawnia się dopiero podczas wykonania programu	znacznie lepsza kontrola poprawności programu - wiele błędów można wykryć już na etapie kompilacji (przed rozpoczęciem wykonania programu)
zwykle tzw. "wiązanie dynamiczne"	zwykle tzw. "wiązanie statyczne"
zwykle tzw. "dynamiczny system typów"	zwykle tzw. "statyczny system typów"
bardzo ograniczone możliwości optymalizacji	znacznie większe możliwości optymalizacji
prostota implementacji	bardziej skomplikowana implementacja

Krótkiego wyjaśnienia wymagają pojęcia statycznego/dynamicznego wiązania i statycznego/dynamicznego systemu typów.

Pojęcie wiązania statycznego bądź dynamicznego dotyczy sposobu interpretowania znaczenia nielokalnych identyfikatorów (zwykle zmiennych). Zmienne nielokalne to zmienne, które nie są zadeklarowane w bieżącej funkcji/procedurze/metodzie (w odniesieniu do kompilacji określeń funkcja/procedura/metoda używa się zamiennie). W takiej sytuacji ich deklaracji poszukuje się w tak zwanym zakresie zewnętrznym, a różnica pomiędzy wiązaniem statycznym i dynamicznym polega właśnie na różnym określeniu co jest tym zakresem zewnętrznym.
W przypadku wiązania statycznego zakresem zewnętrznym jest procedura, wewnątrz której zdefiniowana została procedura z rozważaną zmienną nielokalną (język musi umożliwiać zagnieżdżanie procedur - najbardziej znany przykład to Pascal). Zatem dla określenia znaczenia zmiennej nielokalnej należy analizować statycznie kod źródłowy programu.
W przypadku wiązania dynamicznego zakresem zewnętrznym jest procedura, z wnętrza której wywołana została procedura z rozważaną zmienną nielokalną. Zatem dla określenia znaczenia zmiennej nielokalnej należy prześledzić przebieg wykonania programu (i dynamiczną strukturę wywołań procedur).
Dynamiczne wiązanie zmiennych może znacząco utrudnić wyszukiwanie błędów (w skrajnym przypadku ten sam identyfikator może, w zależności od wcześniejszej ścieżki wykonania, oznaczać różne zmienne!)

Pojęcie statycznego bądź dynamicznego systemu typów wiąże się możliwością statycznego (czyli jedynie na podstawie analizy kodu źródłowego programu, a bez analizy przebiegu wykonania programu) określenia typu każdej zmiennej, a w konsekwencji każdego wyrażenia w programie.
Języki, w których takie określenie jest możliwe, to języki ze statycznym systemem typów. Takie języki zwykle wymagają deklaracji zmiennych zanim zostaną one użyte (a ta deklaracja musi zawierać określenie typu zmiennej).
Języki, w których dla określenia typów zmiennych (i w konsekwencji typów wyrażeń) wymagane jest śledzenie przebiegu wykonania programu, to języki z dynamicznym systemem typów. Języki te mogą wymagać lub nie deklarowania zmiennych przed użyciem, ale nawet jeśli deklaracje są wymagane, to nie muszą zawierać określenia typu. Typ zmiennej określany jest przy pierwszym przypisaniu wartości do tej zmiennej (i jest to typ owej wartości). Tak wyznaczony typ (w zależności od języka) może być już niezmienny do końca wykonania programu (ale dla różnych ścieżek wykonania może być różny, bo pierwsze przypisanie może być różne), albo może zmieniać się wraz z kolejnymi przypisaniami.
Podobnie jak w przypadku wiązania dynamicznego, wyszukiwanie przyczyn błędów w językach z dynamicznym systemem typów może być znacznie trudniejsze niż dla języków ze statycznym systemem typów (ale w prostych przypadkach możliwość użycia zmiennej bez konieczności wcześniejszej jej deklaracji może być wygodna).

W związku z zaletami języków kompilowanych w porównaniu z językami interpretowanymi w dalszej części opracowania rozważane będą jedynie języki kompilowane i proces kompilacji.

W procesie kompilacji można wyróżnić dwie główne fazy.

A) **Faza analizy** - w tej fazie analizowany jest program źródłowy. Jest on przekształcany z postaci tekstowej (kod źródłowy) do reprezentacji pośredniej, zwykle w postaci drzewa. Oprócz drzewa (reprezentującego operacje wykonywane w programie) powstaje również druga struktura danych nazywana tablicą symboli (zawiera ona informacje o wszystkich identyfikatorach występujących w programie). W modelowej sytuacji (a takiej będziemy się trzymać) faza analizy zależy jedynie od języka źródłowego (cechy maszyny docelowej nie mają żadnego znaczenia).

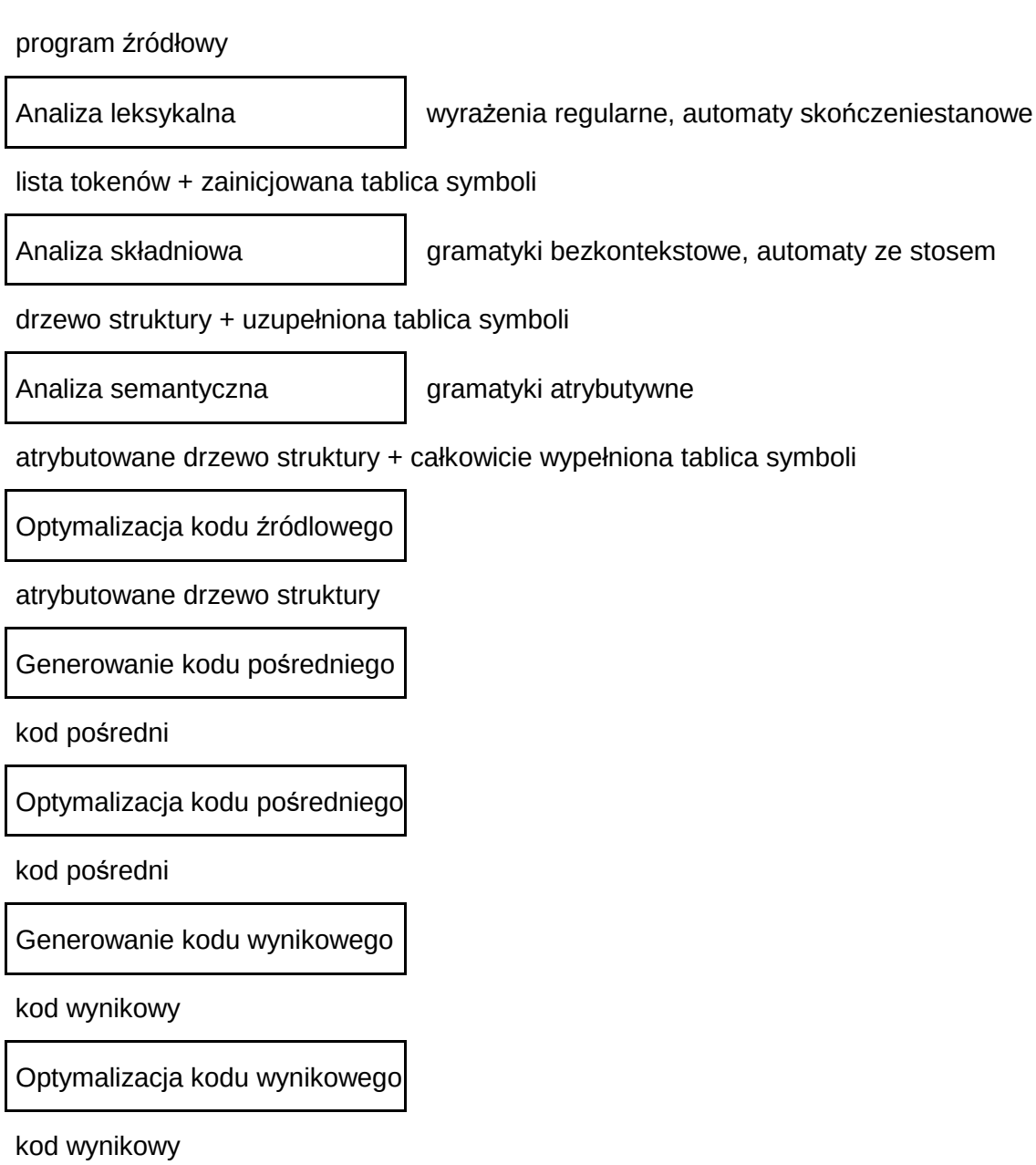
B) **Faza syntezy** - w tej fazie na podstawie reprezentacji pośredniej generowany jest kod wynikowy, elementami tej fazy są również różnego rodzaju optymalizacje. W modelowym przypadku faza ta zależy jedynie od cech maszyny docelowej, język źródłowy nie ma znaczenia. W praktyce jednak pewne cechy języka źródłowego mają wpływ na generowanie kodu (będzie o tym w dalszej części wykładu).

Taki podział na oddzielne fazy znacznie ułatwia tworzenie kompilatora, a jeszcze bardziej wielu kompilatorów. Rozważmy sytuację N języków źródłowych i M maszyn docelowych. Gdybyśmy chcieli stworzyć monolityczne kompilatory z każdego języka źródłowego na każdą maszynę docelową to musiałoby ich być N*M. Natomiast w przypadku oddzielnych modułów (i dobrze zdefiniowanej reprezentacji pośredniej) wystarczy N analizatorów (faza analizy) i M generatorów kodu (faza syntezy), czyli razem N+M modułów, aby stworzyć wszystkie pożądane kompilatory (a dla N,M > 2, N+M jest przecież mniejsze niż N*M).

Warto również zauważyć, że zarówno program źródłowy, jak i wynikowy, ma strukturę liniową (sekwencja instrukcji źródłowych lub wynikowych), natomiast reprezentacja pośrednia jest nieliniowa - drzewiasta.

Wymienione powyżej główne fazy procesu kompilacji można podzielić na kolejne fazy niższego poziomu (i zwykle mówiąc o fazach procesu kompilacji ma się na myśli właśnie ten bardziej szczegółowy podział). Jest on zaprezentowany na poniższym schemacie.

Fazy procesu kompilacji



Poniżej przedstawiony jest krótki opis każdej z faz.

Analiza leksykalna

Dane dla analizy leksykalnej stanowi tekst źródłowy programu (to jedyna faza, która ma bezpośrednią styczność z kodem źródłowym), a wynikiem jest ciąg tokenów (nazywanych również symbolami leksykalnymi lub leksemami). Pojedynczy token to np.: identyfikator, liczba (całkowita lub zmiennopozycyjna), separator (np. ;), operator, nawias, itp. Tokeny mogą być opisane przez wyrażenia regularne, a co za tym idzie mogą być rozpoznawane przez automaty skończeniestanowe. Istnieje więc dobrze ugruntowana teoria leżąca u podstaw analizy leksykalnej. Powstały również narzędzia umożliwiające automatyczne generowanie

analizatorów leksykalnych (nazywanych również skanerami) na podstawie podanych wyrażeń regularnych. Najbardziej znanym takim generatorem jest FLEX.

Rolą analizatora leksykalnego jest również zainicjowanie tablicy symboli oraz zwykle oddzielenie słów kluczowych języka od pozostałych identyfikatorów (słowa kluczowe zwykle "pasują" do wyrażenia regularnego opisującego identyfikatory).

Przykład.
Wyrażenie:

```
a[idx] = x + 2.5
```

składa się z następujących tokenów

```
a     - identyfikator
[     - nawias otwierający
idx   - identyfikator
]     - nawias zamykający
=     - operator przypisania
x     - identyfikator
+     - operator dodawania
2.5   - liczba
```

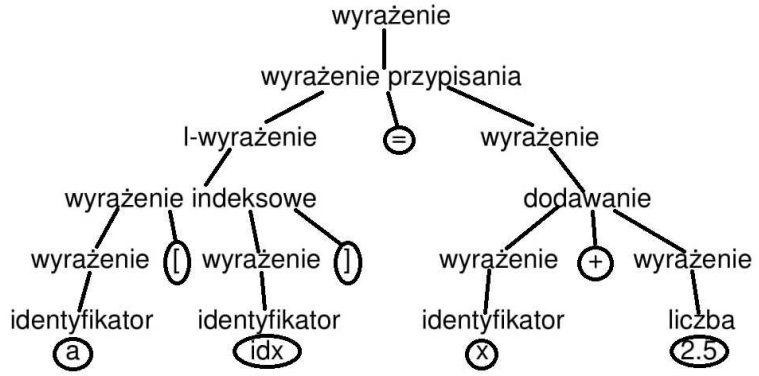
Analiza składniowa

Analiza składniowa, nazywana również syntaktyczną, polega na rozpoznaniu składniowej struktury programu. Jest podobna do gramatycznej analizy zdań w języku naturalnym. Dane dla analizy składniowej stanowi uzyskany w wyniku analizy leksykalnej ciąg tokenów, a wynikiem jest tak zwane drzewo struktury (ang. syntax tree). Moduł analizy składniowej nazywany jest również parserem. Podstawą teoretyczną analizy składniowej są gramatyki bezkontekstowe. Podobnie jak w przypadku analizatorów leksykalnych istnieją generatory analizatorów składniowych, które na podstawie zadanej gramatyki bezkontekstowej generują odpowiedni automat ze stosem. Najbardziej znanym takim generatorem jest YACC (Yet Another Compiler-Compiler) i jego następca Bison.

Dla podanego wcześniej przykładowego wyrażenia

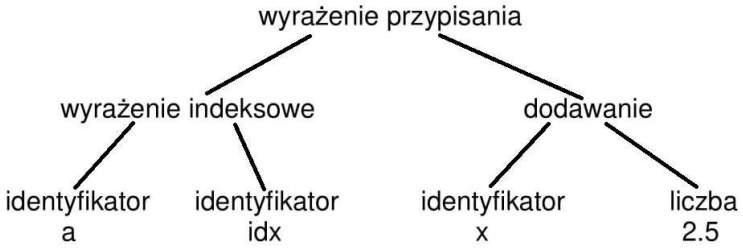
```
a[idx] = x + 2.5
```

Mamy następujące drzewo wywodu gramatycznego (ang. parse tree)



Drzewo wywodu reprezentuje tak zwaną składnię konkretną i zawiera elementy języka źródłowego zbędne z punktu widzenia dalszej analizy. Na przykład nie ma żadnego znaczenia, czy przypisanie oznaczane jest przez = (jak w C), czy przez := (jak w Pascalu), całkowicie wystarczy informacja, że jest to przypisanie.

Do dalszej analizy lepsze jest drzewo struktury (ang. syntax tree) reprezentujące tak zwaną składnię abstrakcyjną. W drzewie struktury elementy zbędne w dalszej analizie są pominięte. Dla podanego przykładu drzewo struktury jest następujące



I takie właśnie drzewo powstaje w wyniku analizy składniowej podanego przykładu.

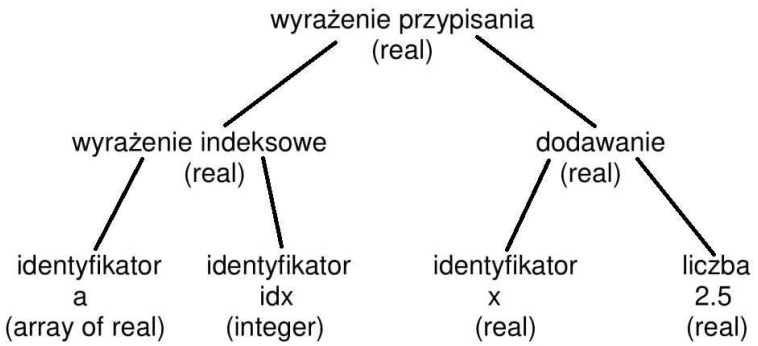
Analiza semantyczna

Zadaniem analizy semantycznej jest dodanie do konstrukcji programowych "znaczenia" (semantyki), które nie może być wyprowadzone z gramatyki bezkontekstowej, czyli nie może być elementem analizy składniowej.
Warto zauważyć, że pełne znaczenie programu może być rozpoznane jedynie przez jego wykonanie - jest to tak zwana semantyka dynamiczna. Jednak pewne elementy wykraczające poza gramatyki bezkontekstowe mogą być rozpoznane bez wykonywania programu - jest to tak zwana semantyka statyczna i ona właśnie jest przedmiotem analizy semantycznej.

Najważniejszym przykładem semantyki statycznej jest analiza typów zmiennych i wyrażeń występujących w programie (oraz zgodności tych typów). W językach z silnym statycznym systemem typów (a takimi będziemy się zajmować) każda zmienna musi być zadeklarowana, a deklaracja musi określać typ zmiennej (stały podczas całego wykonania programu). Na tej podstawie podczas analizy semantycznej wyznaczane są typy wszystkich wyrażeń w programie oraz badana jest zgodność typów elementów tych wyrażeń. Innymi słowy drzewo struktury uzupełniane jest o atrybuty określające typ wartości związanej z każdym z jego węzłów. Takie drzewo nazywane jest atrybutowanym drzewem struktury.
Jeśli dla rozważanego przykładowego wyrażenia

```
a[idx] = x + 2.5
```

założymy dodatkowo, ze
a - jest zadeklarowane jako tablica o elementach typu real
idx - jest zadeklarowane jako integer
x - jest zadeklarowane jako real
to odpowiednie atrybutowane drzewo struktury będzie miało postać



W przypadku wykrycia niezgodności typów analizator semantyczny sygnalizuje odpowiedni błąd.

Zauważmy, że dla języków z dynamicznym systemem typów wykonanie tego rodzaju analizy typów nie jest możliwe podczas kompilacji, a dopiero podczas wykonania programu (co spowalnia wykonanie i opóźnia wykrycie ewentualnych błędów).

Podstawą teoretyczną dla analizy semantycznej są zwykle tak zwane gramatyki atrybutywne (ang. attribute grammars), niestety nie ma narzędzi do automatycznego generowania analizatorów semantycznych.

Optymalizacja kodu źródłowego

Ta faza określana jest również jako optymalizacja niezależna sprzętowo (ang. machine-independent optimization), co lepiej odpowiada jej istocie, ponieważ faza ta wcale nie operuje na kodzie źródłowym, a na drzewie struktury.

Najbardziej znanym przykładem jest tak zwane zwijanie stałych (ang. constant folding), które polega na tym, że fragment drzewa struktury składający się wyłącznie z operacji na stałych zastępowany jest pojedynczym węzłem w drzewie.

Oprócz optymalizacji w fazie tej można również wykrywać konstrukcje potencjalnie błędne, np. użycie niezainicjowanej zmiennej lub tak zwany martwy kod (ang. dead code), czyli kod, który nigdy nie zostanie wykonany.

Generowanie kodu pośredniego (ang. intermediate code)

Celem tej fazy jest przekształcenie reprezentacji programu z drzewiastej na liniową opisującą sekwencję operacji do wykonania. Operacje te powinny być dość zbliżone do rozkazów rzeczywistej maszyny docelowej aby ułatwić dalsze fazy translacji, jednocześnie powinny zachować pewien poziom abstrakcji ukrywając "niewygodne" cechy maszyny docelowej (np. ograniczoną liczbę rejestrów, specyficzne tryby adresowania pamięci). Ponadto jako pojedyncze operacje kodu pośredniego można przyjąć również operacje wykonywane w rzeczywistości przez system operacyjny (np. operacje wejścia/wyjścia).

Optymalizacja kodu pośredniego

To najważniejsza ze wszystkich faz optymalizacji, w rzeczywistości jednym z głównych celów wprowadzenia kodu pośredniego jest właśnie ułatwienie optymalizacji. Przykładowe techniki optymalizacyjne stosowane w tej fazie to eliminowanie wspólnych podwyrażeń i przesuwanie niezmiennych obliczeń przed pętle.

Generowanie kodu wynikowego (ang. target code)

W tej fazie generowane są rozkazy maszyny rzeczywistej. Zwykle jest to kod binarny, ale szczególnie do celów dydaktycznych, mogą być również generowane asemblerowe mnemoniki rozkazów maszynowych.

Optymalizacja kodu wynikowego

W tej fazie ogólniejsze formy rozkazów wygenerowane w poprzedniej fazie zastępowane są przez bardziej wydajne w konkretnym przypadku rozkazy. Na przykład dla liczb całkowitych

mnożenie przez potęgi 2 może być zastąpione przez znacznie szybsze przesunięcie bitowe. Inny ważny przykład to optymalizacja rozkazów skoków.

Fazy generowania kodu i optymalizacji wymagają jeszcze wspólnego komentarza.

- Wszystkie fazy optymalizacji są opcjonalne i jako takie mogą być pominięte (zwróćmy uwagę, że żadna z optymalizacji nie zmienia reprezentacji programu)
- W przypadku kodu pośredniego część optymalizacji może być przesunięta do fazy generowania kodu (po prostu od razu generowany jest "lepszy" kod)
- W związku z upowszechnieniem się koncepcji maszyn wirtualnych obecnie często nie rozróżnia się kodu pośredniego i wynikowego, po prostu generowany jest od razu kod rozkazów maszyny wirtualnej.

W rzeczywistych kompilatorach przebieg kompilacji może przebiegać nieco inaczej - poszczególne fazy mogą się przeplatać. Szczególnie dotyczy to analizy leksykalnej i składniowej - praktycznie nie zdarza się, aby analizator leksykalny przetwarzał od razu cały program źródłowy - zwykle analizator składniowy wywołuje procedurę analizatora leksykalnego, gdy potrzebuje pobrać do analizy kolejny token. W przypadku tak zwanych kompilatorów jednoprzebiegowych analizator składniowy stanowi szkielet całego procesu kompilacji i w razie potrzeby wywołuje procedury realizujące kolejne fazy kompilacji.

Na zakończenie części wprowadzającej warto wspomnieć, że języki wysokiego poziomu można podzielić na kilka klas i jedynie część z tych klas (a w zasadzie jedna) będzie rozważana w dalszej części wykładu.

Ta rozważana klasa to języki imperatywne (znane przykłady to Algol, FORTRAN, Pascal, C). W takich językach program definiowany jest w postaci sekwencji operacji (instrukcji), które mają być wykonane w celu obliczenia wyniku. Podobną cechę posiadają najpopularniejsze języki obiektowe takie jak C++, C#, Java. Moim zdaniem popularność tych języków obiektowych związana jest właśnie z tym, że nie zrywają z koncepcją języków imperatywnych, a dodają do niej elementy charakterystyczne dla programowania obiektowego.
Nie będziemy natomiast rozważać następujących klas języków:

- języki deklaratywne - np. SQL
  kod źródłowy nie zawiera opisu operacji prowadzących do celu, a pożądane relacje między danymi - rola kompilatora jest inna niż w przypadku języków imperatywnych
- języki funkcyjne - np. Lisp, Haskell, F#
  mają silne podstawy matematyczne (rachunek lambda - w matematyce to dużo więcej niż funkcje lambda z C++ czy C#), jednak w klasycznej postaci wymagają zerwania z koncepcjami znanymi z języków imperatywnych (np. nie ma operacji przypisania)
- języki programowania w logice - np. Prolog
  w zasadzie Prolog to język deklaratywny, ponieważ program również opisuje relacje pomiędzy danymi, ale nie chodzi tu o dostęp do baz danych. Istotą Prologu są mechanizmy automatycznego wnioskowania i są one wbudowane w translator, co sprawia, że jest on zupełnie inny niż dla języków imperatywnych
- języki "ideologicznie" obiektowe - np. Smalltalk
  wszystko jest obiektem, a obiekty wysyłają do siebie komunikaty, np. w Smalltalku nie ma pojęcia instrukcji, są natomiast obiekty "blok kodu", do których można wysłać komunikat, w odpowiedzi na który obiekt "blok kodu" wykona się
- języki "dynamiczne" - np. Python, JavaScript
  mają wiele cech interpreterów (a interpreterami nie zajmujemy się)

2. Wybrane cechy języków programowania wysokiego poziomu

Niektóre cechy języków wysokiego poziomu mają wpływ nie tylko na etap analizy, ale również na generowanie kodu. Cechy te zostaną pokrótce omówione.

Zaczniemy jednak od omówienia różnych rodzajów zmiennych. Podział na kategorie będzie dotyczył czasu istnienia zmiennych.

1) Zmienne statyczne - istnieją przez cały czas wykonania programu. Nazwa bierze się stąd, że mają one (zwykle) na stałe przyporządkowaną lokalizację w pamięci. Dla twórcy kompilatora zmienne statyczne są łatwe w obsłudze (czyli mało interesujące).

2) Zmienne automatyczne (lokalne) - czasem ich istnienia jest okres wykonania pewnej konstrukcji programowej, zwykle procedury lub bloku. Są tworzone przy wejściu do owej konstrukcji programowej i likwidowane przy jej opuszczaniu. W przypadku rekurencyjnych wywołań funkcji/procedury każde wywołanie ma własny zestaw zmiennych lokalnych w związku z tym adresy tych zmiennych są dynamicznie wyznaczane podczas wykonania programu. Ta cecha sprawia, że obsługa zmiennych lokalnych w kompilatorze jest bardziej skomplikowana. Ten rodzaj zmiennych jest najbardziej interesujący dla twórcy kompilatora.

3) Zmienne nieograniczone - są tworzone podczas wykonania programu i nigdy nie są jawnie likwidowane. W starszych językach oznaczało to, że istnieją do końca wykonania programu (stąd nazwa zmienne nieograniczone), nowsze języki (lub środowiska wykonawcze) zwykle dysponują mechanizmami odzyskiwania pamięci (odśmiecania), które mogą zlikwidować takie zmienne. Mechanizmy odzyskiwania pamięci to bardzo ważny element, jednak pozostający nieco z boku głównej linii działania kompilatora i w tym opracowaniu nie będą omawiane (podobnie jak zmienne nieograniczone).

4) Zmienne sterowane (kontrolowane) - są jawnie tworzone i likwidowane przez programistę i w związku z tym kompilator (ani jego twórca) się nimi nie zajmuje. Czasem kompilator może próbować wykryć użycie niezainicjowanej zmiennej (tym problemem również nie będziemy się zajmować).

W przypadku zmiennych automatycznych oprócz czasu istnienia zmiennej należy zdefiniować również zakres dostępności (widoczności) zmiennej.
Zwykle przyjmuje się, że zmienna jest dostępna w zasięgu obejmującym konstrukcję programową (procedurę lub blok), w której została zadeklarowana (począwszy od miejsca deklaracji) oraz wszystkie zakresy wewnętrzne za wyjątkiem tych, w których zadeklarowana została zmienna o tej samej nazwie (przesłonięcie - niektóre języki np. C# nie pozwalają na przesłonięcie, zgłaszają błąd).

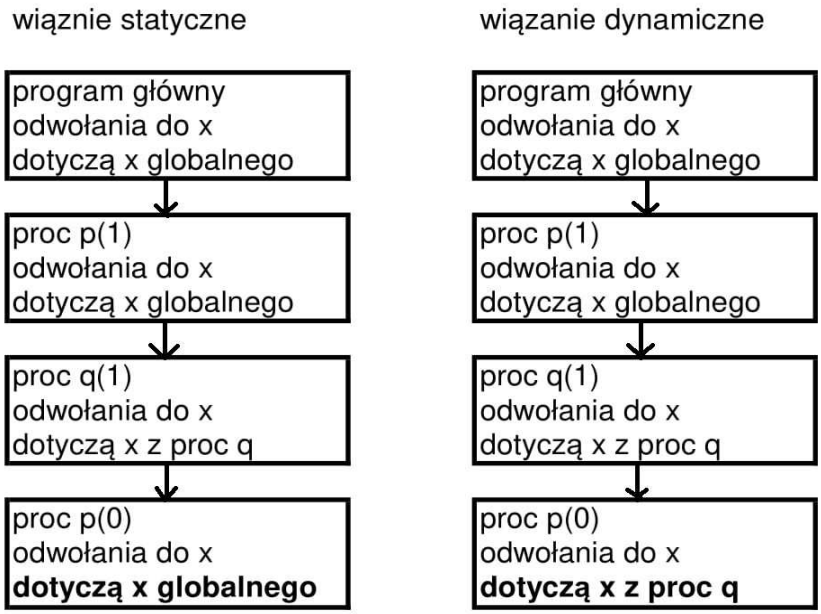
Zwykle przyjmuje się, że pojęcia zasięg wewnętrzny/zewnętrzny odnosi się do sposobu deklaracji procedury. To znaczy, że procedura zadeklarowana wewnątrz innej procedury tworzy zasięg wewnętrzny. Są to tak zwane **zasięgi statyczne** (static scoping rule), a odpowiadające im zasady dostępności zmiennych opisują wiązanie statyczne (static binding).

Drugą możliwością są zasięgi dynamiczne (dynamic scoping rule) i związane z nimi wiązanie dynamiczne (dynamic binding). W tym przypadku o tym, który zasięg jest wewnętrzny lub zewnętrzny względem innego, nie decydują deklaracje procedur, a sekwencja wywołania. Procedura wywołana z innej procedury tworzy zasięg wewnętrzny (chociaż mogą być zadeklarowane na tym samym poziomie).

W przypadku odwołań do zmiennych nielokalnych (nie zadeklarowanych w aktualnym zakresie) poszukiwane są zmienne o danej nazwie w "kolejno coraz bardziej zewnętrznych" zakresach. Tutaj pojawiają się różnice przypadku statycznego i dynamicznego wiązania zmiennych.

Przykład

```
program
   {
   int x;
   proc p(int n)
      {
      proc q(int k)
         {
         int x;
         call p(k-1);
         }
      x = n;
      if ( n>0 ) call q(n);
      }
   call p(1);
   }
```



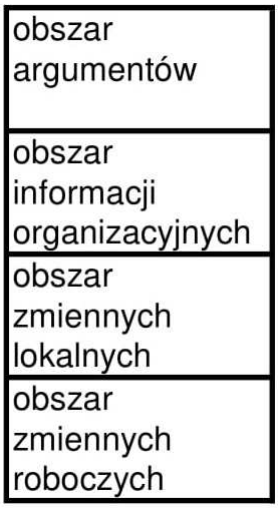
Zauważmy, że w przypadku wiązania statycznego wszystkie odwołania do zmiennej x w procedurze p dotyczą zmiennej globalnej, natomiast dla wiązania dynamicznego znaczenie odwołań do x w procedurze p zależy od historii wywołań (ta sama linia w programie może oznaczać dostęp do różnych zmiennych!), to może prowadzić do trudnych do usunięcia błędów.

Warto również zauważyć, że wiązanie dynamiczne wymaga, aby informacje z tablicy symboli były dostępne podczas wykonania programu, podczas gdy dla wiązania statycznego tablica symboli jest niezbędna jedynie podczas kompilacji.

Ponadto wiązanie dynamiczne pociąga za sobą dynamiczny system typów (co dodatkowo spowalnia wykonanie i może opóźnić wykrycie błędów niezgodności typów).

W dalszej części wykładu (i tego opracowania) **będziemy zajmować się wyłącznie statycznym wiązaniem zmiennych**.

Kolejnym elementem wymagającym omówienia jest tak zwany rekord aktywacji procedury. Struktura rekordu aktywacji jest następująca:



1) Rozmiar obszaru argumentów zwykle jest stały dla danej procedury, ale w językach zezwalających na procedury ze zmienną liczbą argumentów lub zmiennym rozmiarem pojedynczego argumentu (np. tablica dynamicznego rozmiaru) rozmiar obszaru argumentów może być różny dla różnych wywołań tej samej procedury (nie będziemy się takimi przypadkami zajmować).

2) Obszar informacji organizacyjnych zwykle ma identyczną strukturę dla wszystkich procedur (i wszystkich wywołań), szczegóły zależą od konkretnego języka programowania (i implementacji kompilatora), ale zawsze elementem informacji organizacyjnych jest adres powrotu procedury (adres, do którego sterowanie ma przejść po zakończeniu wykonywania procedury - adres w kodzie procedury wywołującej) oraz adres rekordu aktywacji procedury wywołującej.

3) Rozmiar obszaru zmiennych lokalnych, podobnie jak rozmiar obszaru argumentów, zwykle jest stały dla danej procedury, ale w językach zezwalających na procedury z tablicami lokalnymi dynamicznego rozmiaru może być różny dla różnych wywołań tej samej procedury (nie będziemy się takimi przypadkami zajmować).

4) Różnica pomiędzy zmiennymi lokalnymi a zmiennymi roboczymi polega na tym, że zmienne lokalne to zmienne widoczne w kodzie programu (zadeklarowane przez programistę), a zmienne robocze to zmienne wygenerowane przez kompilator, zwykle do pamiętania pośrednich wyników obliczeń. Obszar zmiennych roboczych może się dynamicznie zmieniać podczas wykonania programu - nie będziemy się nim zajmować.

Mechanizmy przekazywania parametrów

Mechanizmy przekazywania parametrów to kolejna cecha języka źródłowego mająca wpływ na fazę generowania kodu.
Przede wszystkim zwróćmy uwagę, że pojęcie to dotyczy sposobów przekazywania pojedynczego parametru. Kwestia kolejności obliczania parametrów to zupełnie co innego.

Z punktu widzenia twórcy kompilatora parametry procedury są lokalizacjami w jej rekordzie aktywacji, a mechanizmy przekazywania parametrów określają, w jaki sposób z tych lokalizacji korzystać.

Dla użytkownika języka ważniejsze jest oczywiście, w jaki sposób różne mechanizmy przekazywania parametrów wpływają na przepływ informacji pomiędzy procedurą wywołującą a wywoływaną i ewentualnie z powrotem.

Omówimy kolejno najpopularniejsze mechanizmy przekazywania parametrów.

1) Przekazywanie przez wartość (pass by value)
W tej metodzie parametry są wyrażeniami obliczanymi podczas wywołania procedury (przed przekazaniem sterowania do procedury wywoływanej). W większości języków wewnątrz procedury traktowane są tak samo jak zmienne lokalne. Z punktu użytkownika ważne jest, że po powrocie procedury wartości parametrów są takie same jak w chwili wywołania. Jest tak dlatego, że procedura operuje na kopiach parametrów we własnym rekordzie aktywacji, a nie na oryginalnych zmiennych. Jest to najpopularniejszy sposób przekazywania parametrów (w wielu językach jedyny).

2) Przekazywanie przez zmienną (pass by reference).
Nazywane również przekazywaniem przez referencję. W tej metodzie parametr aktualny musi być obiektem mającym lokalizację w pamięci (zwykle zmienną lub elementem tablicy). Właśnie ta lokalizacja (adres) umieszczana jest w rekordzie aktywacji jako parametr. Wewnątrz procedury dodany jest niejawnie dodatkowy poziom pośredniości, a mianowicie przy korzystaniu z parametru następuje dostęp do pamięci wskazywanej przez przekazany jako parametr adres. W konsekwencji wszelkie operacje dotyczą oryginalnej zmiennej i po wyjściu z procedury wszystkie zmiany wartości zmiennej pozostają w mocy (i to właśnie jest najważniejsza cecha tego sposobu przekazywania parametrów z punktu widzenia użytkownika języka).
Uwaga: Czasem wybieramy ten sposób przekazywania parametrów nie dlatego, że chcemy zachować zmiany dokonane wewnątrz procedury, a dlatego, że chcemy uniknąć czasochłonnego kopiowania dużych obiektów (co byłoby konieczne przy przekazywaniu przez wartość).

3) Przekazywanie przez wartość-wynik (pass by value-result)
Z punktu widzenia użytkownika języka jest podobna do przekazywania przez zmienną (zmiany dokonane przez procedurę wywoływaną pozostają po powrocie z niej), ale wewnętrzna implementacja jest zupełnie inna.
Mechanizm jest następujący
- Przy wywołaniu parametr jest obliczany analogicznie jak przy przekazywaniu przez wartość, jednak dodatkowo do procedury przekazywany jest również adres zmiennej (jak przy przekazywaniu przez zmienną).

- Wewnątrz procedury sposób korzystania z parametru jest identyczny jak przy przekazywaniu przez wartość.
- Przy powrocie z procedury wartość parametru (być może zmieniona wewnątrz procedury) jest kopiowana do zmiennej będącej parametrem aktualnym (wykorzystywany jest przekazany przy wywołaniu adres parametru).

Metoda ta daje inne wyniki niż przekazywanie przez zmienną jedynie w przypadku tak zwanego aliasingu, czyli sytuacji, gdy do tej samej komórki pamięci można dostać się na kilka sposobów (np. za pomocą dwóch parametrów, albo przez parametr i zmienną nielokalną).

Przekazywanie parametrów przez wartość-wynik jest zdecydowanie rzadziej stosowanym mechanizmem (był stosowany w języku Ada).

4) Przekazywanie przez nazwę (pass by name).
To najbardziej skomplikowany sposób przekazywania parametrów. Po raz pierwszy był zastosowany już w Algolu 60, jednak się nie upowszechnił. Idea polega na tym, że parametr jest obliczany przy każdym dostępie do niego. Widać, że tak naprawdę to nie jest przekazywana ani wartość parametru, ani jego adres, a funkcja pozwalająca na jego obliczenie (a ściślej na obliczenie jego adresu). Oczywiście najciekawsze efekty można uzyskać wtedy, gdy przy każdym dostępie do parametru uzyskujemy jego różną wartość (możliwa jest również sytuacja, że różne dostępy to tak naprawdę dostępy do różnych komórek pamięci).
Ważną odmianą tego mechanizmu przekazywania parametrów jest tak zwane leniwe obliczanie parametrów (lazy evaluation). Polega ona na tym, że wartość parametru obliczana jest przy pierwszym dostępie do niego i zapamiętywana, przy kolejnych dostępach nie ma już ponownego obliczania wartości parametru, a korzysta się z owej zapamiętanej wartości (można ją zmodyfikować). Taki mechanizm stosowany jest w językach funkcyjnych.

System typów

System typów to kolejna ważna cecha języków programowania mająca wpływ nie tylko na fazę analizy, ale również na generowanie kodu.

Pod względem systemu typów języki programowania można podzielić na dwie główne grupy:
- języki ze statycznym systemem typów
- języki z dynamicznym systemem typów

Języki ze statycznym systemem typów

To języki, w których wymagane jest określenie typu każdej ze zmiennych w chwili deklaracji tej zmiennej. Określenie to jest zwykle jawne, ale może też być niejawne.
Przykłady niejawnego określenia typu to:
- deklaracja var w języku C# (typ zmiennej wynika z postaci inicjalizatora tej zmiennej)
- konwencja w języku Fortran, że nie zadeklarowane zmienne o nazwach rozpoczynających się od liter: I, J, K, L, M, N są typu integer, a nie zadeklarowane zmienne o nazwach rozpoczynających się od innych liter są typu double.

W takich językach typ zmiennej jest niezmienny przez cały czas wykonania programu i może być określony podczas kompilacji. Co więcej, typ dowolnego elementu każdego z wyrażeń (czyli każdego podwyrażenia) również może być określony podczas kompilacji.

W konsekwencji wszelkie błędy związane nieprawidłowymi operacjami mającymi coś wspólnego z typami mogą być wykryte podczas kompilacji.

Takie języki określane są również jako języki silnie utypowione (strongly typed).

Języki z dynamicznym systemem typów

To języki, w których w deklaracjach zmiennych nie ma obowiązku umieszczania informacji o typie tych zmiennych, a nawet w ogóle nie jest wymagane deklarowanie zmiennych. Typ zmiennych określa się podczas wykonania na podstawie wartości przypisanej zmiennej. Zdarza się, że po pierwszym przypisaniu typ jest już ustalony, ale może też być tak, że zmienia się on wielokrotnie podczas wykonania programu (jeśli do zmiennej będą w różnych chwilach przypisane wartości różnych typów). W takich językach cała kontrola typów i błędów z nimi związanych przeniesiona jest na czas wykonania programu, czego skutkiem jest:

- opóźnione wykrywanie błędów związanych z typami
- znaczące spowolnienie wykonania programu

w porównaniu z językami ze statycznym systemem typów.
Zwolennicy takich języków twierdzą, że zaletą jest mniejszy formalizm i większa wygoda programowania. Moim zdaniem, zalety te w najmniejszym stopniu nie rekompensują strat związanych z opóźnionym wykrywaniem błędów i mniejszą wydajnością (oczywiście nie licząc króciutkich skrypcików, w których błędy widać od razu).

Takie języki określane są również jako języki słabo utypowione (weakly typed).

Uwaga 1:
Możliwość konwersji (jawnych i niejawnych) pomiędzy typami nie ma związku z powyższym podziałem. Zdarza się, że język C jest określany jako język ze słabym systemem typów ponieważ umożliwia wiele niejawnych konwersji.
To nieprawda - język C ma silny statyczny system typów, ponieważ typ każdej zmiennej i każdego wyrażenia da się określić podczas kompilacji.

Uwaga 2:
Dziedziczenie i funkcje wirtualne nie powodują osłabienia systemu typów. Wprawdzie zmiennej typu bazowego można użyć jako wskaźnika/referencji do obiektu klasy pochodnej, ale wszelkie kontrole poprawności nadal można przeprowadzić podczas kompilacji.

Uwaga 3:
Deklaracja zmiennej z użyciem słowa kluczowego dynamic w języku C# rzeczywiście tworzy zmienną dynamicznie typowaną (jej typ i dozwolone na niej operacje określone są dopiero podczas wykonania programu).

Od teraz będziemy zajmować się jedynie językami ze statycznym systemem typów.

Warto zauważyć jeszcze jedną korzystną cechę języków ze statycznym systemem typów - jedynie w takich językach można podczas kompilacji określić rozmiar pamięci niezbędnej do przechowywania zmiennej, a to bardzo ułatwia szybki dostęp do zmiennej. W językach z dynamicznym systemem typów alokacja zmiennych również musi być dynamiczna.

Podział typów

Typy można podzielić na:
- typy proste
- typy złożone

Typy proste nie mają wewnętrznej struktury, dostęp możliwy jest jedynie do zmiennej jako całości (a nie do jej fragmentów).
Można wyróżnić
- typy proste predefiniowane
- typy proste definiowane przez użytkownika

Przykłady typów predefiniowanych (standardowych) to typy liczbowe (integer, float, double), typ znakowy (char), typ logiczny (bool) oraz (to ciekawy przykład) typ void.

Przykłady typów prostych definiowanych przez użytkownika to
- typy wyliczeniowe
- typy okrojone (np. znane z Pascala 1..10)
- wskaźniki (tak, wskaźniki to typy proste!)

Typy złożone mają wewnętrzną strukturę, możliwy jest dostęp do fragmentów obiektu.
Można wyróżnić
- tablice
- rekordy (struktury/klasy)
- unie
- inne typy złożone

Konkretne typy złożone są zwykle definiowane przez użytkownika (albo przez biblioteki dodane do języka), ale zdarzają się też złożone typy predefiniowane wbudowane w rdzeń języka (np. typ string). Mechanizmy definiowania typów złożonych są oczywiście elementem rdzenia języka.

A) Tablice - na naszym poziomie zaawansowania w programowaniu nie ma potrzeby tłumaczyć idei pojęcia tablicy.

Warto jednak zwrócić uwagę na dwa możliwe podejścia do składni definiowania tablic, które mają głębsze konsekwencje.

1) definicje w rodzaju: int tab[10,20,30];

w tym przypadku zwykle można odwoływać się:
- do całej tablicy (w tym przypadku 3-wymiarowej) tab
- do pojedynczego elementu tab[i,j,k]

natomiast nie można się odwoływać do mniej wymiarowych fragmentów tablicy

2) definicje w rodzaju: int tab[10][20][30];

w tym przypadku zwykle można odwoływać się:

- do całej tablicy (3-wymiarowej) tab
- do podtablicy 2-wymiarowej tab[i]
- do podtablicy 1-wymiarowej tab[i][j]
- do pojedynczego elementu tab[i][j][k]

B) Struktura to obiekt, którego składowe mogą być różnych typów (w odróżnieniu od elementów tablicy) a dostęp do składowych uzyskujemy za pomocą ich nazw (a nie indeksów, jak w przypadku elementów tablicy). Do rekordów/struktur wrócimy jeszcze przy okazji typów rekurencyjnych.

C) Unie to struktury z nakładającymi się polami (szczegóły pomijamy, ponieważ nie wnoszą nic ciekawego).

D) Inne typy - niektóre języki definiują inne specyficzne dla siebie typy (np. zbiory w Pascalu) - szczegóły pomijamy.

Dalej zakładamy, że język umożliwia definiowanie typów przez użytkownika (programistę) i oczywiście nadawanie typom nazw (szczegóły składniowe, jak się to robi, są nieistotne).

Typy rekurencyjne

Typ rekurencyjny to typ, którego obiekt zawiera jako swoją składową obiekt tego samego typu.

Niektóre języki udają, że jest to możliwe wprost.
Np. w C# możliwa jest definicja klasy

```
class BST
        {
        int     val;
        BST     left;
        BST     right;
        }
```

To oczywiście rodzaj manipulacji ponieważ składowymi klasy BST nie są obiekty tej klasy, tylko referencje do nich (zresztą ogólnie pojęcie referencji ma w sobie coś z manipulacji).

Inne języki są bardziej uczciwe i wymagają użycia wskaźników, np. C++

```
class BST
        {
        int     val;
        BST     *left;
        BST     *right;
        }
```

Równoważność typów

Określenie równoważności typów jest bardzo ważnym elementem języka programowania i kompilatora. Jest niezbędne np. dla określenia poprawności operacji przypisania i skojarzenia parametru aktualnego i formalnego przy wywołaniu procedury.

Równoważność typów można definiować na różne sposoby.

A) Równoważność nazw - to bardzo restrykcyjna definicja równoważności typów.

Def. Dwa typy są równoważne według nazw wtedy i tylko wtedy gdy typy te mają tą samą nazwę.

Przykład 1
Załóżmy że mamy zdefiniowane typy
type t1 = integer;
type t2 = integer;

Typy t1 i t2 nie są równoważne według równoważności nazw, ponadto żaden z nich nie jest równoważny typowi integer
dla deklaracji:
x : integer;
y : t1;
z : t2;
każda ze zmiennych x, y, z jest innego typu.

Jeśli język dopuszcza w deklaracjach zmiennych użycie opisów typów, sytuacja komplikuje się jeszcze bardziej, ponieważ każdy z opisów typów generuje niejawną definicję typu o unikalnej nazwie.

Przykład 2

type tab = array [1..10] of integer;

a: tab;
b : array [1..10] of integer;
c, d : array [1..10] of integer;
e : tab;

Każda ze zmiennych a, b, c jest innego typu, natomiast zmienne c i d są tego samego (nienazwanego jawnie) typu, a zmienne a i e są tego samego typu tab.

B) Nieco swobodniejszą definicją równoważności typów jest równoważność deklaracji.
W tej definicji równoważności przyjmuje się że definicje typów w postaci

type t2 = t1;

w rzeczywistości nie definiują nowego typu t2, a nową nazwę (alias) dla typu t1.
W konsekwencji przy takiej definicji wszystkie zmienne z Przykładu 1 są tego samego typu (do którego można się odwoływać za pomocą nazw integer, t1, t2).

C) Zupełnie inna jest idea strukturalnej równoważności typów

Def. Dwa typy są równoważne strukturalnie wtedy i tylko wtedy, gdy są tym samym typem prostym lub ich kolejne składowe są równoważne strukturalnie.

Ideą tej definicji jest zasada, że najważniejsza jest reprezentacja zmiennej w pamięci i jeśli ta reprezentacja jest jednakowa, to typy są równoważne.
Zakładamy przy tym, że składowe są umieszczone w pamięci bez przerw i w takiej kolejności jak kolejność składowych w deklaracji typu (ważna jest kolejność składowych, a nie ich nazwy!).

Przykład 3 (prosty)

```
type t1 = record
        x: integer;
        y : real;
        z : array [1..10] of integer
end;
type t2 = record
        i: integer;
        r : real;
        t : array [1..10] of integer
end;
```

Typy t1 i t2 są równoważne strukturalnie

Przykład 4 (ciekawszy)

```
type t1 = record
        x : real;
        y : pointer to t1
end;
type t2 = record
        val : real;
        next : pointer to t2
end;
```

Definicja równoważności strukturalnej jest rekurencyjna, zatem proces badania równoważności również przebiega rekurencyjnie

- czy typy t1 i t2 są równoważne ?
  nie są to typy proste, więc badamy składowe
  - czy pierwsze składowe są równoważne? - TAK
  - czy drugie składowe są równoważne?
    są to wskaźniki, a wskaźniki są równoważne, gdy wskazują na typy równoważne
    czy typy t1 i t2 są równoważne?

I jest problem: podczas badania równoważności typów t1 i t2 pojawiło się pytanie o równoważność typów t1 i t2 - rozwiązaniem jest **założenie równoważności**.

Założenie równoważności mówi, że jeśli podczas badania równoważności dwóch typów pojawi się ponownie pytanie o równoważność tej samej pary typów, to za drugim razem odpowiadamy, że są one równoważne.

W bardziej skomplikowanych przypadkach może być konieczne stworzenie całej listy "otwartych pytań", ale założenie równoważności nadal ma zastosowanie.

3. Modele środowiska wykonawczego

Można wyróżnić trzy główne modele środowiska wykonawczego (ang. runtime environment).
1) Model statyczny
2) Model stosowy, a w nim podmodele
    a) dla języków bez procedur lokalnych (zagnieżdżonych)
    b) dla języków z procedurami lokalnymi (zagnieżdżonymi)
    c) dla języków z procedurami jako parametrami (i procedurami zagnieżdżonymi)
3) Model dynamiczny

Modele te zostaną poniżej omówione bardziej szczegółowo.

Środowisko w pełni statyczne

W pełni statyczne środowisko wykonawcze jest odpowiednie dla języków posiadających następujące cechy

- brak możliwości dynamicznej alokacji pamięci (czyli brak operatorów new i delete oraz funkcji malloc, free i podobnych)
- brak jawnych wskaźników
- brak możliwości rekurencyjnych wywołań procedur
- brak procedur lokalnych (zadeklarowanych wewnątrz innych procedur)

Jak widać ograniczenia są bardzo poważne i w konsekwencji jedynym powszechnie znanym językiem programowania spełniającym te ograniczenia jest (był) język FORTRAN (w swojej pierwotnej wersji z lat 60-tych XX wieku).

W modelu statycznym wszystkie zmienne (nie tylko zmienne globalne) są alokowane statycznie, czyli mają stałe adresy. Jest to możliwe ponieważ przy powyższych ograniczeniach można statycznie zaalokować rekordy aktywacji wszystkich procedur (po jednym na każdą procedurę).
Ze względu na wymienione powyżej cechy model statyczny jest najłatwiejszy do implementacji (nie wymaga zarządzania rekordami aktywacji podczas wykonania programu).

Ciekawostka: W nowszych implementacjach FORTRAN-u funkcje rekurencyjne nie były formalnie zakazane (nie powodowały błędu kompilacji ani wykonania), ale wszystkie instancje tej samej procedury korzystały z jednego rekordu aktywacji, czyli zmienne lokalne były "wspólne" dla wszystkich wywołań tej samej funkcji (analogiczny efekt można osiągnąć w języku C deklarując zmienną lokalną ze słowem static).

Środowisko stosowe

Stosowy model środowiska wykonawczego (ang. stack-based runtime environment) jest niezbędny dla języków umożliwiających rekurencyjne wywoływanie procedur. W modelu stosowym każdemu wywołaniu procedury odpowiada odrębny rekord aktywacji, zatem jednej procedurze może odpowiadać wiele rekordów aktywacji, dzięki czemu każde z wywołań może mieć odrębny zestaw zmiennych lokalnych.

Niestety możliwość tworzenia wielu rekordów aktywacji dla jednej procedury oznacza, że nie mogą one być tworzone statycznie podczas kompilacji, a muszą być tworzone dynamicznie

podczas wykonania programu (w chwili wywołania procedury) i dynamicznie likwidowane (w chwili powrotu z procedury).

Najlepszym (i najprostszym) sposobem organizacji rekordów aktywacji spełniającym powyższe wymagania jest **stos rekordów aktywacji** (stąd nazwa modelu środowiska wykonawczego).

Konsekwencją dynamicznego tworzenia rekordów aktywacji jest to, że zmienne lokalne nie mają już stałych adresów. W modelu stosowym do zaadresowania zmiennej lokalnej potrzebne są dwa elementy

- położenie zmiennej w rekordzie aktywacji (to jest stałe)
- adres rekordu aktywacji (to jest zmienne) - ściślej nie musi to być (i zwykle nie jest) adres początku rekordu aktywacji, a adres jakiegoś ustalonego punktu w rekordzie aktywacji

Taki sposób adresowania zmiennych lokalnych powoduje kolejne konsekwencje.
Po pierwsze, konieczność pamiętania, gdzieś poza rekordami aktywacji, adresu bieżącego rekordu aktywacji (a ściślej owego ustalonego punktu w bieżącym rekordzie aktywacji). Często jest to specjalny rejestr procesora określany jako frame pointer (FP).
Po drugie, w rekordzie aktywacji, w obszarze informacji organizacyjnych, musi być oprócz adresu powrotu z procedury zapamiętany również adres rekordu aktywacji procedury wywołującej (obie te informacje są niezbędne do odtworzenia stanu środowiska po powrocie z procedury). Innymi słowy, w rekordzie aktywacji musi być zapamiętany poprzedni frame pointer, a pole w rekordzie aktywacji, które go przechowuje to **control link**, nazywany również linkiem dynamicznym. W rzeczywistości zwykle właśnie to pole, a nie początek rekordu aktywacji, jest miejscem, na które wskazuje frame pointer.

Konieczność dynamicznego tworzenia/likwidowania rekordów aktywacji sprawia, że wywołanie/powrót z procedury to nie jest już pojedyncza operacja, a cała sekwencja działań.

Typowa **sekwencja wywołania** jest następująca:
1) obliczenie i umieszczenie na stosie argumentów
2) zapamiętanie (na stosie) adresu powrotu i skok
3) zapamiętanie (na stosie) poprzedniej wartości frame pointer jako control link
4) obliczenie nowej wartości frame pointer (często jest to po prostu adres szczytu stosu)
5) rezerwacja (na stosie) pamięci na zmienne lokalne

Zauważmy, że w ten sposób rekord aktywacji tworzony jest na stosie "po kawałku" (zwykle w procesorze nie ma rozkazu "utwórz rekord aktywacji", a taka operacja często jest dostępna w kodzie pośrednim).

Typowa **sekwencja powrotu** jest następująca:
1) likwidacja zmiennych lokalnych
2) odtworzenie poprzedniej wartości frame pointer (na podstawie control link)
3) powrót skokiem
4) usunięcie ze stosu argumentów

W procesorach nie posiadających sprzętowego wspomagania obsługi stosu (w przeszłości były takie!) wspomniane sekwencje mogą wyglądać nieco inaczej.

Stosowy model środowiska wykonawczego jest obecnie najpopularniejszy, jest odpowiedni dla szerokiej klasy języków, jednak różnice w owych językach sprawiają, że powstało kilka wariantów tego modelu. Omówione zostaną trzy najważniejsze warianty.

Model stosowy dla języków bez procedur lokalnych

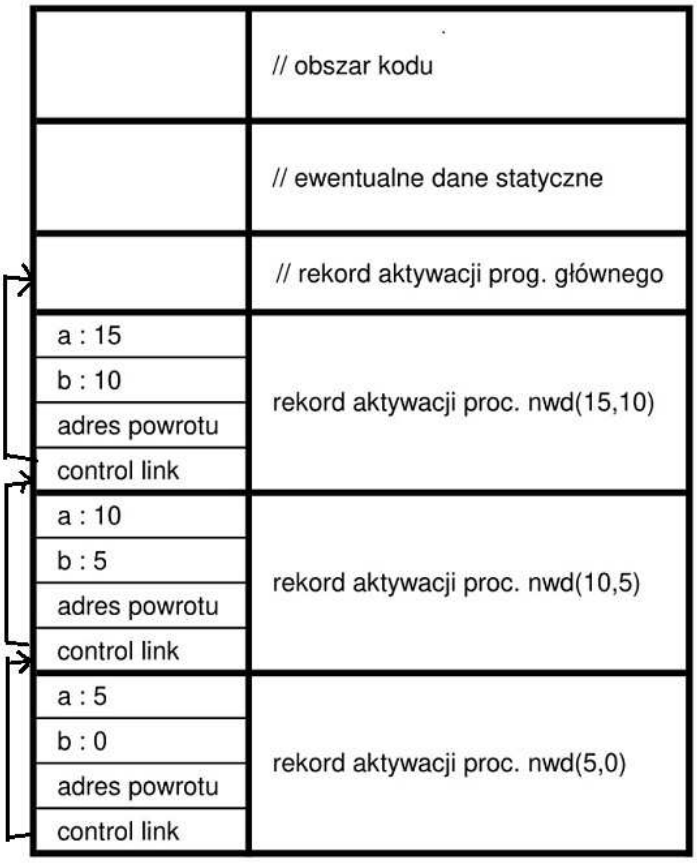
To najprostszy wariant modelu stosowego. Uproszczenie wynika z faktu, że dla takich języków zmienne mogą być albo globalne (alokowane statycznie jak w modelu statycznym i posiadające stałe adresy) albo całkowicie lokalne (alokowane w rekordzie aktywacji bieżącej procedury i dostępne za pośrednictwem rejestru frame pointer). Przykładem języka, dla którego odpowiedni jest ten wariant modelu stosowego jest język C.

Rozważmy prostą funkcję obliczającą rekurencyjnie największy wspólny dzielnik zadanych dodatnich liczb całkowitych oraz jej wywołanie.
W języku C implementacja może być następująca.

```
int nwd(int a, int b)
      {
      return b ? nwd(b,a%b) : b ;
      }

main()
      {
      nwd(15,10)
      }
```

Podczas wykonania programu utworzone zostaną następujące rekordy aktywacji



Uwaga: Z przedstawionego rysunku nie należy wyciągać wniosku, że control link jest zawsze ostatnim elementem rekordu aktywacji - w tym prostym przykładzie jest tak wyłącznie dlatego, że w procedurze nwd nie ma w ogóle zmiennych lokalnych, zatem rekord aktywacji zawiera jedynie parametry i informacje organizacyjne.

Dotychczas milcząco zakładaliśmy, że każda z funkcji ma stałą liczbę parametrów oraz, że rozmiary pojedynczych parametrów są stałe.
Jednak model stosowy (nawet w tym najprostszym wariancie) jest odpowiedni zarówno dla funkcji ze zmienną liczbą parametrów jak i dla języków dopuszczających zmienny rozmiar pojedynczego parametru).

Rozwiązaniem problemu zmiennej liczby parametrów jest przetwarzanie i umieszczenie w rekordzie aktywacji parametrów aktualnych w odwrotnej kolejności (to znaczy w kolejności począwszy od prawego do lewego). Języki dopuszczające zmienną liczbę parametrów zwykle umożliwiają pomijanie jedynie parametrów końcowych, zatem takie przetwarzanie "od końca" sprawia, że rzeczywiście istniejące parametry zawsze będą miały stałe położenie względem obszaru informacji organizacyjnych rekordu aktywacji (i to wystarczy do ich prawidłowej obsługi).

Problem zmiennego rozmiaru pojedynczego parametru jest nieco bardziej skomplikowany i jego rozwiązanie wymaga wprowadzenia dodatkowego poziomu pośredniości. Pomysł polega na tym, że w rekordzie aktywacji oprócz samego parametru (który w związku ze zmiennym rozmiarem nie może mieć stałego położenia w rekordzie aktywacji) dodatkowo umieszczany jest wskaźnik do tego parametru (i ten wskaźnik ma już stałe położenie w rekordzie aktywacji).

Uwaga 1: Problem zmiennego rozmiaru pojedynczego parametru powstaje, gdy język umożliwia przekazywanie przez wartość całych tablic różnych rozmiarów, co wiąże się z kopiowaniem do rekordu aktywacji całej tablicy (czyli nie tak jak w C/C# gdzie przekazywany jest jedynie adres/referencja początku tablicy).

Uwaga 2: Podobnie jak problem zmiennego rozmiaru pojedynczego parametru można rozwiązać problem lokalnych tablic zmiennego rozmiaru.

Model stosowy dla języków z procedurami lokalnymi

W tym modelu oprócz zmiennych lokalnych (znajdujących się w rekordzie aktywacji aktualnie wykonywanej procedury) i zmiennych globalnych (zadeklarowanych statycznie i mających stałe adresy) pojawiają się jeszcze zmienne nielokalne zadeklarowane w procedurach zewnętrznych względem procedury aktualnie wykonywanej. Takie zmienne znajdują się oczywiście w rekordzie aktywacji procedury, w której zostały zadeklarowane i aby uzyskać do nich dostęp trzeba sięgnąć w głąb stosu rekordów aktywacji.

Omówiony poprzednio control link umożliwia jedynie dostęp do rekordu aktywacji procedury wywołującej (niezależnie od tekstowej struktury zagnieżdżeń procedur), zatem może być wykorzystany jedynie do implementacji dynamicznych zasięgów zmiennych (a jak było wspomniane wcześniej w tym opracowaniu będą omawiane jedynie zasięgi statyczne).

Aby zaimplementować statyczne zasięgi zmiennych konieczne jest wprowadzenie kolejnej informacji zapamiętywanej w obszarze informacji organizacyjnych. Ta dodatkowa informacja

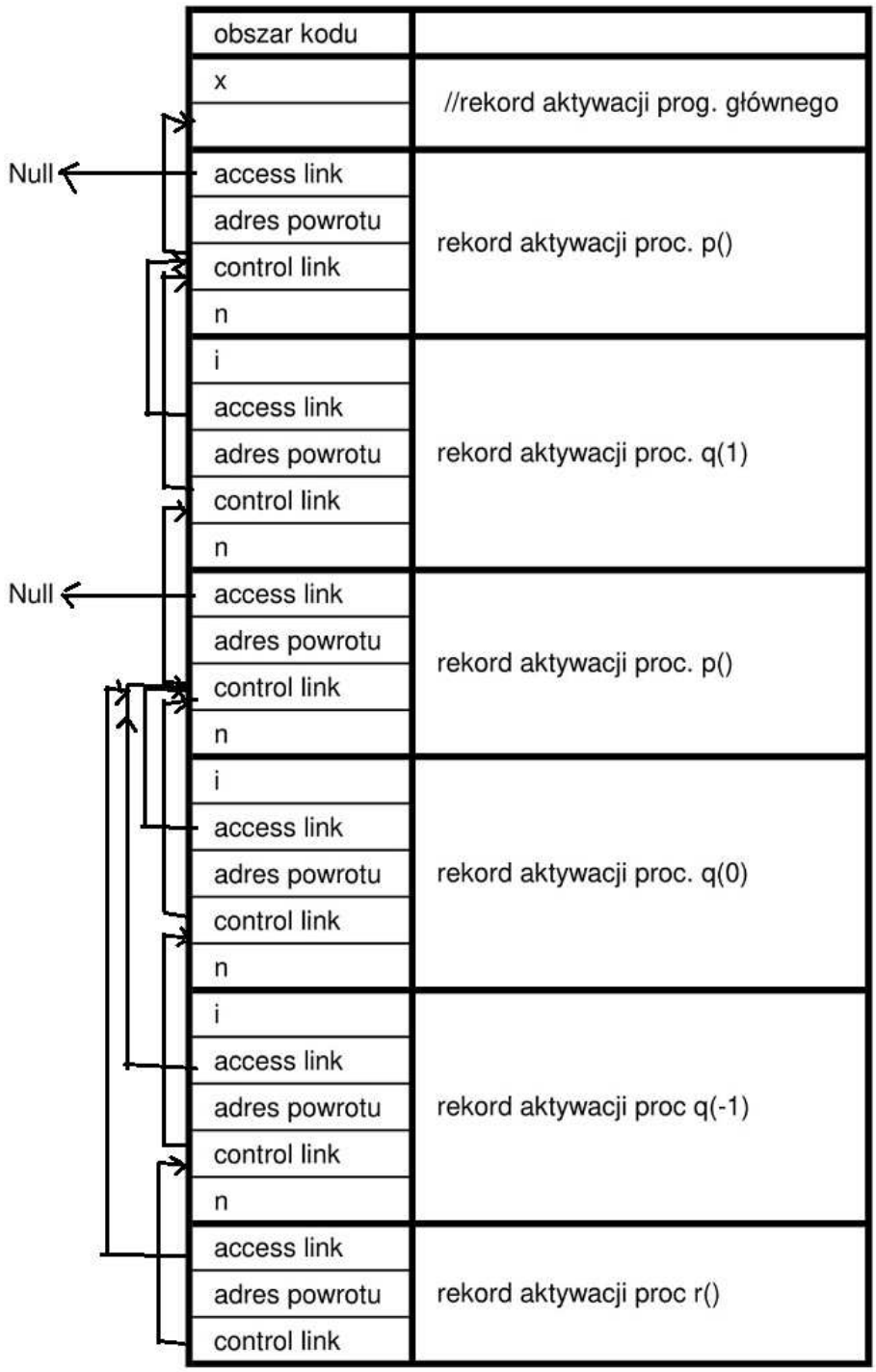
to **access link** (nazywany również linkiem statycznym), który wskazuje na tak zwane **środowisko definiujące**, czyli na rekord aktywacji procedury wewnątrz, której procedura wywoływana jest zadeklarowana. Jak łatwo zauważyć tak określony access link umożliwia implementację zasięgów statycznych. Łatwo również zauważyć, że access link i control link są sobie równe jedynie jeśli wywołanie procedury następuje z procedury wewnątrz, której jest deklaracja procedury wołanej (skądinąd to całkiem częsty przypadek).

Przykład (zagmatwany)

Rozważmy program

```
program
   {
   int x;
   proc p()
      {
      int n;
      proc q(int i)
         {
         int n;
         x = i-1;
         if ( i>0 )
            p();
         else
            if ( i<0 )
               r();
            else
               q(i-1);
         }
      proc r()
         {
         n = 2;
         }
      q(x);
      }
   x = 1;
   p();
   }
```

W chwili wywołania procedury r stos rekordów aktywacji wygląda tak jak na przedstawionym dalej rysunku



Uwaga 1: W przypadku wywołań rekurencyjnych access link procedury zagnieżdżonej wskazuje na "dynamicznie najbliższy" rekord aktywacji procedury statycznie zewnętrznej. Zauważmy, że pozostaje to w całkowitej zgodzie ze statycznymi zasięgami zmiennych (zmienne z kolejnych wywołań rekurencyjnych przesłaniają się wzajemnie i widoczna jest ta z wywołania "dynamicznie najbliższego").

Uwaga 2: Procedury globalne mają "puste" (czyli równe null) access linki. Można również przyjąć, że access linki procedur globalnych wskazują na rekord aktywacji programu głównego (jest to szczególnie wygodne, gdy z powodów implementacyjnych zdecydujemy się traktować program główny tak samo jak inne procedury).

Aby uzyskać dostęp do zmiennych nielokalnych należy najpierw za pomocą access linku wyznaczyć rekord aktywacji procedury, wewnątrz której dana zmienna jest zadeklarowana, a następnie, znając położenie tej zmiennej w rekordzie aktywacji, uzyskać dostęp do samej zmiennej (analogicznie jak do zmiennych lokalnych).

Większość języków umożliwiających zagnieżdżanie procedur umożliwia również zagnieżdżanie wielopoziomowe, co może powodować potrzebę dostępu do zmiennych zadeklarowanych w procedurze wiele poziomów wyżej. Nie powoduje to jednak żadnych problemów, w takim przypadku należy po prostu wielokrotnie "przejść po access linku" (dla każdej procedury można podczas analizy składniowej obliczyć (i umieścić w tablicy symboli) atrybut "poziom zagnieżdżenia", dzięki któremu wiadomo będzie ile razy należy "przejść po access linku").

Uwaga: Możliwe jest również utworzenie i pamiętanie w rekordzie aktywacji "tablicy access linków" (nazywanej zwykle tablicą DISPLAY), której rozmiar odpowiada poziomowi tekstowego zagnieżdżenia danej procedury. Kolejne jej elementy odpowiadają bezpośredniemu przejściu odpowiednio 1, 2, 3, ... poziomy wyżej. Przy pomocy tablicy DISPLAY można uzyskać dostęp do zmiennych zadeklarowanych wiele poziomów wyżej "w jednym kroku", bez konieczności wielokrotnego przechodzenia po pojedynczych access linkach. Takie rozwiązanie może przyspieszyć działanie programu, ale jest bardziej skomplikowane i nie będzie dalej omawiane.

W wariancie stosowego modelu środowiska wykonawczego z procedurami zagnieżdżonymi konieczna jest niewielka modyfikacja opisanych wcześniej sekwencji wywołania i powrotu (zmiana dotyczy oczywiście obsługi access linków).

Sekwencja wywołania jest w tym przypadku następująca:
1) obliczenie i umieszczenie na stosie argumentów
2) obliczenie i umieszczenie na stosie access linku
3) zapamiętanie (na stosie) adresu powrotu i skok
4) zapamiętanie (na stosie) poprzedniej wartości frame pointer jako control link
5) obliczenie nowej wartości frame pointer (często jest to po prostu adres szczytu stosu)
6) rezerwacja (na stosie) pamięci na zmienne lokalne

Natomiast sekwencja powrotu jest następująca:
1) likwidacja zmiennych lokalnych
2) odtworzenie poprzedniej wartości frame pointer (na podstawie control link)
3) powrót skokiem
4) usunięcie ze stosu access linku i argumentów

Należy jeszcze wyjaśnić w jaki sposób można obliczyć access link (bo w odróżnieniu od control linku, który jest po prostu bieżącą wartością rejestru frame pointer, access link trzeba obliczać).

Załóżmy, że procedura p o poziomie zagnieżdżenia zp wywołuje procedurę q o poziomie zagnieżdżenia zq (przypominam, że poziomy zagnieżdżenia są znane podczas kompilacji).

Może wystąpić jeden z dwóch przypadków

1) Przypadek, gdy zq > zp
Jedyna możliwość to zq = zp +1, a procedura q jest zadeklarowana bezpośrednio wewnątrz procedury p. W takim przypadku access link = control link (zasięgi statyczne i dynamiczne są równoważne).

2) Przypadek, gdy zq <= zp
W tym przypadku należy cofnąć się po łańcuchu access linków (poczynając od rekordu aktywacji procedury p) zp-zq+1 razy, a rekord aktywacji, który w ten sposób odnajdziemy będzie tym, na który ma pokazywać właśnie obliczany access link. W szczególności w ważnym praktycznie przypadku gdy zq=zp (co oznacza, że obie procedury p i q są zadeklarowane wewnątrz tej samej procedury) otrzymujemy, że nowo obliczany access link (dla procedury q) jest równy access linkowi z bieżącego rekordu aktywacji, czyli
nowy access link = poprzedni access link.

Uwaga: Opisany powyżej proces pokazuje, że access link należy obliczyć jeszcze przed wąsko rozumianym wywołaniem (czyli przed krokiem 3 przedstawionej wyżej sekwencji wywołania).

Model stosowy dla języków z procedurami jako parametrami

W przypadku języków umożliwiających przekazywanie procedur jako parametrów, ale nie umożliwiających definiowania procedur lokalnych sprawa jest prosta - wystarczy aby parametrem odpowiadającym procedurze był po prostu wskaźnik do kodu procedury. Tak jest np. w językach C/C++.

Problem pojawia się dla języków z procedurami lokalnymi, które jednocześnie umożliwiają przekazywanie procedur jako parametrów. W tym przypadku zwyczajny wskaźnik do kodu procedury nie wystarczy jako parametr, ponieważ w chwili wywołania procedury przekazanej jako parametr potrzebny jest również access link, a okazuje się że **obliczenie access linku w chwili wywołania procedury przekazanej jako parametr może być niemożliwe**.
Natomiast obliczenie przyszłego access linku zawsze jest możliwe w chwili, gdy procedura po raz pierwszy jest przekazywana jako parametr (robi się to w dokładnie taki sam sposób jak przy bezpośrednim wywołaniu procedury).
Rozwiązanie problemu jest takie, że parametr proceduralny jest w rzeczywistości parą <wskaźnik do kodu procedury, przyszły access link>, taka para nazywana jest domknięciem (ang. closure) procedury.

Organizacja rekordu aktywacji w modelu w rozważanym aktualnie wariancie modelu stosowego jest taka sama jak w wariancie poprzednim (jedyną różnicą jest sposób reprezentacji parametrów proceduralnych).

Poniżej podany jest prosty przykład pokazujący niemożność obliczenia access linku w chwili wywołania procedury przekazanej jako parametr.

```
program
   {
   procedure p(procedure a())
      {
      a();
      }
   procedure q()
      {
      int x;
      procedure r()
         {
         write(x);
         }
      x = 2;
      p(r);
      }
   q();     // wnętrze programu głównego
   }
```

Zauważmy, że procedura r wywołana jest z wnętrza procedury p (a wcześniej przekazana do procedury p jako parametr). Problem polega na tym, że procedura p nic nie wie o procedurze r (bezpośrednie wywołanie procedury r z wnętrza procedury p jest niemożliwe), a zatem nie jest możliwe obliczenie access linku w chwili rzeczywistego wywołania procedury r. Natomiast procedura q może obliczyć niezbędny w przyszłości access link w chwili przekazywania procedury r jako parametru.

Uwagi:

1) Procedury przekazane jako parametry muszą być wywoływane inaczej (na poziomie kodu maszynowego) niż "zwykłe procedury" - muszą to być wywołania pośrednie (indirect call), "zwykłe" wywołania mogą być realizowane jako wywołania bezpośrednie (direct call).

2) Czasem dla ujednolicenia wszystkie wywołania realizowane są jako pośrednie.

3) Jeśli język dopuszcza zmienne typu procedura lub procedury jako wyniki funkcji, to takie zmienne/wyniki również są reprezentowane jako domknięcia (ale takie rzeczy to już w modelu dynamicznym).

4) Jeśli język dopuszcza etykiety jako zmienne/parametry, to takie zmienne/parametry również są reprezentowane jako domknięcia (tak były takie języki, w których etykietę można było przekazać jako parametr, a potem wyskoczyć na zewnątrz z funkcji i wylądować "gdzieś"!).

Środowisko dynamiczne

Dla języków dopuszczających procedury jako wyniki funkcji, zmienne typu procedura lub zmienne/parametry typu etykieta model stosowy nie jest wystarczający.
Jest tak dlatego, że po opuszczeniu procedury mogą nadal pozostawać odwołania do jej obiektów lokalnych, a obiekty te w modelu stosowym byłyby przecież zlikwidowane w chwili opuszczania procedury.
Zatem **w modelu dynamicznym rekord aktywacji procedury nie jest likwidowany w chwili opuszczania tej procedury**, czyli rekordy aktywacji nie mogą być zorganizowane w strukturę stosu. W modelu dynamicznym rekord aktywacji procedury jest likwidowany dopiero, gdy znikną wszystkie odwołania do jego elementów - w tym modelu niezbędny jest garbage collector.

Sama organizacja pojedynczego rekordu aktywacji w modelu dynamicznym jest taka sama jak w modelu stosowym z procedurami lokalnymi.

Modelem dynamicznym nie będziemy się dalej zajmować (może się on przydać np. przy implementacji kompilatorów/interpreterów języków funkcyjnych).

4. Analiza leksykalna

Zadaniem analizy leksykalnej jest przekształcenie tekstu źródłowego programu w ciąg **tokenów** czyli symboli podstawowych języka.
Symbole podstawowe to np.: identyfikatory, stałe, słowa kluczowe, symbole operacji (+,-,*,/), średnik, nawiasy i inne symbole.

Struktura wewnętrzna symboli podstawowych (tokenów) może być opisana za pomocą **wyrażeń regularnych** i co za tym idzie mogą one być rozpoznawane przez **automaty skończeniestanowe**.
I tu bardzo przydaje się wiedza z przedmiotu TAJF (Teoria Automatów i Języków Formalnych). W tym krótkim opracowaniu informacje o wyrażeniach regularnych i automatach skończeniestanowych nie będą przedstawione - zakładamy, że są znane.

Reprezentację zewnętrzną tokenu nazywamy leksemem.
W przypadku wielu tokenów istnieje jednoznaczna odpowiedniość token - leksem (np. tokenowi oznaczającemu operator dodawania odpowiada leksem '+', a tokenowi oznaczającemu operator mnożenia leksem '*'), ale niektóre tokeny mają wiele reprezentacji zewnętrznych i każdej z nich odpowiada inny leksem. Najważniejsze takie tokeny to identyfikatory i stałe.

Moduł kompilatora realizujący analizę leksykalną nazywany jest również skanerem.

Słowa kluczowe

Ważną klasą symboli podstawowych są słowa kluczowe danego języka.
Pewien problem powstaje w związku z tym, że często słowa kluczowe "pasują" do wyrażenia regularnego opisującego identyfikatory, a przecież muszą one być od identyfikatorów odróżnione.

Możliwe jest kilka podejść do tej kwestii.

1) Odróżniać jakoś słowa kluczowe od identyfikatorów.
Można np. wymusić aby słowa kluczowe rozpoczynały się od jakiegoś symbolu nieużywanego do niczego innego (tak jest np. z dyrektywami preprocesora w języku C rozpoczynającymi się od znaku #) albo przyjąć, że słowa kluczowe pisane są wielkimi literami, a identyfikatory małymi.
To rozwiązanie jest jednak niewygodne i podatne na błędy pisowni i dlatego jest rzadko stosowane.

2) Przyjąć, że słowa kluczowe są w ogólności nierozróżnialne od identyfikatorów, a o tym, czy dany leksem to słowo kluczowe czy identyfikator, decyduje kontekst.
I rzeczywiście są języki, w których można definiować identyfikatory np. if, while, int i tym podobne.
Niestety w takich językach można napisać bardzo nieczytelny program (to oczywiście zależy od programisty), ale przede wszystkim takie podejście bardzo komplikuje kompilator i dlatego jest rzadko stosowane (z bardziej znanych języków stosujących to podejście można wymienić Fortran i PL/I, czyli języki dość stare i dziś już niezbyt popularne).

3) Przyjąć, że słowa kluczowe są zastrzeżonymi identyfikatorami, czyli pasują do ogólnego wyrażenia regularnego opisującego identyfikatory, ale nie wolno definiować kolidujących z nimi identyfikatorów. W tym podejściu konieczny może być dodatkowy etap odróżniający słowa kluczowe od innych identyfikatorów.
To podejście jest najpopularniejsze, ale również ma wady.
Głównym problemem jest utrudnienie rozwoju języka. Naturalne jest, że program poprawny w starszej wersji języka powinien być poprawny również w nowszej wersji tego języka. Ale to wymaga od twórców języka, aby z góry przewidzieli, jakie konstrukcje pojawią się w przyszłych wersjach języka i aby zastrzegli identyfikatory odpowiadające słowom kluczowym jeszcze nieobecnym w aktualnej wersji. Bez tego programiści mogliby deklarować identyfikatory, które w przyszłości staną się słowami kluczowymi i w przyszłości takie programy stałyby się niepoprawne.

Ciekawostka:
Czasem twórcy języków ratują się dodając nowe znaczenie dla istniejącego od dawna w języku słowa kluczowego, tak jest np. dla słowa auto w C++. Obecnie jest ono używane w deklaracjach zmiennych, w których kompilator sam ma wywnioskować typ zmiennej na podstawie jej inicjalizatora, a tymczasem słowo kluczowe auto jest w języku C (a zatem i w C++) od samego początku (wczesne lata 70-te) - mało kto pamięta co ono wtedy znaczyło (i nadal znaczy).

Oprócz przekształcenia tekstu źródłowego programu w ciąg tokenów, podczas analizy leksykalnej rozpoczyna się również proces tworzenia **tablicy symboli**.
Tablica symboli zawiera informacje o wszystkich występujących w programie identyfikatorach. Szczegóły zostaną podane później, ale widać, że samo wpisanie identyfikatorów (a ściślej ich zewnętrznej reprezentacji) do tablicy symboli musi nastąpić podczas analizy leksykalnej jako jedynego etapu kompilacji korzystającego bezpośrednio z kodu źródłowego.

Interesującym pomysłem na obsługę słów kluczowych jest wstępne wypełnienie tablicy symboli identyfikatorami odpowiadającymi właśnie słowom kluczowym (w ten sposób nie będzie można zdefiniować tych identyfikatorów w innym znaczeniu, ponieważ wpisy w tablicy symboli muszą być jednoznaczne).

Wyodrębnianie leksemów

W klasycznej teorii automatów i języków formalnych postawiony problem polega na rozpoznaniu, czy dany napis (cały) odpowiada zadanemu wyrażeniu regularnemu.

W przypadku kompilacji jest inaczej. Cały napis (czyli cały kod źródłowy) nie jest przecież pojedynczym leksemem (a to leksemy opisane są wyrażeniami regularnymi), a ciągiem leksemów.
Analizator leksykalny sam musi te leksemy wyodrębnić.

Może się zdarzyć, że pewne słowo ***w*** może być pełnym leksemem odpowiadającym tokenowi ***t1*** i jednocześnie początkiem słowa ***wv*** odpowiadającego tokenowi ***t2***.
W takim przypadku analizator leksykalny musi zdecydować, czy właśnie został wyodrębniony token ***t1***, czy kontynuować analizę w nadziei rozpoznania tokenu ***t2***.

Teoretycznie możliwych jest kilka rozwiązań.

1) Wypróbowywać wszystkie warianty - zdecydowanie odrzucamy ten pomysł (trudności techniczne z wycofywaniem się z błędnych ścieżek, a przede wszystkim nieakceptowalna, wykładnicza złożoność obliczeniowa).

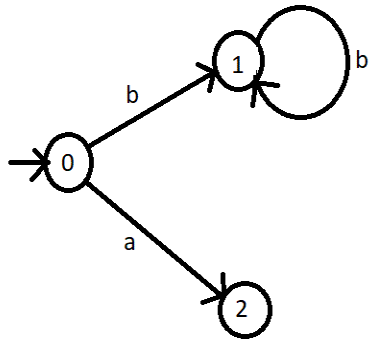
2) "Podglądać" pewną liczbę symboli w przód i na tej podstawie podjąć decyzję - również odrzucamy ten pomysł (problemy podobne jak dla poprzedniego pomysłu chociaż mniejsze).

3) Zastosować **zasadę najdłuższego dopasowania**.
Zasada ta mówi: "Kontynuuj wczytywanie znaków z wejścia dopóki nie napotkasz znaku, który nie może być poprawną kontynuacją właśnie analizowanego leksemu (tego znaku już nie wczytujemy). Jeśli w chwili zatrzymania automat jest w stanie końcowym, to zwróć rozpoznany token (odpowiadający temu stanowi końcowemu), jeśli nie, zwróć błąd leksykalny.
Powyższa wersja zasady najdłuższego dopasowania jest odpowiednia dla większości języków i jest wykorzystywana przez kompilatory. Jednak niektóre języki (np. Fortran) wymagają nieco zmodyfikowanej wersji (szczegóły pomijamy).

Implementacja analizatora leksykalnego

Rozważmy alfabet {a,b} i proste wyrażenie regularne: a + bb*

Wyrażeniu temu odpowiada następujący automat



Istnieją dwie podstawowe metody implementacji automatów skończeniestanowych (zauważmy, że implementacja automatu sprowadza się tak naprawdę do implementacji funkcji przejścia).

Dalej przedstawione zostaną pseudokody obu metod implementacji podanego powyżej przykładowego automatu.

Metoda 1 - za pomocą instrukcji wyboru

```
function Automat1()
   {
   stan := 0;   // początkowy
   while ( stan!=error i nie koniec wejścia )
      {
      znak := następny znak z wejścia;
      case stan of
         {
         0: case znak of
            {
            'a':   stan := 2;
            'b':   stan := 1;
            other: stan := error;
            }
         1: case znak of
            {
            'b': ;    // nic
            other: stan := error;
            }
         2: stan := error;
         }
      }
   return stan;
   }
```

W tej metodzie funkcję przejścia implementujemy jako zagnieżdżoną instrukcję wyboru. Zewnętrzny poziom zależy od stanu, a wewnętrzny od wczytywanego znaku.

Wady:
1) Ręczne kodowanie jest dla większych automatów bardzo żmudne.
2) Każdy automat wymaga kodowania od nowa.

Zalety:
1) Ręczne kodowane w prosty sposób umożliwia optymalizacje.

Metoda 2 - za pomocą tablicy przejść

```
function Automat2()
   {
   stan := 0;   // początkowy
   while ( stan!=error i nie koniec wejścia )
      {
      znak := następny znak z wejścia;
      stan := TabPrzejsc[stan,znak];
      }
   return stan;
   }
```

Tablica przejść dla tego automatu wygląda następująco:

	a	b	other
0	2	1	error
1	error	1	error
2	error	error	error
error	error	error	error

Pierwszy wiersz opisuje znaki z alfabetu (wejście), pierwsza kolumna stany automatu, a pozostała część macierzy stan, do którego przejdzie automat dla danej pary (stan, wejście).

Zauważmy, że jeśli usuniemy z tablicy wpisy 'error' to otrzymamy macierz rzadką

	a	b	other
0	2	1	
1		1	
2			
error			

Oczywiście w przypadku większych automatów macierz również będzie znacznie większa, ale prawdopodobnie będzie też znacznie rzadsza, możliwa więc będzie efektywna pamięciowo implementacja.

Wady: Potencjalnie duża macierz przejść (ale patrz uwaga powyżej)
Zalety: Prosta i ogólna implementacja.

Uwagi końcowe (do obu metod):

1) Każdemu tokenowi odpowiada inny stan końcowy automatu (w klasycznej teorii automatów nie ma znaczenia, w którym ze stanów końcowych zatrzyma się automat, tutaj ma to podstawowe znaczenie).

2) Wygodnie jest dodać specjalny stan error, do którego przechodzimy w chwili zerwania obliczeń automatu (czyli gdy funkcja przejścia nie jest zdefiniowana dla danej pary (stan, wejście)).

3) Z uwagi na to że analizator zatrzymuje się w chwili wczytania znaku, który nie jest poprawną kontynuacją leksemu, konieczna jest dodatkowa, nieznana w teorii automatów, operacja - zwróć znak na wejście.

5. Analiza składniowa - informacje ogólne

Zadaniem analizy składniowej (nazywanej również analizą syntaktyczną) jest rozpoznanie struktury składniowej programu. Dane dla analizy składniowej stanowi ciąg tokenów dostarczonych przez analizę leksykalną, a wynikiem jest drzewo struktury (ang. syntax tree), uzupełniane następnie atrybutami podczas analizy semantycznej. Podczas analizy składniowej uzupełniane są również informacje zawarte w tablicy symboli. Moduł kompilatora dokonujący analizy składniowej nazywany jest parserem (a sama analiza składniowa parsowaniem).

Zauważmy, że przed rozpoczęciem analizy składniowej wcale nie jest potrzeby kompletny ciąg tokenów odpowiadających kodowi źródłowemu, analizator składniowy może wywoływać moduł analizatora leksykalnego każdorazowo, gdy potrzebuje pobrać kolejny token z wejścia i taka organizacja kompilatora jest dominującym rozwiązaniem.

W prostszych przypadkach parser może również nie tworzyć jawnego drzewa struktury do wykorzystania przez kolejne moduły kompilatora, a zamiast tego może po prostu w odpowiednich momentach (po rozpoznaniu pełnej jednostki syntaktycznej, np. wyrażenia, instrukcji, deklaracji) wywołać procedury dokonujące analizy semantycznej i generowania kodu.
Przy takim podejściu analizator składniowy stanowi niejako szkielet kompilatora, do którego "podoczepiane" są inne moduły owego kompilatora. Zatem nic dziwnego, że jeden z pierwszych generatorów analizatorów składniowych nazywał się YACC - Yet Another Compiler-Compiler (Jeszcze Jeden Kompilator Kompilatorów).
Zauważmy, że przy takim podejściu różne fazy procesu kompilacji przeplatają się.

Obsługa błędów składniowych

W praktyce bardzo ważnym elementem analizatora składniowego jest obsługa błędów. Jest tak dlatego, że błędy składniowe są dużo trudniejsze do zdiagnozowania niż błędy leksykalne i semantyczne. Błąd leksykalny (np. nieprawidłowo zapisana liczba albo pojawienie się w kodzie źródłowym znaku spoza alfabetu języka) zostanie rozpoznany natychmiast w miejscu gdzie błąd rzeczywiście wystąpił. Podobnie typowy błąd semantyczny, jakim jest użycie niezadeklarowanego identyfikatora, zostanie natychmiast zauważony przez analizator semantyczny (chociaż o tym, jak go naprawić - dodać deklarację czy poprawić literówkę, to już musi zdecydować programista). Natomiast może się zdarzyć, że bardzo typowy błąd składniowy, jakim jest pominięcie nawiasu zamykającego (np. '}'), zostanie rozpoznany w znacznym oddaleniu od miejsca, w którym w rzeczywistości powstał (w skrajnym przypadku na końcu kodu źródłowego - każdy spotkał się chyba z komunikatem "unexpected end of file" albo jakimś podobnym). A zatem w przypadku błędów składniowych miejsce zauważenia objawu błędu może być znacznie oddalone od miejsca rzeczywistego powstania błędu i, co za tym idzie, właściwa diagnoza przyczyny tego błędu jest utrudniona.

W tym opracowaniu postąpimy jednak tak jak zwykle się postępuje na kursach dotyczących kompilatorów, czyli powiemy, że obsługa błędów jest bardzo ważna, po czym odłożymy ją na później.

Trochę teorii

Uwaga: Podobnie jak w części dotyczącej analizy leksykalnej zakładamy znajomość zagadnień omawianych na przedmiocie TAJF i nie będą one wyjaśniane.

Do formalnego opisu składni języków programowania używa się **gramatyk bezkontekstowych** (ang. context-free grammars), a do ich rozpoznawania służą **automaty ze stosem** (ang. pushdown automaton).
Używane w analizie leksykalnej gramatyki regularne (i wyrażenia regularne) oraz automaty skończeniestanowe nie są wystarczające do opisu składni, ponieważ nie można przy ich pomocy wyrazić konstrukcji rekurencyjnych (zagnieżdżone bloki, zagnieżdżone if'y, struktury jako składowe struktur). Wprawdzie gramatyki bezkontekstowe również nie są wystarczające do pełnego opisu typowego języka programowania, np. nie można przy ich pomocy wyrazić wymagania, aby zmienna była przed użyciem zadeklarowana (nawet w potocznym rozumieniu widać, że to zależność kontekstowa - zmiennej można użyć jedynie w kontekście jej wcześniejszej deklaracji), ale elementy wykraczające poza możliwości gramatyk bezkontekstowych odkładane są do etapu analizy semantycznej.

Niestety w ogólności gramatyki bezkontekstowe rozpoznawane są przez niedeterministyczne automaty ze stosem i niestety, w odróżnieniu od automatów skończeniestanowych, dla niektórych niedeterministycznych automatów ze stosem nie da się skonstruować równoważnego automatu deterministycznego (a przecież tylko taki może być elementem kompilatora). Na szczęście tego rodzaju problemy nie pojawiają się w praktyce.

Z punktu widzenia gramatyk podobny problem stanowią gramatyki niejednoznaczne.
Samo pojęcie gramatyk niejednoznacznych szczegółowo omawiane w tym miejscu nie będzie (jest znane z TAJF), wystarczy nieformalna definicja mówiąca, że gramatyka jest niejednoznaczna gdy dla pewnego słowa możliwe jest więcej niż jedno drzewo wywodu. Jednak problem, co zrobić jeśli "najbardziej intuicyjna" gramatyka opisująca daną konstrukcję językową jest niejednoznaczna, wymaga omówienia (bo oczywiście, mówiąc potocznie, znaczenie każdej konstrukcji językowej musi być zdefiniowane jednoznacznie).

Sposoby radzenia sobie z niejednoznacznością gramatyki

1) Wprowadzenie dodatkowych pozagramatycznych reguł rozstrzygania niejednoznaczności wskazujących, które z wielu możliwych drzew wywodu jest tym właściwym
Zalety:
- Nie zmieniamy gramatyki, czyli jej nie komplikujemy - nadal pozostaje intuicyjna
Wady:
- Sama gramatyka nie jest kompletnym opisem języka (co jest niekorzystne z teoretycznego punktu widzenia)

2) Modyfikacja gramatyki, aby wymusić jej jednoznaczność
Zalety:
- Gramatyka wystarcza do kompletnego opisu języka (to ważne z teoretycznego punktu widzenia)
- Wiadomo, jak poradzić sobie z typowymi przypadkami niejednoznaczności
Wady:
- Zdarza się, że przekształcona gramatyka jest znacznie mniej intuicyjna
- Istnieje niebezpieczeństwo niezamierzonej zmiany języka

Podział metod analizy składniowej

A) **Metody Top-Down**
Rozbiór składniowy rozpoczynamy od symbolu początkowego gramatyki (czyli od korzenia drzewa wywodu, czyli od góry, bo przecież w informatyce drzewa mają korzenie u góry). Wczytując kolejne symbole wejściowe (tokeny) staramy się przewidzieć jaka produkcja została użyta do ich wygenerowania (można to zrobić jednoznacznie jeśli początki prawych stron produkcji są różne). Kolejne wczytywane symbole mogą albo potwierdzać, że prawidłowo przewidzieliśmy użytą produkcję, albo stanowić podstawę do przewidzenia kolejnej produkcji, albo sygnalizować błąd (gdy nie są dozwolone jako kontynuacja analizowanej produkcji).
Te metody odtwarzają tzw. **wyprowadzenie lewostronne** (pojęcie znane z TAJF).

B) **Metody Bottom-Up**
Wczytujemy symbole (tokeny) z wejścia aż wczytamy ciąg symboli, który jest prawą stroną pewnej produkcji, a następnie zastępujemy ów ciąg symbolem stojącym po lewej stronie tej produkcji (nazywa się to redukcją). Kontynuujemy to postępowanie dopóki nie zredukujemy ciągu wszystkich symboli wejściowych do symbolu początkowego gramatyki. Zaczynamy więc od liści drzewa (a one w informatyce są na dole - bottom) i posuwamy się w górę drzewa (w stronę korzenia).
W ten sposób odtwarzamy tzw. **wyprowadzenie prawostronne** (pojęcie znane z TAJF), ale **w odwrotnej kolejności** (ang. reverse order).

Przykłady gramatyk dla wyrażeń arytmetycznych

Rozważymy gramatyki opisujące wyrażenia arytmetyczne składające się z czterech podstawowych działań arytmetycznych, nawiasów oraz liczb.
Zakładamy, że formalna definicja gramatyki bezkontekstowej znana jest z TAJF.

Pierwsza wersja gramatyki - niestety niepoprawna (gramatyka jest niejednoznaczna)

Gramatyka G1 = (N1, T, P1, S)
gdzie:
- zbiór nieterminali: N1 = { exp, op }
- zbiór terminali: T = { +, -, *, / (,), num }
- symbol początkowy: S = exp
- oraz produkcje P1

exp -> exp op exp | (exp) | num
op -> + | - | * | /

Niestety ta gramatyka jest niejednoznaczna, ponieważ nie wprowadza rozróżnienia priorytetów operatorów (znalezienie przykładów pokazujących niejednoznaczność pozostawiamy jako proste ćwiczenie domowe)

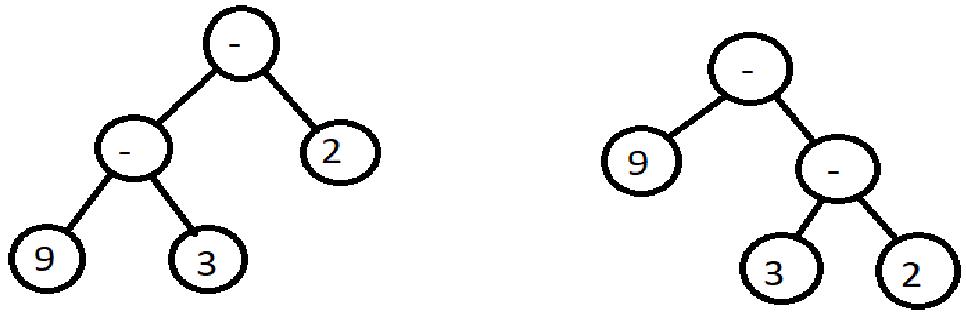
Druga wersja gramatyki - niestety również niepoprawna (gramatyka nadal jest niejednoznaczna)

Gramatyka G2 = (N2, T, P2, S)
gdzie
- zbiór nieterminali: N2 = { exp, term, factor, addop, mulop }
- zbiór terminali: T = { +, -, *, / (,), num } (ten sam co w gramatyce G1)
- symbol początkowy: S = exp
- oraz produkcje P2

exp -> exp addop exp | term
addop -> + | -
term -> term mulop term | factor
mulop -> * | /
factor -> (exp) | num

Ta gramatyka rozróżnia priorytety operatorów addytywnych i multiplikatywnych. W gramatyce takie rozróżnienie uzyskujemy dzięki wprowadzeniu dla każdego poziomu priorytetu oddzielnego nieterminala. Im niższy priorytet operatora, tym "bliżej symbolu początkowego" gramatyki jest produkcja z danym operatorem (taki operator będzie bliżej korzenia drzewa struktury, czyli będzie obliczany później).

Niestety ta gramatyka nadal jest niejednoznaczna, ponieważ nie określa łączności operatorów
Np. dla wyrażenia 9 - 3 - 2 możliwe są dwa drzewa wywodu



Zgodnie z zasadami matematyki chcielibyśmy, aby jedynym poprawnym było drzewo z lewej strony, ponieważ oddaje lewostronną łączność operatorów arytmetycznych (to, że jest narysowane z lewej strony, to przypadek). Wynik obliczeń dla lewego drzewa to 4, a dla prawego 8.

Na poziomie gramatyki przyczyną niejednoznaczności są produkcje zawierające po obu stronach operatora ten sam nieterminal (np. exp addop exp). W tym przypadku, aby usunąć niejednoznaczność, należy jedno z wystąpień nieterminala exp zamienić na nieterminal term. Aby uzyskać pożądaną łączność lewostronną, należy pozostawić exp po lewej stronie operatora, uzyskujemy w ten sposób **gramatykę lewostronnie rekurencyjną**.

Trzecia wersja gramatyki - prawidłowa

Gramatyka G3 = (N2, T, P3, S)
gdzie
- zbiór nieterminali: N2 = { exp, term, factor, addop, mulop } (ten sam, co w G2)
- zbiór terminali: T = { +, -, *, / (,), num } (ten sam, co w G1 i G2)
- symbol początkowy: S = exp
- oraz produkcje P3

exp -> exp addop term | term
addop -> + | -
term -> term mulop factor | factor
mulop -> * | /
factor -> (exp) | num

Ta gramatyka jest jednoznaczna i prawidłowo oddaje priorytety i łączność operatorów.
Co więcej, jest intuicyjna i dobrze oddaje strukturę wyrażeń.

Niestety, jak się wkrótce okaże, również nie jest bez wad.
Niestety, jeśli gramatyka jest lewostronnie rekurencyjna to nie jest gramatyką LL(k) dla żadnego k, a to oznacza, że nie może być podstawą dla analizatora składniowego działającego metodą Top-Down (więcej będzie o tym w dalszej części opracowania).

Problem "wiszącego elsa"

Problem ten pojawia się np. w przypadku następującego fragmentu programu

```
if ( a>0 ) if ( b>3 ) x=1; else x=2;
```

Powyższy fragment jest celowo napisany w jednej linii, aby formatowanie nie sugerowało odpowiedzi. A pytanie jest następujące: czy gałąź else ma być "doczepiona" do if'a z warunkiem (a>0), czy może if'a z warunkiem (b>3)?

Najprostsza intuicyjna gramatyka opisująca instrukcję if jest następująca

ifstmt -> if (condition) stmt | if (condition) stmt else stmt

gdzie nieterminalami są ifstmt, stmt i condition, a terminalami słowa kluczowe if i else oraz nawiasy.

Niestety ta gramatyka jest niejednoznaczna i nie odpowiada na postawione wcześniej pytanie. Przekształcenie gramatyki tak, aby była jednoznaczna, nie jest wcale oczywiste i otrzymana jednoznaczna gramatyka jest zagmatwana i nieintuicyjna.

Dlatego w tym przypadku (inaczej niż dla wyrażeń arytmetycznych) zwykle pozostawia się gramatykę niejednoznaczną i dodaje się **pozagramatyczną regułę** mówiącą, że else "doczepia się" do "najbliższego" if'a.
Okazuje się, że w praktyce dodanie takiej reguły do implementacji automatu jest bardzo proste (a nawet zbyt proste - można nieświadomie rozwiązać poprawnie problem i nawet nie zauważyć, że w ogóle był jakiś problem - a to niedobrze z dydaktycznego punktu widzenia).

6. Analiza składniowa - metody Top-Down

Metody Top-Down przetwarzają wejściowy ciąg tokenów odtwarzając wyprowadzenie lewostronne. Konstrukcję drzewa struktury rozpoczynamy od korzenia (czyli od góry) i poruszamy się w dół w stronę liści. Na podstawie podglądu symboli wejściowych analizatory przewidują, którą produkcję należy zastosować (teoretycznie można podglądać wiele symboli, w praktyce podgląda się jeden) - to tak zwane analizatory przewidujące.

Wśród metod Top-Down można wyróżnić dwie grupy:

1) Analizatory wykorzystujące metodę zejść rekurencyjnych - są odpowiednie do ręcznego pisania analizatorów (i są wystarczająco ogólne, aby były w praktyce wykorzystywane).

2) Analizatory typu LL(1) - obecnie są rzadziej wykorzystywane, do pisania ręcznego są niewygodne, jako podstawa generatorów zostały wyparte przez ogólniejsze metody LR(1). Są jednak cennym przykładem dydaktycznym.

Metoda Zejść Rekurencyjnych (Recursive Descent Method)

Przykład - wyrażenia arytmetyczne

Rozważmy gramatykę wyrażeń:

```
exp    -> exp addop term | term
addop  -> + | -
term   -> term mulop factor | factor
mulop  -> * | /
factor -> ( exp ) | num
```

Idea metody zejść rekurencyjnych jest bardzo prosta. Dla każdego nieterminala gramatyki należy stworzyć funkcję rozpoznającą napis możliwy do wygenerowania z tego nieterminala. Funkcję tę tworzy się na podstawie prawych stron produkcji, w których ten nietermimal jest po lewej stronie.
Tak naprawdę funkcje należy napisać jedynie dla istotnych nietermiali (w powyższej gramatyce to exp, term i factor), pisania funkcji dla nieterminali pomocniczych (w tej gramatyce to addop i mulop) można uniknąć.

Ponadto niezbędna jest funkcja wczytująca kolejny token z wejścia i sprawdzająca, czy jest on zgodny z oczekiwaniem - ta funkcje zwykle nazywana jest Match (dopasuj).
Jeśli token jest zgodny z oczekiwaniami, funkcja po prostu przesuwa wejście (mamy podgląd kolejnego tokenu), a jeśli jest niezgodny, to zgłaszany jest błąd (można zgłosić błąd na różne sposoby, np. za pomocą mechanizmu wyjątków - szczegóły pomijamy (ale pamiętamy, że obsługa błędów jest bardzo ważna).

Niestety próba bezpośredniego wykorzystania powyższej gramatyki jako podstawy dla analizatora stosującego metodę zejść rekurencyjnych nie powiedzie się.

Konieczne jest przekształcenie gramatyki.
A ściślej nie przekształcenie (jak to miało miejsce w poprzednim rozdziale), a zapis tej samej gramatyki z użyciem nieco innej notacji.

Powyższa gramatyka zapisana jest w **zwykłej notacji BNF** (Backus-Naur Form).
Niektóre produkcje zapisane w tej notacji nie mogą być bezpośrednio wykorzystane przy tworzeniu parsera wykorzystującego metodę zejść rekurencyjnych. W szczególności nie mogą być wykorzystane produkcje lewostronnie rekurencyjne np.

```
exp  ->  exp addop term | term
```

Jest tak dlatego, że funkcje dla nieterminala stojącego po lewej stronie produkcji powstają na podstawie prawej strony tej produkcji. Zatem wnętrze funkcji Exp (odpowiadającej nieterminalowi exp) musiałoby rozpoczynać się od wywołania funkcji Exp - i powstaje nieskończony ciąg wywołań rekurencyjnych.

Aby poradzić sobie z tym problemem należy zastanowić się, jak wygląda "rozwinięcie" prawej strony tej produkcji - jest to ciąg symboli

```
term addop term  addop term ... addop term
```

który w połączeniu z alternatywną prawą stroną może też zredukować się do pojedynczego symbolu

```
term
```

I taki właśnie ciąg jest w **rozszerzonej notacji BNF** (ang. Extended Backus-Naur Form, w skrócie **EBNF**) zapisywany jako

```
term { addop term }
```

gdzie nawiasy { } są odpowiednikiem * w notacji wyrażeń regularnych, czyli oznaczają dowolną liczbę (w tym 0) powtórzeń swojej zawartości.

Zatem w notacji EBNF gramatyka przyjmuje postać

```
exp    ->  term { addop term }
addop  ->  + | -
term   ->  factor { mulop factor }
mulop  ->  * | /
factor ->  ( exp ) | num
```

Tak zapisana gramatyka może już być podstawą dla parsera stosującego metodę zejść rekurencyjnych (nie ma groźby nieskończonego ciągu wywołań rekurencyjnych).

W załączniku do tego opracowania znajdują się dwa kompletne przykłady (można je skompilować i uruchomić) kalkulatorów wyrażeń arytmetycznych, stosujących do parsowania tych wyrażeń metodę zejść rekurencyjnych. Jeden z przykładów korzysta z opisanej powyżej gramatyki w notacji EBNF, a drugi z gramatyki w klasycznej notacji, ale z usuniętą lewostronną rekursją (usuwanie lewostronnej rekursji bez zmiany notacji na EBNF będzie omówione w dalszej części opracowania).
Zrealizowane tam kalkulatory nie obliczają wyrażenia bezpośrednio, a konstruują najpierw drzewo struktury i obliczenia wykonują na podstawie tego drzewa. Jest to bliższe działaniu kompilatora, wystarczy zamiast procedury obliczania wartości wyrażenia umieścić procedurę generowania kodu dla tego wyrażenia (co też robi się na podstawie drzewa) i otrzymamy fragment kompilatora. Jeśli skompilowane przykłady wywołamy z dodatkowym parametrem

(nazwa pliku) to we wskazanym pliku umieszczony zostanie XML-owy opis wygenerowanego drzewa (oba kalkulatory pomimo różnic wewnętrznych tworzą dla tego samego wyrażenia identyczne drzewa struktury).

Problem "wiszącego elsa"

Instrukcję if (z elsem lub bez) opisuje następujący intuicyjny fragment gramatyki:

```
ifstmt  ->  if ( exp ) stmt  |  if ( exp ) stmt else stmt
```

Niestety produkcja ta jest niejednoznaczna.
Pomimo tego może być podstawą analizatora stosującego metodę zejść rekurencyjnych - trzeba tylko w implementacji analizatora dodać pozagramatyczną regułę rozstrzygania niejednoznaczności. Okazuje się, że robi się to w tak naturalny sposób, że można w ogóle nie zauważyć istnienia właśnie rozwiązanego problemu.

Przedstawiona dalej funkcja IfStmt nie tylko rozpoznaje instrukcję if, ale tworzy też drzewo struktury odpowiedniego fragmentu programu. Przyjmujemy przy tym, że
- węzeł drzewa struktury ma typ SyntaxTree
- funkcja MakeStmtNode tworzy węzeł drzewa struktury odpowiadający instrukcji/operacji określonej przez jej argument
- funkcja Stmt rozpoznaje dowolną instrukcję, a funkcja Exp wyrażenie
- funkcja Match działa we wcześniej opisany sposób

```
SyntaxTree IfStmt()
{
SyntaxTree temp;
Match(IF);                  // dopasowanie słowa kluczowego if
temp = MakeStmtNode(IF);    // utworzenie węzła odpowiadającego instrukcji if
Match(OPEN_PAR);            // dopasowanie nawiasu otwierającego
temp.test = Exp();          // rozpoznanie i utworzenie drzewa dla wyrażenia
Match(CLOSE_PAR);           // dopasowanie nawiasu zamykającego
temp.thenPart = Stmt();     // rozpoznanie i utworzenie drzewa dla sekcji then
if ( token == ELSE )        // test, czy następny token to else
    {
    Match(ELSE);            // jeśli tak, to dopasowanie słowa kluczowego else
    temp.elsePart = Stmt(); // rozpoznanie i utworzenie drzewa dla sekcji else
    }
else
    temp.elsePart = NULL;   // jeśli nie było else to sekcja else jest pusta (NULL)
return temp;                // zwracanie utworzonego fragmentu drzewa
}
```

Pozagramatyczna reguła rozstrzygania niejednoznaczności ukryta jest w linii
`if ( token == ELSE )`
Tu właśnie "doczepiamy" else do aktualnie analizowanego (czyli "najbliższego") if'a.
Przy okazji powyższa linia rozwiązuje drugi problem pojawiający się w analizatorach korzystających z metod Top-Down, czyli wspólne początki różnych wariantów prawych stron produkcji (pierwszym problemem omówionym przy okazji wyrażeń są produkcje

lewostronnie rekurencyjne). W ogólności problem wspólnych początków jest bardziej złożony i może wymagać głębszych analiz (przecież analizator ma wybrać produkcję na podstawie jednego podglądanego symbolu i jeśli rozpoczyna on wiele prawych stron, to jest problem), ale w tym przypadku rozwiązanie jest, jak widać, proste.

Analizatory typu LL(1)

Analizatory tego rodzaju do implementacji automatu ze stosem używają jawnego stosu, a nie mechanizmu rekursji jak w metodzie zejść rekurencyjnych.
Nazwa metody analizy składniowej pochodzi stąd, że analizatory przetwarzają wejście od strony lewej do prawej (pierwsza litera L), odtwarzają wyprowadzenie lewostronne (druga litera L) i podglądają 1 symbol (token) z wejścia (liczba 1). W ogólności istnieje cała grupa analizatorów LL(k) podglądających k symboli, ale w praktyce korzysta się jedynie z LL(1), ponieważ zwiększenie k znacząco komplikuje analizator, a nie rozwiązuje żadnych istotnych problemów (z lewostronną rekursją i wspólnymi początkami prawych stron różnych produkcji).

Przykład
Rozważmy prościutką gramatykę

S -> (S) S | ε

Gramatyka ta definiuje język poprawnie sparowanych nawiasów.
Rozważmy też proste wejście: ( )
Ponadto znak $ oznacza dno stosu (pusty stos) oraz koniec wejścia (odpowiednik EOF).

Analiza przebiega w następujący sposób

Nr	Stos parsera	Wejście	Akcja
1	$ S	()$	S -> (S) S
2	$ S) S (	()$	dopasuj
3	$ S) S	)$	S -> ε
4	$ S)	)$	dopasuj
5	$ S	$	S -> ε
6	$	$	akceptuj

Krok 1) Zaczynamy z początkowym symbolem gramatyki na stosie i analizowanym napisem na wejściu. Jako akcję wybieramy zastosowanie produkcji, z której da się wygenerować pierwszy znak z wejścia (znak ten jest dostępny w podglądzie i na tej podstawie możemy wybrać odpowiednią produkcję). Jeśli na szczycie stosu jest nieterminal, to zawsze wybieramy jako akcje zastosowanie jednej z produkcji, w których ten nieterminal jest po lewej stronie. Zastosowanie produkcji polega na zdjęciu ze stosu szczytowego nieterminala i w zamian umieszczeniu na stosie prawej strony zastosowanej produkcji w odwrotnej kolejności (kolejność odwracamy aby na szczycie stosu był początek prawej strony). Akcja zastosowania produkcji nie zmienia wejścia.

Krok 2) Teraz na szczycie stosu jest terminal. W takiej sytuacji wykonujemy akcję *dopasuj* (match). Akcja ta polega na porównaniu symbolu na szczycie stosu z symbolem na początku wejścia. Jeśli są one równe to usuwamy symbol ze szczytu stosu (pop) i wczytujemy symbol z wejścia (przesuwamy "głowicę czytającą" wejście). Jeśli symbole są różne to zgłaszamy błąd.

Krok 3) Teraz stosujemy ε-produkcję, co odpowiada usunięciu symbolu ze szczytu stosu bez zmiany wejścia.

Kroki 4) i 5) są analogiczne do kroków 2) i 3)

Krok 6) Stos i wejście są puste, to oznacza pomyślną akceptację ciągu wejściowego. Sytuacja, gdy stos jest pusty, a wejście niepuste oznacza błąd (nieprawidłowy ciąg wejściowy). Wyjaśnienia wymaga sposób wyboru produkcji, którą należy zastosować, gdy na szczycie stosu analizatora jest nieterminal. Wykorzystywana jest do tego tzw. tablica przejść analizatora LL(1). Jest to 2-wymiarowa tablica oznaczana zwykle przez M, której wiersze indeksowane są nieterminalami (N), a kolumny terminalami (T) gramatyki. Element M[N,T] tablicy przejść zawiera produkcje, których lewą stroną jest nieterminal N,
a prawa strona rozpoczyna się od terminala T (a ściślej terminal T może rozpoczynać napis wygenerowany z prawej strony produkcji). Są też dodatkowe reguły dotyczące produkcji z wycieralną prawą stroną.

Dla gramatyki z powyższego przykładu tablica przejść jest postaci

M[N,T]	(	)	$
S	S -> (S) S	S -> ε	S -> ε

Uwaga 1: Nie wolno zapominać o kolumnie dla dodatkowego terminala $ (koniec wejścia).

Aby algorytm działał poprawnie, żąda się aby z każdym elementem macierzy była związana co najwyżej jedna produkcja. I taka właśnie jest nieformalna definicja gramatyk LL(1).

Def. (nieformalna). Gramatyka bezkontekstowa jest typu LL(1), gdy związana z nią tablica przejść zawiera na każdej pozycji co najwyżej jedną produkcję.

Uwaga 2: Jeśli z jakimś elementem tablicy przejść związana jest więcej niż jedna produkcja, to gramatyka nie jest typu LL(1) i oczywiście nie może być podstawą analizatorów LL(1). Taka sytuacja nazywana jest konfliktem w tablicy przejść.

Uwaga 2a: Konflikt w tablicy przejść nie musi oznaczać, że gramatyka jest niejednoznaczna, istnieją jednoznaczne gramatyki bezkontekstowe, które nie są LL(1). Ale oczywiście każda gramatyka niejednoznaczna spowoduje konflikty.

Uwaga 2b: Niektóre konflikty i związane z nimi niejednoznaczności można w prosty sposób rozwiązać. Np. w przypadku niejednoznaczności związanych z "wiszącym elsem" pojawi się konflikt, w którym jedną z dwóch produkcji związanych z pewnym elementem tablicy przejść jest ε-produkcja - należy ją po prostu zignorować.

Uwaga 3: Dla bardziej złożonych gramatyk wiele elementów tablicy przejść jest pustych - nie oznacza to błędu w gramatyce. Natomiast odwołanie do takiego pustego elementu tablicy przejść podczas działania analizatora oznacza błędne wejście.

Na kolejnej stronie przedstawiony jest algorytm działania analizatora LL(1)

Algorytm analizatora LL(1)

```
umieść na stosie symbol początkowy gramatyki S;
while ( stos niepusty )
    {
    top = szczyt stosu;
    next = następny token z wejścia;  // podgląd
    switch ( top )
        {
        terminal:
            if ( top == next )
                {
                usuń szczyt stosu;
                przesuń wejście;
                }
            else
                throw Error();
            break;
        nieterminal:
            if ( M[top,next] )
                {
                usuń szczyt stosu;
                umieść na stosie prawą stronę produkcji M[top,next]
                        w kolejności odwrotnej
                }
            else
                throw Error();
            break;
        }  // switch ( top )
    }  // while
if ( wejście puste )
    akceptuj;
else
    throw Error();
```

Pozostaje opisać formalnie sposób tworzenia macierzy przejść M i wtedy proces konstruowania i korzystania z analizatorów LL(1) będzie w pełni zautomatyzowany.

Do utworzenia tablicy przejść konieczne jest obliczenie zbiorów First i Follow. Definicje tych zbiorów powinny być znane z TAJF, teraz przypomnę jedynie ideę tych zbiorów.
- First(A), gdzie A jest nieterminalem, jest to zbiór terminali, od których może rozpoczynać się jakikolwiek napis wygenerowany z nieterminala A
- First(α), gdzie α jest ciągiem symboli gramatyki, jest to zbiór terminali, od których może rozpoczynać się jakikolwiek napis wygenerowany z ciągu α
- Follow(A), gdzie A jest nieterminalem, jest to zbiór terminali, które mogą występować bezpośrednio po napisie wygenerowanym z nieterminala A

Do utworzenia tablicy przejść potrzebne będzie obliczenie zbiorów First dla prawych stron wszystkich produkcji oraz zbiorów Follow dla tych nieterminali, dla których istnieją produkcje z wycieralnymi prawymi stronami.

Algorytm wyznaczania zbiorów First
dla pojedynczych symboli gramatyki

```
foreach ( a ∈ T) First(a) = {a}
foreach ( A ∈ N) First(A) = ∅
zmiany = true
while ( zmiany )
    {
    zmiany = false
    foreach ( A→a1a2...an∈P )                  // dla każdej produkcji gramatyki
        {
        for ( k=1 ; k≤n ; k=k+1 )              // dla każdego symbolu ak
            {                                  // z prawej strony produkcji
            if ( First(ak) - {ε} ⊄ First(A) )  // sprawdzamy czy ak należy do
                {                              // zbioru First(A) z poprzedniej iteracji
                zmiany = true
                First(A) = First(A) ∪ (First(ak) - {ε})
                }
            if (ε ∉ First(ak) )                // jeśli ak nie jest wycieralny to
                break                          // kolejnych symboli nie analizujemy
            }
        if ( k==n+1 )                          // jeśli prawa strona jest wycieralna
            First(A) = First(A) ∪ {ε}          // to do First(A) dodajemy ε
        } //   foreach ( A→a1a2...an∈P )
    } //   while ( zmiany )
```

Algorytm wyznaczania zbiorów First
dla ciągów symboli gramatyki $\alpha = a_1a_2...a_n$

```
First(α) = ∅
for ( k=1 ; k≤n ; k=k+1 )
    {
    First(α) = First(α) ∪ (First(ak) - {ε})
    if (ε ∉ First(ak) )
        break
    }
if ( k==n+1 )
    First(α) = First(α) ∪ {ε}
```

Kluczowe fakty dotyczące zbiorów Follow

1) $\$ \in Follow(S)$
2) Jeśli $A \to \alpha B \beta \in P$ to $First(\beta)-\{\varepsilon\} \subseteq Follow(B)$
3) Jeśli $A \to \alpha B \beta \in P$ oraz $\varepsilon \in First(\beta)$ to $Follow(A) \subseteq Follow(B)$

Algorytm wyznaczania zbiorów Follow
dla pojedynczych symboli gramatyki

```
foreach ( A ∈ N) Follow(A) = ∅
Follow (S) = {$}
zmiany = true
while ( zmiany )
    {
    zmiany = false
    foreach ( A→a1a2...an∈P )      // dla każdej produkcji gramatyki
        foreach ( ak ∈ N )         // dla każdego nieterminala ak
            {                      // z prawej strony produkcji
            X = Follow(ak)         // X = zbiór Follow z poprzedniej iteracji
            Follow(ak) = Follow(ak) ∪ (First(ak+1ak+2...an) - {ε})      // fakt 2
            if ( ε ∈ First(ak+1ak+2...an) lub k==n )                   // fakt 3
                Follow(ak) = Follow(ak) ∪ Follow(A)
            if ( Follow(ak) ≠ X )  // zbiór Follow zmienił się
                zmiany = true
            }
    }
```

Algorytm tworzenia tablicy przejść M[N,T] analizatora LL(1)

```
foreach ( A ∈ N )                  // dla każdego nieterminala z gramatyki
    foreach ( a ∈ T ∪ {$} )        // dla każdego terminala z gramatyki
        M[A,a] = ∅
foreach ( A -> α )                 // dla każdej produkcji gramatyki
    {
    foreach ( a ∈ First(α) )
        M[A,a] = M[A,a] ∪ { A->α }
    if ( ε ∈ First(α) )
        foreach ( a ∈ Follow(A) )
            M[A,a] = M[A,a] ∪ { A->α }
    }
```

Przekształcenia gramatyki

Na poprzednich stronach opisany został algorytm działania analizatorów LL(1) i sposób tworzenia tablicy przejść tych analizatorów. Co jednak zrobić, jeśli w tablicy przejść pojawiły się konflikty? W takim przypadku należy zmodyfikować gramatykę, oczywiście tak, aby nie zmienić rozpoznawanego języka.

Uwaga: Opisane dalej przekształcenia nie gwarantują, że otrzymana w ich wyniku gramatyka będzie typu LL(1), ale jeśli nie zostaną wykonane, to gramatyka na pewno nie będzie LL(1).

Usuwanie lewostronnej rekursji

Lewostronna rekursja w naturalny sposób pojawia się przy opisie wyrażeń arytmetycznych, ponieważ dobrze oddaje lewostronną łączność operatorów.
Niestety **gramatyka lewostronnie rekurencyjna nie jest LL(1)** (uzasadnienie pozostawiamy jako ćwiczenie domowe).
W przypadku analizatorów typu LL(1) nie stosujemy sztuczki ze zmianą sposobu zapisu gramatyki na notację EBNF. Należy przekształcić gramatykę, pozostając przy klasycznej notacji.
Produkcje lewostronnie rekurencyjne mogą mieć kilka form. Zajmiemy się jedynie tymi najprostszymi.

1) Prosta lewostronna rekursja bezpośrednia opisana jest przez produkcje postaci

$A \rightarrow A\,\alpha \mid \beta$

przy czym β nie zaczyna się od A.
Te produkcje zastępujemy następującymi (dodając nieterminal A')

$A \rightarrow \beta\,A'$

$A' \rightarrow \alpha\,A' \mid \varepsilon$

Łatwo zauważyć, że obie wersje gramatyki generują ten sam język (β α α ... α),
a wersja druga nie jest lewostronnie rekurencyjna.

2) Ogólna lewostronna rekursja bezpośrednia opisana jest przez produkcje postaci

$A \rightarrow A\,\alpha 1 \mid A\,\alpha 2 \mid A\,\alpha 3 \mid \ldots \mid A\,\alpha n \mid \beta 1 \mid \beta 2 \mid \beta 3 \mid \ldots \mid \beta m$

przy czym żadne z β1, β1, ... , βk nie zaczyna się od A.
Te produkcje zastępujemy następującymi (dodając nieterminal A')

$A \rightarrow \beta 1\,A' \mid \beta 2\,A' \mid \beta 3\,A' \mid \ldots \mid \beta m\,A'$

$A' \rightarrow \alpha 1\,A' \mid \alpha 2\,A' \mid \alpha 3\,A' \mid \ldots \mid \alpha n\,A' \mid \varepsilon$

Istnieje jeszcze lewostronna rekursja pośrednia, która również sprawia, że gramatyka nie jest typu LL(1). Taką rekursję również można wyeliminować (szczegóły pomijamy).

Lewostronna faktoryzacja

Innym problemem sprawiającym, że gramatyka nie jest LL(1) są wspólne początki (przedrostki) wielu prawych stron produkcji z tym samym nieterminalem z lewej strony. Takie produkcje również w naturalny sposób pojawiają się w gramatyce (np. if z elsem lub bez)

W najprostszym przypadku produkcje te mają postać:

$A \rightarrow \alpha\,\beta \mid \alpha\,\gamma$

I zamieniamy je na produkcje

$A \rightarrow \alpha\,A'$

$A' \rightarrow \beta \mid \gamma$

z dodatkowym nieterminalem A'
Takie przekształcenie nazywane jest lewostronną faktoryzacją.

W ogólnym przypadku więcej niż dwie prawe strony mogą mieć wspólny przedrostek, a nawet dla różnych par prawych stron owe wspólne przedrostki mogą być różnej długości. Istnieje algorytm faktoryzacji również dla takiego ogólnego przypadku (szczegóły pomijamy).

Powtórzona uwaga: Opisane wyżej przekształcenia nie gwarantują, że otrzymana w ich wyniku gramatyka będzie typu LL(1), ale jeśli nie zostaną wykonane, to gramatyka na pewno nie będzie LL(1).

7. Analiza składniowa - metody Bottom-Up

Metody Bottom-Up przetwarzają wejściowy ciąg tokenów od strony lewej do prawej odtwarzając wyprowadzenie prawostronne. Drzewo wywodu konstruowane jest począwszy od liści (od dołu), a poruszamy się w stronę korzenia (do góry). Analizator odczytuje kolejne symbole z wejścia i umieszcza je na stosie. Akcja ta nazywana jest przesunięciem (shift). W chwili, gdy na szczycie stosu znajdzie się ciąg symboli stanowiący prawą stronę produkcji użytej w wyprowadzaniu, analizator wykonuje drugi rodzaj akcji, czyli redukcję (reduce) prawej strony produkcji do symbolu stanowiącego jej lewą stronę. W praktyce oznacza to, że ciąg symboli ze szczytu stosu zastępowany jest pojedynczym nieterminalem. Umieszczenie na stosie symbolu początkowego gramatyki przy odczytaniu całego wejścia oznacza pomyślne zakończenie analizy. Postępując w ten sposób odtwarzamy wyprowadzenie prawostronne, ale w odwrotnej kolejności (reverse order).

Metody Bottom-Up, podobnie jak Top-Down stanowią całą grupę metod opartych na tym samym pomyśle, ale różniących się szczegółami realizacyjnymi i zakresem stosowalności. Wszystkie te metody są jednak bardzo skomplikowane i niezbyt nadają się do bezpośredniej implementacji, są natomiast podstawą dla generatorów parserów.

Ogólnie można powiedzieć, że metody Bottom-Up są silniejsze niż Top-Down (np. lewostronna rekursja nie jest dla nich żadnym problemem). Zauważmy, że w metodach Top-Down próbowaliśmy przewidzieć, która produkcja została zastosowana na podstawie pojedynczego podglądanego symbolu, natomiast w metodach Bottom-Up decyzję o wyborze produkcji (czyli wykonaniu operacji redukcji) podejmujemy dopiero gdy przeczytamy całą prawą stronę tej produkcji, czyli w momencie, w którym mamy znacznie więcej informacji. Zatem większa "moc" metod Bottom-Up nie jest niczym dziwnym.

Rodzina metod LR(k)

Metody Bottom-Up analizują wejście od strony lewej do prawej (stąd litera L w nazwie), odtwarzają wyprowadzenie prawostronne (stąd litera R w nazwie) i (przynajmniej teoretycznie) mogą podglądać k symboli (stąd k w nazwie).

1) Najprostszą metodą z tej grupy jest metoda LR(0). Jak widać z nazwy, w ogóle nie korzysta ona z podglądu. Niestety jest zbyt słaba (co nie jest dziwne), aby rozpoznać wszystkie konstrukcje spotykane w językach programowania, nie ma więc znaczenia praktycznego. Jednak jako najprostsza z całej grupy jest często wykorzystywana do pokazania idei działania metod LR.

2) Nieco mocniejsza, ale również nie rozwiązująca wszystkich problemów praktycznych, jest metoda SLR(1), czyli Simple LR(1). Jest ona prostym rozszerzeniem metody LR(0) o podgląd jednego symbolu.

4) (Nietypowa numeracja celowa) Najsilniejszą omawianą metodą jest klasyczna metoda LR(1) (nazywana też kanoniczną metodą LR(1)). Metoda ta umożliwia rozpoznawanie wszelkich konstrukcji pojawiających się w praktyce w językach programowania. Wymaga jednak bardzo dużych tablic pomocniczych (jest tak dlatego,

że podglądany symbol uwzględniany jest od samego początku konstrukcji analizatora, w metodzie SLR(1) podgląd jest dodawany do "gotowego" analizatora LR(0)).

3) W praktyce często stosuje się analizatory LALR(1) (ang. Lookahead LR(1)). Są one nieco słabsze niż klasyczne analizatory LR(1), ale wystarczające do zastosowań praktycznych, a wymagają znacznie mniejszych tablic pomocniczych (można powiedzieć, że to skompresowane tablice z metody LR(1)).
Znane generatory YACC i Bison korzystają właśnie z metody LALR(1).

Analizatory LR, podobnie jak LL, do implementacji automatu ze stosem używają jawnego stosu. Poniżej przedstawiony zostanie przykład działania analizatora LR(0).

Przykład

Rozważmy gramatykę analogiczną do tej z przykładu dla analizatorów LL. Z przyczyn technicznych (szczegóły pomijamy) dodamy jednak nowy symbol początkowy i dodatkową produkcję. Gramatyka ma zatem postać

S' -> S
S -> (S) S | ε

Gramatyka ta definiuje język poprawnie sparowanych nawiasów.
Rozważmy też proste wejście: ( )
Ponadto znak $ oznacza dno stosu (pusty stos) oraz koniec wejścia (odpowiednik EOF).

Analiza przebiega w następujący sposób

Nr	Stos parsera	Wejście	Akcja
1	$	( ) $	przesunięcie
2	$ (	) $	redukcja S -> ε
3	$ (S	) $	przesunięcie
4	$ (S)	$	redukcja S -> ε
5	$ (S) S	$	redukcja S -> (S) S
6	$ S	$	redukcja S' -> S
	$ S'	$	akceptacja

Analizator wykonuje dwa podstawowe rodzaje akcji

- Przesunięcie (shift) - odczytanie tokenu z wejścia (przesunięcie głowicy czytającej) i umieszczenie go na stosie
- Redukcja (reduce) - usunięcie kilku szczytowych elementów stosu i umieszczenie na ich miejscu pojedynczego nieterminala

Uwagi:
1) Zauważmy, że analizator musi mieć możliwość wglądu dowolnie daleko "w głąb stosu", aby rozpoznać możliwość redukcji, ale to nie jest problem, ponieważ na stosie są elementy już odczytane, zatem to zupełnie co innego niż dowolnie głęboki podgląd wejścia.
2) Zauważmy, że w przypadku redukcji S -> ε nic ze stosu nie usuwamy, a jedynie umieszczamy na nim S.
3) Szczegółowe zasady wyboru, czy dokonać przesunięcia czy redukcji (i jakiej) są dość skomplikowane i to właśnie one stanowią o różnicach pomiędzy różnymi wariantami metod Bottom-UP.

Z konstrukcją analizatorów LR związana jest rozbudowana teoria, której większość pominiemy, wprowadźmy jednak nieco terminologii.

1) W każdym momencie zawartość stosu połączona (konkatenacja) z jeszcze nieodczytanym wejściem stanowi tak zwaną **prawostronną formę zdaniową** (right sentential form), czyli formę zdaniową, którą można otrzymać z symbolu początkowego gramatyki za pomocą wyprowadzenia prawostronnego (oczywiście dotyczy to poprawnego napisu wejściowego).

2) Ciąg symboli znajdujący się na stosie nazywany jest **żywotnym przedrostkiem** (viable prefix) prawostronnej formy zdaniowej.

3) Jeśli na szczycie stosu znajduje się ciąg symboli stanowiący prawą stronę pewnej **produkcji wykorzystanej w wyprowadzeniu prawostronnym**, to analizator wykonuje akcję redukcji zgodnie z tą produkcją. Jej prawa strona (czyli ciąg symboli ze szczytu stosu) nazywa się **podstawą (handle) prawostronnej formy zdaniowej** (można się też spotkać z terminem uchwyt prawostronnej formy zdaniowej - to dosłowne tłumaczenia słowa handle).

Prawidłowe rozpoznawanie podstaw (uchwytów) jest głównym zadanie analizatorów Bottom-Up. Zauważmy, że nie każde pojawienie się na szczycie stosu prawej strony jakiejś produkcji oznacza znalezienie podstawy (uchwytu) i możliwość wykonania redukcji. Można to zrobić jedynie gdy w wyniku otrzymamy poprawną prawostronną formę zdaniową. Zauważmy też, że prawa strona ε-produkcji (czyli nic) zawsze znajduje się na szczycie stosu (a przecież nie zawsze stosujemy taką redukcję!).

Okazuje się, że do sterowania przebiegiem procesu analizy metodami LR można wykorzystać automat skończeniestanowy. Czyli do sterowania działaniem automatu ze stosem wykorzystujemy automat skończeniestanowy. Ów wewnętrzny automat skończeniestanowy konstruuje się na podstawie analizowanej gramatyki, a jego stanami są tak zwane LR(0)-pozycje (albo LR(1)-pozycje, zależnie od tego czy mamy analizator LR(0) czy LR(1)).

I tu zaczyna się bardzo ciekawa i niestety skomplikowana teoria leżąca u podstaw analizatorów LR, którą całkowicie pominiemy.
A w zasadzie prawie całkiem pominiemy, ponieważ jeden element teorii "wyłazi na zewnątrz" i wiedza o nim jest przydatna przy korzystaniu z generatorów parserów (np. YACC, Bison), a tym będziemy się zajmować.

Otóż w pierwszym kroku konstrukcji automatu sterującego (którego stanami są pozycje gramatyki) otrzymujemy automat niedeterministyczny. W skonstruowanym na podstawie tego automatu niedeterministycznego automacie deterministycznym stanami są już zbiory pozycji gramatyki. I tu pojawia się problem, ponieważ to właśnie te pozycje decydują, jaką akcję ma podjąć "główny" automat. Może się okazać, że spośród kilku pozycji wchodzących w skład stanu automatu sterującego, jedna nakazuje inną akcję, a druga inną - jest to tak zwany konflikt (zwróćmy uwagę, że to całkiem podobne do sytuacji, gdy na tej samej pozycji tablicy przejść analizatora LL mamy informacje o kilku produkcjach). Konflikt może świadczyć o niejednoznaczności gramatyki, ale nie musi - być może gramatyka jest jednoznaczna, ale nie jest klasy LR(k) - nie każda jednoznaczna gramatyka bezkontekstowa jest klasy LR(k).

Oczywiście najlepsze są gramatyki, dla których nie ma konfliktów.
Jednak czasem możemy (świadomie!) zaakceptować gramatykę powodującą konflikty przesunięcie-redukcja (shift-reduce). Taki konflikt powoduje np. najbardziej naturalna wersja

gramatyki opisującej instrukcje if z elsem i bez. Generatory parserów mają zwykle wbudowaną regułę rozstrzygającą taki konflikt poprzez wybór akcji przesunięcie (i to da dobry wynik dla wspomnianej gramatyki opisującej if'y).
Natomiast konflikt redukcja-redukcja zwykle świadczy o błędzie w definicji gramatyki i konieczności jej poprawienia.

Na zakończenie pominiętej teorii jeszcze kilka faktów (do uwierzenia bez dowodów).

Niech G oznacza gramatykę bezkontekstową
Zachodzą następujące implikacje
1) Jeśli G jest LL(k), to jest LL(k+1).
2) Jeśli G jest LR(k), to jest LR(k+1).
3) Jeśli G jest LL(k), to jest LR(k).
4) Jeśli G jest LALR(1), to jest LR(1).
5) Jeśli G jest SLR(1), to jest LALR(1).
6) Jeśli G jest LR(0), to jest SLR(1).
Nie zachodzi żadna z implikacji przeciwnych do powyższych.

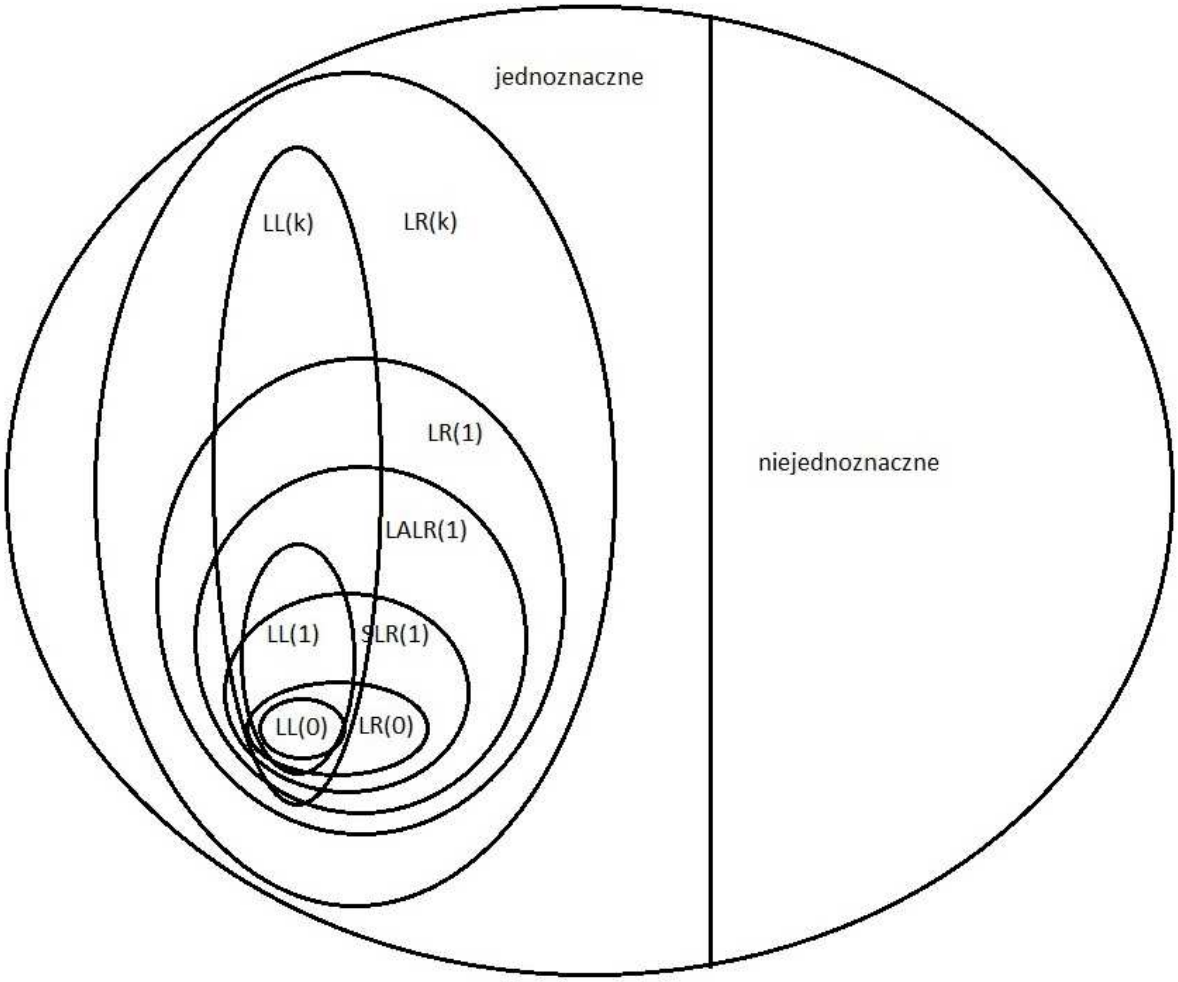
Ponadto
Każdy jednoznaczny język bezkontekstowy ma gramatykę LR(1), ale może oczywiście też mieć inne gramatyki (czyli gramatyki definiujące ten sam język), które nie są LR(1).

Przykład
Rozważmy trzy gramatyki definiujące ten sam język

G0	z produkcjami:	S -> a A c	i	A -> A b b | b
G1	z produkcjami:	S -> a A c	i	A -> b b A | b
G2	z produkcjami:	S -> a A c	i	A -> b A b | b

G0 jest LR(0), G1 jest LR(1) (ale nie LR(0)), a G2 nie jest LR(k) dla żadnego k.

Na zakończenie rysunek pokazujący zależności pomiędzy różnymi klasami gramatyk bezkontekstowych.



8. Obsługa błędów

Na wstępnie warto zauważyć, że specyfikacje języków programowania nie zawierają informacji co robić w razie wystąpienia błędu, zatem ta kwestia zależy całkowicie od twórcy kompilatora.
Aby uzyskać dobrą obsługę błędów, powinna ona być zaplanowana od początku projektu kompilatora, co zresztą korzystnie wpływa również na inne elementy tego projektu.

Dobra obsługa błędów powinna:
1) Zgłaszać możliwie precyzyjne komunikaty o błędach.
2) Po rozpoznaniu błędnej konstrukcji odzyskiwać kontrolę i kontynuować analizę w celu wykrycia potencjalnych kolejnych błędów.
3) Nie spowalniać procesu kompilacji poprawnych programów.

Dokładniejszego wyjaśnienia wymaga kwestia odzyskiwania kontroli, rozumiana jako przywrócenie wewnętrznego stanu kompilatora do stanu takiego, jakby błędu nie było, dzięki czemu możliwa staje się analiza dalszej części programu. Kwestia ta zostanie omówiona w dalszej części tego opracowania.

Kategorie błędów

Błędy można podzielić na następujące kategorie:

1) Błędy leksykalne - na wejściu pojawia się ciąg znaków nie będący prawidłowym tokenem. Obsługa tego rodzaju błędów jest prosta. Wykrywane są natychmiast w miejscu, w którym powstały. Łatwo również wypisać zrozumiały i precyzyjny komunikat o błędzie.

2) Błędy składniowe - np. nieprawidłowe wyrażenie, brakujący lub nadmiarowy średnik (lub inny separator), nie sparowane nawiasy. Niektóre z błędów składniowych mogą być wykrywane i sygnalizowane natychmiast po powstaniu, natomiast inne np. brakujący nawias zamykający '}' mogą być wykryte znacznie później (objaw błędu występuje w znacznym oddaleniu od jego rzeczywistej przyczyny). Tego rodzaju późno wykryte błędy znacznie utrudniają wspomniane wcześniej odzyskiwanie kontroli.

3) Błędy semantyczne - np. niezgodność typów lub niezadeklarowany identyfikator. Warto zauważyć, że błąd niezadeklarowanego identyfikatora może być spowodowany rzeczywistym brakiem deklaracji, prostą literówką (w miejscu użycia nieprawidłowo zapisanego identyfikatora lub w miejscu deklaracji), albo np. wcześniejszym błędem składniowym w deklaracji i w konsekwencji jej zignorowaniem przez kompilator.

4) Błędy logiczne - np. nieskończony ciąg wywołań rekurencyjnych - w większości przypadków są poza zasięgiem możliwości diagnostycznych kompilatorów i w związku z tym nie są wykrywane.

Pomimo tego, że powyższe kategorie błędów związane są z różnymi fazami kompilacji, obsługa błędów zwykle umieszczona jest w obrębie analizy składniowej.

Odzyskiwanie kontroli po wystąpieniu błędu

Odzyskiwanie kontroli po wystąpieniu błędu nazywane jest również wydobywaniem się z błędów. Proces ten jest konieczny, ponieważ w przeciwnym razie kompilacja musiałaby być przerywana po pierwszym błędzie, a to byłoby bardzo uciążliwe (każdy kto miał nieprzyjemność używania popularnego kiedyś kompilatora Pascala firmy Borland miał okazję przekonać się o tym osobiście). Niestety odzyskiwanie kontroli nie jest prostym procesem, a jego niepowodzenie może spowodować tak zwaną lawinę błędów, czyli zalew komunikatów o pozornych, w rzeczywistości nieistniejących błędach co jest spowodowane niewłaściwą interpretacją przez kompilator dalszego ciągu programu. Aby tego uniknąć, zwykle blokuje się komunikaty o błędach do chwili dopasowania co najmniej kilku symboli wejściowych do poprawnych konstrukcji językowych, co przyjmuje się jako oznakę odzyskania kontroli. Jeśli się to nie powiedzie, kompilator powinien przerwać analizę.

Można wyróżnić następujące strategie odzyskiwania kontroli po wystąpieniu błędu

1) Tryb paniki
Jest to najprostsza, ale wbrew nazwie często całkiem dobra metoda odzyskiwania kontroli. Idea polega na tym, że po zauważeniu błędu w jakiejś konstrukcji językowej staramy się pominąć całą tą błędną konstrukcję i wznowić analizę od konstrukcji następnej w nadziei, że jest ona poprawna. Szczegóły różnią się w zależności od metody analizy, ale zawsze polegają na pomijaniu symboli z wejścia aż do napotkania odpowiedniego symbolu synchronizacyjnego oraz, być może, na modyfikacji (usuwaniu) symboli na stosie analizatora. Synchronizację w tym trybie przeprowadza się dla symboli (nieterminali) o oczywistym znaczeniu np. instrukcja, wyrażenie czy blok. Jako symbole synchronizacyjne zatrzymujące pomijanie elementów z wejścia przydają się elementy zbiorów Follow. W analizatorach LR coś takiego można uznać za odpowiednik operacji redukcji dla niepoprawnego wejścia. Ważną cechę tej metody jest to, że nigdy nie powoduje ona "zapętlenia" się procesu obsługi błędów.

2) Korekcja na poziomie frazy
Podstawą tej metody jest spostrzeżenie, że większość błędów to błędy "typowe". Np. dla Pascala to zgubiony lub nadmiarowy średnik, brak ':' w przypisaniu, albo zastąpienie średnika przecinkiem na liście parametrów (widać, że dla każdego języka taka lista jest inna np. dla C/C++ z powyższej zostanie jedynie brak średnika). Idea jest taka, że kompilator lokalnie modyfikuje wejście, aby stało się ono poprawne (np. dodając brakujący średnik). Praktyczne realizacja tej metody może polegać na modyfikacji tablic sterujących działaniem analizatora składniowego. Przypominam, że tworzone są one na podstawie gramatyki i niektóre z ich pól (zwykle wiele) mogą być puste. Właśnie odwołania do tych pustych pól oznaczają błędy. Korekcja na poziomie frazy może polegać na umieszczeniu w tych pustych polach odwołań do procedur naprawiających dany rodzaj błędu. Należy przy tym uważać, aby proces naprawy błędu nie zapętlił się (co mogłoby się zdarzyć, gdyby kompilator ciągle twierdził, że czegoś brakuje i to dodawał).

3) Produkcje dla błędów
To inny sposób radzenia sobie z typowymi błędami. Po prostu dodajemy do gramatyki produkcje opisujące owe typowe błędy. Akcją związaną z tymi produkcjami jest wypisanie odpowiedniego komunikatu. Zaletą tej metody jest to, że w ogóle nie ma potrzeby wydobywać się z tak opisanego błędu (bo formalnie przecież wszystko jest zgodne z gramatyką). Wadą jest natomiast to, że zmodyfikowana gramatyka może mieć gorsze

własności (w szczególności może stać się niejednoznaczna). No i oczywiście nie wszystkie błędy da się "wprowadzić do gramatyki", z pozostałymi trzeba sobie radzić w inny sposób.

4) Korekcja globalna
To raczej rozwiązanie teoretyczne.
Znane są algorytmy, które dla danej gramatyki G i danego niepoprawnego wejścia x znajdują takie y możliwe do wygenerowania z G, ze transformacja x w y jest możliwie najtańsza w sensie liczby wstawień/usunięć/zamian symboli z gramatyki.
Jednak te algorytmy są zbyt kosztowne obliczeniowo, aby były stosowane w praktyce, a poza tym nie ma gwarancji że znalezione y jest tym, o co chodziło programiście, który napisał x.
Jednak elementy tego rodzaju analizy mogą być wykorzystane do korekty na poziomie frazy.

Na koniec warto wspomnieć, że kompilatory oprócz błędów sygnalizują również ostrzeżenia, ale to inny problem, który pominiemy.

9. Analiza semantyczna

Podczas analizy semantycznej badamy własności programu, których nie da się wyrazić za pomocą języków bezkontekstowych, nie mogą więc być badane metodami analizy składniowej. Obie te fazy są jednak ze sobą ściśle związane. Można powiedzieć, że podczas analizy semantycznej nadajemy znaczenie konstrukcjom rozpoznanym podczas analizy składniowej. Oczywiście pełne znaczenie (czyli obliczony wynik) można uzyskać jedynie wykonując program, jest to tak zwana **semantyka dynamiczna**.
Podczas analizy semantycznej badamy te cechy programu, które można określić przed jego wykonaniem, a jednocześnie leżą poza zasięgiem analizy składniowej, jest to tak zwana **semantyka statyczna**.

Dwa najbardziej charakterystyczne dla analizy semantycznej zagadnienia to kontrola typów (type checking) oraz zagadnienia związane z deklaracjami zmiennych. W tym drugim przypadku chodzi nie tylko o typ deklarowanej zmiennej, ale również o zakresy istnienia (scope) i widoczności (visibility) zmiennych i związany z tym problem skojarzenia wystąpienia aplikacyjnego (użycia zmiennej) z odpowiednim wystąpieniem definiującym (deklaracją) zmiennej,

Zauważmy, że to czy dane zagadnienie należy do semantyki statycznej czy dynamicznej zależy od konkretnego języka, np. w językach z dynamicznym wiązaniem zmiennych (zakresami dynamicznymi) skojarzenie wystąpienia aplikacyjnego z definiującym należy do semantyki dynamicznej (i jest badane podczas wykonania), a w językach ze statycznym wiązaniem zmiennych (zakresami statycznymi) do semantyki statycznej (i jest badane podczas analizy semantycznej).
Podobnie jest w przypadku kontroli typów - dla języków z dynamicznym systemem typów jest ona realizowana podczas wykonania, a dla języków ze statycznym systemem typów podczas analizy statycznej (czyli podczas kompilacji).

Zauważmy, że oba główne obszary związane z analizą semantyczną, czyli deklaracje (i odwołania do) zmiennych oraz kontrola typów dotyczą kontroli poprawności programów i badania dodatkowych reguł, które nie mogą być wyrażone za pomocą samej składni. Zatem niektóre drzewa struktury zbudowane podczas analizy składniowej mogą być podczas analizy semantycznej odrzucone jako nie reprezentujące poprawnych programów.
Natomiast w językach dopuszczających niejawne konwersje typów podczas analizy semantycznej drzewa struktury mogą być modyfikowane poprzez dodanie operacji konwersji, które nie mają swojego odpowiednika w kodzie źródłowym.

Często własności semantyczne programów opisuje się w postaci dodatkowych reguł semantycznych związanych z poszczególnymi produkcjami gramatyki opisującej składnię języka. Ten sposób opisu semantyki języków programowania nazywa się **semantyką sterowaną składnią** (syntax directed semantics). Natomiast formalnym narzędziem stosowanym do opisu semantyki są gramatyki atrybutywne. Przy ich pomocy drzewo struktury można uzupełnić o związane z poszczególnymi węzłami atrybuty (atrybuty te nie muszą być trzymane bezpośrednio w drzewie, np. atrybuty związane z identyfikatorami trzymane są w tablicy symboli).

W zależności od organizacji kompilatora analiza semantyczna (czyli obliczanie wspomnianych wyżej atrybutów) może stanowić odrębną fazę kompilacji (i wymagać dodatkowego przejścia drzewa struktury) albo też może być zintegrowana z tworzeniem drzewa podczas analizy składniowej. To drugie podejście charakterystyczne jest dla kompilatorów jednoprzebiegowych i możliwe jest jedynie dla języków bez odwołań w przód (czyli do jeszcze nie zdefiniowanych elementów programu).

Definicje

Dla każdego symbolu X gramatyki (terminalnego lub nieterminalnego) określamy zbiór atrybutów związanych z tym symbolem (np. typ wyrażenia, wartość stałej). Atrybut **a** dla symbolu **X** oznaczamy **X.a**. Zbiór atrybutów symbolu **X** oznaczamy **Attr(X)**.

Dla każdej produkcji p= (X_0 -> X_1 X_2 ... X_n) określamy zbiór **reguł semantycznych** (nazywanych również **regułami atrybutowania**) związanych z tą produkcją. Reguły te opisują zależności pomiędzy atrybutami różnych symboli występujących w danej produkcji. Są one postaci:

$X_i.a = f (... , X_j.b , ... , X_k.c , ...)$

gdzie $X_i.a$, $X_j.b$, $X_k.c$ są atrybutami poszczególnych symboli, a f dowolną funkcją (zwykle bardzo prostą)

Zbiór reguł semantycznych związanych z produkcją **p** oznaczamy **R(p)**.

Uwaga: Formalnie zakłada się, że różne symbole mają różne zbiory atrybutów. Nawet jeśli atrybuty różnych symboli tak samo się nazywają, to są różne. To jedynie (bardzo często stosowane) uproszczenie notacji, gdy atrybuty opisują tą samą cechę różnych symboli.

Definicja (jedna z wielu możliwych)
Gramatyką atrybutywną nazywamy trójkę uporządkowaną

GA = (G , Attr , R)

gdzie
- G jest gramatyką bezkontekstową G = (N,T,P,S)
- $Attr = \bigcup_{X \in N \cup T} Attr(X)$ jest zbiorem atrybutów
- $R = \bigcup_{p \in P} R(p)$ jest zbiorem reguł semantycznych (reguł atrybutowania)

Bez wnikania w zbędne zawiłości formalne, gramatyka atrybutywna jest porządnie zdefiniowana, jeśli dla dowolnego drzewa struktury można obliczyć wszystkie atrybuty we wszystkich węzłach tego drzewa i dla każdego takiego możliwego obliczenia (np. obliczając atrybuty w innej kolejności) wynik zawsze jest taki sam.

Od teraz zajmujemy się jedynie gramatykami atrybutywnymi porządnie zdefiniowanymi.

Dla każdego symbolu X gramatyki zbiór jego atrybutów można podzielić na dwa zbiory: zbiór atrybutów syntetyzowanych i zbiór atrybutów dziedziczonych.

Definicja: Atrybut **X.a** jest **syntetyzowany** jeśli dla pewnej produkcji

$p = (X \rightarrow \alpha)$

istnieje reguła semantyczna

$(X.a = f (...)) \in R(p)$

Zbiór atrybutów syntetyzowanych symbolu **X** oznaczany **Syn(X)**.

Definicja: Atrybut **X.a** jest **dziedziczony** jeśli dla pewnej produkcji

$q = (Y \rightarrow \beta X \gamma)$

istnieje reguła semantyczna

$(X.a = f (...)) \in R(q)$

Zbiór atrybutów dziedziczonych symbolu X oznaczany **Inh(X)**.

Uwaga 1: Dla porządnie zdefiniowanych gramatyk atrybutywnych dla każdego symbolu X zachodzi

$Syn(X) \cap Inh(X) = \emptyset$

oraz

$Syn(X) \cup Inh(X) = Attr(X)$

Oznacza to, że każdy atrybut jest albo syntetyzowany albo dziedziczony, ale żaden nie jest jednocześnie syntetyzowany i dziedziczony (chodzi oczywiście o atrybuty jednego symbolu, w przypadku jednakowo oznaczanych atrybutów różnych symboli może się zdarzyć, że dla jednego symbolu jest to atrybut syntetyzowany, a dla innego dziedziczony).

Uwaga 2: Atrybuty syntetyzowane symbolu X są obliczane na podstawie informacji "przychodzących" z poddrzewa o korzeniu w X, atrybuty dziedziczone są obliczane na podstawie informacji przychodzących z "góry drzewa" (od rodzica) i/lub od "rodzeństwa".

Uwaga 3: Formalnie dopuszcza się istnienie atrybutów syntetyzowanych dla symboli terminalnych gramatyki (liści drzewa). Oczywiście nie istnieją produkcje z terminalem po lewej stronie. Zakłada się, że wartości tych atrybutów dane są z zewnątrz (spoza drzewa). Typowy przykład to typ oraz wartość stałej dostarczane przez analizator leksykalny na podstawie reprezentacji stałej. Czasem tego rodzaju atrybuty wyróżnia się jako oddzielną klasę atrybutów wbudowanych, my jednak będziemy traktowali je jako dane z góry atrybuty syntetyzowane.

Uwaga 4: Możliwe są również atrybuty dziedziczone dla korzenia drzewa (chociaż korzeń oczywiście nie ma rodzica ani "rodzeństwa"). Przykładem mogą być ustawione z zewnątrz opcje kompilatora.

Przykłady

Przykład 1. Gramatyka deklaracji zmiennych
G = (N,T,P,S)
N = { decl, type, varlist }
T = { int, float, ident, **,** , **;** }
S = decl
Produkcje P

1)	decl -> type varlist **;**	varlist.dtype = type.dtype
2)	type -> int	type.dtype = integer
3)	type -> float	type.dtype = real
4)	$varlist_0$ -> ident **,** $varlist_1$	ident.dtype = $varlist_0$.dtype $varlist_1$.dtype = $varlist_0$.dtype
5)	varlist -> ident	ident.dtype = varlist.dtype

Symbole nieterminalne type i varlist oraz terminal ident mają atrybut dtype. Dla symbolu type jest on syntetyzowany, a dla varlist i ident dziedziczony. Pozostałe symbole nie mają atrybutów.
Zauważmy, że dla produkcji 4, w której symbol varlist występuje dwukrotnie, konieczne było rozróżnienie tych wystąpień (wystąpienie po lewej stronie ma zawsze indeks 0, a wystąpienia po prawej stronie kolejne numery począwszy od 1)

Przykład 2. Gramatyka wyrażeń - typy.
G = (N,T,P,S)
N = { exp, term, factor, addop, mulop }
T = { **+** , - , ***** , / , ident , int_const , FP_const, **(** , **)** }
S = decl

Produkcje P

exp_0 -> exp_1 addop term	exp_0.type = (exp_1.type == real || term.type==real) ? real : integer
exp -> term	exp.type = term.type
addop -> **+**	// nie ma dołączonej reguły dotyczącej atrybutu type
addop -> -	// nie ma dołączonej reguły dotyczącej atrybutu type
$term_0$ -> $term_1$ mulop factor	$term_0$.type = ($term_1$.type == real || factor.type==real) ? real : integer
term -> factor	term.type = factor.type
mulop -> *****	// nie ma dołączonej reguły dotyczącej atrybutu type
mulop -> /	// nie ma dołączonej reguły dotyczącej atrybutu type
factor -> ident	factor.type = informacja o ident z tablicy symboli
factor -> int_const	factor.type = integer
factor -> FP_const	factor.type = real
factor -> **(** exp **)**	factor.type = exp.type

Zauważmy, że nie ze wszystkimi produkcjami gramatyki muszą być skojarzone jakieś reguły atrybutowania.

Przedstawione wyżej konstrukcje warunkowe to bardzo częsta sytuacja w regułach atrybutowania.

Atrybut type jest w tej gramatyce atrybutem syntetyzowanym. W przypadku terminali int_const (stała całkowitoliczbowa) i FP_const (stała zmiennopozycyjna) jest on dostarczany przez analizator leksykalny, a w przypadku terminala ident jest pobierany z tablicy symboli (jest ona dokładniej opisana w dalszej części opracowania).

Obliczanie atrybutów

Zależności pomiędzy atrybutami można nałożyć na drzewo struktury i otrzymamy **graf zależności** atrybutów, jest to oczywiście graf skierowany.

Def. **Gramatyka atrybutywna jest acykliczna** jeśli dla dowolnego drzewa struktury wygenerowanego z tej gramatyki, związany z nim graf zależności atrybutów jest acykliczny.

Dalej będziemy zajmować się jedynie acyklicznymi gramatykami atrybutywnymi, możliwość pojawienia się cyklicznych zależności pomiędzy atrybutami oznacza po prostu błąd w regułach atrybutowania.

Istnieje ogólny algorytm obliczania atrybutów dla gramatyk acyklicznych. Po prostu graf zależności atrybutów należy posortować topologicznie (jest to możliwe, ponieważ jest to graf skierowany bez cykli) i obliczać atrybuty zgodnie z tym porządkiem topologicznym.
Tak naprawdę jest to odpowiednik rozwiązywania układu równań (reguły atrybutowania tworzą przecież układ równań) metodą eliminacji zmiennych.
W omawianej metodzie kompilator sam wyznacza kolejność obliczania atrybutów na podstawie drzewa struktury i dla każdego programu kolejność ta może być nieco inna.
Jest to tak zwana metoda **parse tree method**.
Jest ona niestety bardzo powolna i w związku z tym nie jest stosowana.

W praktyce stosowane są metody, w których to twórca kompilatora określa kolejność obliczania atrybutów i kolejność ta jest jednakowa dla wszystkich drzew struktury (czyli wszystkich programów). Takie metody obliczania atrybutów określane są jako **rule-based methods**.
Oczywiście metody te mogą być stosowane jedynie dla węższej klasy gramatyk atrybutywnych nazywanych gramatykami wieloprzebiegowymi (istnieje dość rozbudowana teoria dotycząca gramatyk wieloprzebiegowych, jednak w tym opracowaniu zostanie ona pominięta).

W praktyce ważną klasę stanowią gramatyki atrybutywne umożliwiające obliczanie atrybutów na bieżąco podczas analizy składniowej (parser-driven attribute evaluation). Technika ta jest szczególnie użyteczna dla pisanych ręcznie parserów metodą zejść rekurencyjnych. Atrybuty syntetyzowane mogą być po prostu zwracane jako wyniki funkcji odpowiadających symbolom gramatyki, a atrybuty dziedziczone mogą być przekazywane jako parametry tych funkcji (atrybuty pochodzące od rodzica) lub pamiętane w zmiennych lokalnych funkcji (atrybuty pochodzące od "rodzeństwa").
Natomiast dla parserów LL(1) i LR(1) posiadających jawny stos służący do przetwarzania produkcji gramatyki (parsing stack) zwykle tworzony jest drugi równoległy stos zawierający atrybuty (tak zwany value stack). W praktyce oba stosy mogą być implementowane łącznie jako jeden.

Szczegóły są oczywiście zależne od zastosowanej metody analizy składniowej, ale ogólne cechy gramatyk umożliwiające obliczanie atrybutów na bieżąco podczas analizy składniowej są wspólne dla wszystkich metod analizy.

Zauważmy, że nie ma żadnych problemów z obliczaniem podczas parsowania atrybutów syntetyzowanych. Po przetworzeniu całego poddrzewa o korzeniu w danym węźle następuje zawsze powrót do tego węzła i wtedy można obliczyć atrybuty syntetyzowane. Zatem gramatyki atrybutywne zawierające jedynie atrybuty syntetyzowane nadają się do przetwarzania na bieżąco (on the fly).

Def. Gramatykę atrybutywną nazywamy **S-atrybutywną** wtedy i tylko wtedy, gdy wszystkie jej symbole mają jedynie atrybuty syntetyzowane.

Warunek, aby gramatyka była S-atrybutywna, jest oczywiście wystarczający, aby atrybuty można było obliczać na bieżąco, ale nie jest to warunek konieczny. Dozwolone są również atrybuty dziedziczone, ale z pewnymi ograniczeniami. Ograniczenia te są następujące.

Gramatykę atrybutywną nazywamy **L-atrybutywną** wtedy i tylko wtedy gdy dla dowolnej produkcji

$p = (X_0 \rightarrow X_1 X_2 \ldots X_n)$

dla każdego $a \in Inh(X_i)$, $i = 1, 2, \ldots , n$

$a = f(\ldots , b , \ldots)$

gdzie $b \in Inh(X_0) \cup \bigcup_{j=1}^{i-1} Attr(X_j)$

Czyli wszelkie atrybuty dziedziczone symbolu X_i zależą jedynie od atrybutów wcześniejszych symboli z prawej strony produkcji (możemy je nazwać "starszym rodzeństwem") i od atrybutów dziedziczonych symbolu z lewej strony produkcji (rodzica). Patrząc z drugiej strony atrybuty dziedziczone symbolu X_i nie mogą zależeć od atrybutów późniejszych symboli z prawej strony produkcji (czyli od "młodszego rodzeństwa") ani od atrybutów syntetyzowanych symbolu z lewej strony produkcji (bo one mogą zależeć od atrybutów wszystkich symboli z prawej strony produkcji).

Jeśli gramatyka atrybutywna jest L-atrybutywna, to atrybuty można obliczyć na bieżąco podczas analizy składniowej (w przypadku analizatorów LL(1) oczywiście gramatyka musi dodatkowo być typu LL(1), bo to nie wynika z tego że jest L-atrybutywna).

Istnieje też inny sposób na obliczanie atrybutów na bieżąco.

Twierdzenie. Dowolna gramatyka atrybutywna może być przekształcona bez zmiany generowanego przez nią języka, w taki sposób, aby wszystkie atrybuty dziedziczone jej nieterminali zostały zastąpione przez atrybuty syntetyzowane.

Uwaga 1: Atrybuty dziedziczone terminali nie przeszkadzają w obliczeniach atrybutów w locie.
Uwaga 2: Wspomniane wyżej przekształcenie gramatyki zwykle sprawia, że jest ona całkowicie nieintuicyjna i w związku z tym jest rzadko stosowane.

10. Tablica symboli

Tablica symboli jest, obok drzewa struktury, drugą główną strukturą danych kompilatora. Zawiera ona informacje o wszystkich nazwach (identyfikatorach) występujących w programie. Jest wykorzystywana głównie podczas analizy semantycznej oraz generowania kodu, ale tworzona jest wcześniej już na etapie analizy leksykalnej (bo przecież to jedyny etap kompilacji operujący na kodzie źródłowym, a tam są wspomniane identyfikatory) oraz składniowej (wtedy okazuje się, czy identyfikator to nazwa zmiennej, funkcji, typu, czy jeszcze czegoś innego).

Tablica symboli jako struktura danych musi umożliwiać wykonanie następujących operacji:

1) Wstawienie symbolu - wykonywane w chwili odczytania identyfikatora w kodzie źródłowym (a w zasadzie gdy analizator składniowy rozpozna konstrukcję deklarującą identyfikator czyli deklarację zmiennej, nagłówek funkcji, deklarację typu),

2) Wyszukiwanie symbolu - wykonywane przy odwołaniach do identyfikatorów podczas analizy semantycznej i generowania kodu w celu pobrania atrybutów,

3) Usunięcie symbolu - wykonywane, gdy obiekt oznaczony identyfikatorem przestaje być dostępny, zwykle przy opuszczeniu bloku, w którym był zadeklarowany (uwaga w kompilatorach wieloprzebiegowych nie można usuwać symboli, jeśli mogą być potrzebne w kolejnym przebiegu).

W tablicy symboli mogą być przechowywane różnego rodzaju atrybuty
- nazwa
- typ
- zakres dostępności (może być "przechowywany" niejawnie w operacjach dostępu)
- lokalizacja w pamięci i rozmiar zajmowanej pamięci
- inne niezbędne dla danego rodzaju symbolu informacje (np. dla funkcji informacje o parametrach formalnych)

Jak widać z powyższej listy operacji tablica symboli spełnia definicję struktury danych nazywanej słownikiem. I rzeczywiście do implementacji tablicy symboli można użyć struktur danych używanych jako słowniki, najczęściej są to tablice haszowane, ale mogą też być drzewa zrównoważone.

W wielu językach programowania nazwy można kojarzyć z obiektami różnych rodzajów. Najważniejsze grupy to:
- stałe
- zmienne
- procedury/funkcje
- typy

W zależności od specyfiki języka (wymagań dotyczących unikalności nazw) informacje o obiektach różnych rodzajów mogą być trzymane w odrębnych tablicach symboli lub w jednej wspólnej (a wówczas informacja o rodzaju obiektu jest jednym z atrybutów)

Omówienia wymaga jeszcze wpływ struktury blokowej języka na tablicę symboli.
Możliwe są dwa rozwiązania

A) Wspólna tablica symboli dla wszystkich zakresów.
To rozwiązanie wydaje się być gorsze. Przy wszelkim dostępie do symbolu musimy podać nie tylko jego nazwę, ale również interesujący nas zakres (który musi być pamiętany jako atrybut symbolu). To znacznie komplikuje operacje na tablicy symboli. Zwróćmy też uwagę, że jeśli nie znaleźliśmy symbolu w jakimś zakresie, to zwykle należy poszukać go w zakresie zewnętrznym co jeszcze bardziej komplikuje operacje.

B) Oddzielna tablica symboli dla każdego zakresu.
To rozwiązanie wygląda lepiej. Wymaga jednak dodatkowych operacji.

4) Otworzenie nowego zakresu - utworzenie pustej tablicy symboli dla tego zakresu,

5) Zamknięcie zakresu - ta operacja to raczej ułatwienie, likwidacja tablicy symboli dla danego zakresu automatycznie usuwa wszystkie symbole z tego zakresu, zatem operacja usunięcia pojedynczego symbolu staje się niepotrzebna,

Dla kompilatorów jednoprzebiegowych, w których po przetworzeniu danego zakresu związaną z tym zakresem tablicę symboli można rzeczywiście całkowicie zlikwidować, naturalna jest organizacja ogólnej tablicy symboli w postaci stosu tablic symboli dla poszczególnych zakresów. Operacje otwarcia i zamknięcia zakresu to po prostu operacje Push i Pop, a przy wyszukiwaniu symbolu przeglądamy kolejne elementy stosu tablic symboli począwszy od szczytowego (wstawiamy zawsze do szczytowego).

Dla kompilatorów wieloprzebiegowych wymagających wielokrotnego przetwarzania każdego zakresu nie można usuwać tablicy symboli związanej z opuszczanym zakresem ponieważ będzie ona potrzebna w kolejnym przebiegu. Powstaje wówczas struktura danych nazywana **stosem kaktusowym**.

11. Generowanie kodu

Generowanie kodu, czyli faza syntezy, nie ma tak mocnych podstaw teoretycznych jak faza analizy. Jest tak dlatego, że generowanie kodu jest silnie związane z maszyną docelową i musi uwzględniać szczegóły jej architektury, a to jest trudne do teoretycznego opisu. Właśnie w związku ze stopniem skomplikowania maszyny rzeczywistej, i co za tym idzie rzeczywistego kodu maszynowego, wprowadza się pojęcie kodu pośredniego, a sam proces generowania kodu jest rozbijany na dwa etapy: generowanie kodu pośredniego i generowanie kodu wynikowego.

Pierwotnie za ideą powstania kodu pośredniego stała możliwości pominięcia pewnych szczegółów maszyny rzeczywistej, takich jak np. liczba rejestrów (i ograniczenia ich użycia), sposób adresowania pamięci, dynamiczna alokacja pamięci, dostęp do plików dyskowych i ogólnie dostęp do urządzeń zewnętrznych, czyli ogólnie kod pośredni jest po prostu kodem na wyższym poziomie abstrakcji niż kod rzeczywisty, co upraszcza jego generowanie, a także upraszcza wiele optymalizacji.

Opisany powyżej proces kompilacji z wyraźnie oddzielonymi etapami generowania kodu pośredniego i wynikowego to model klasyczny.
Obecnie, w związku z upowszechnieniem się idei maszyn wirtualnych (najbardziej znane to maszyna wirtualna Javy i maszyna wirtualna .NET) coraz częściej kompilatory generują, zamiast rzeczywistych rozkazów maszynowych, rozkazy owych maszyn wirtualnych. Ideą powstania maszyn wirtualnych również była abstrakcja od szczegółów maszyn rzeczywistych, zatem zestaw rozkazów owych maszyn wirtualnych można traktować jak kod pośredni. Można więc przyjąć, że obecnie kompilatory coraz częściej kończą pracę na etapie generowania kodu pośredniego, tłumaczenie go na rozkazy maszyny rzeczywistej pozostawiając środowisku wykonawczemu.
I takie właśnie podejście będzie realizowane na tym wykładzie.

Powszechne są dwie główne idee kodu pośredniego.
Główna różnica koncepcyjna między nimi polega na różnym podejściu do problemu ograniczonej liczby rejestrów w maszynach rzeczywistych.

Pierwszy pomysł polega na przyjęciu, że w kodzie pośrednim rejestrów jest dowolnie dużo czyli nigdy ich nie zabraknie. Jest to koncepcja tak zwanego kodu trójadresowego. Opiera się ona na tym, że typowa operacja podstawowa (np. dodawanie) ma dwa argumenty oraz wynik i każdy z tych elementów opisany jest adresem, a tym adresem jest właśnie rejestr (a ściślej pseudorejestr) kodu pośredniego, czyli kod operacji zawiera trzy adresy. Obecnie idea ta jest realizowana w praktyce w projekcie LLVM.

Drugi pomysł polega na całkowitej rezygnacji z rejestrów. Dzięki temu eliminujemy problem "rejestry były, ale się skończyły, co robić?". Całkowitą rezygnację z rejestrów osiąga się przez wprowadzenie architektury stosowej - każda operacja pobiera swoje argumenty ze stosu (i je z niego usuwa), a następnie umieszcza na stosie wynik. Obecnie najbardziej znane realizacje tej idei to maszyny wirtualne Javy i .NET, a pierwszą realizacją była P-maszyna, czyli maszyna wirtualna Pascal-a, stworzona jeszcze w latach 70-tych ubiegłego wieku.

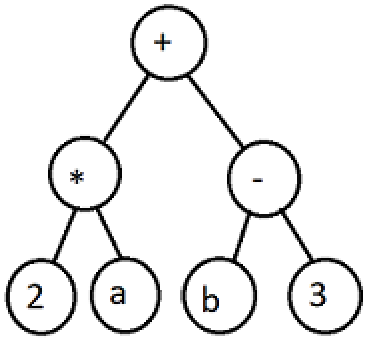
W tym opracowaniu jako przykład pierwszego podejścia (dowolnie dużo rejestrów, kod trójadresowy) będzie służył kod LLVM, natomiast jako przykład drugiego podejścia (maszyna stosowa) będzie służył kod CIL (kod maszyny wirtualnej .NET).
Ani kod LLVM, ani kod CIL, nie będzie jednak szczegółowo omawiany (zainteresowani mogą zapoznać się ze źródłami wskazanymi w bibliografii) - podane będą jedynie proste przykłady.

Przykład

Rozważmy wyrażenie (zmienne a i b to 32-bitowe zmienne całkowitoliczbowe).

2*a + (b-3)

Wyrażeniu temu odpowiada drzewo struktury



Klasyczny kod trójadresowy był w założeniu przyjazny dla czytelnika (za to niepraktyczny do rzeczywistego użycia w kompilatorze). W takiej klasycznej formie kodu trójadresowego kod powyższego wyrażenia jest następujący:

```
t1 = 2 * a
t2 = b - 3
t3 = t1 + t2
```

Każdy rozkaz kodu trójadresowego wykonuje jedną prostą operację.
Zmienne t1, t2, t3 reprezentujące obliczane wartości odpowiadają właśnie pseudorejestrom (i można takich zmiennych zdefiniować tyle, ile potrzeba).

Kod LLVM będący formalizacją koncepcji kodu trójadresowego nie jest już tak przyjazny, za to jest podstawą wielu jak najbardziej prawdziwych projektów informatycznych. Kod LLVM dla powyższego wyrażenia jest następujący:

```
%t1 = load i32, i32* %a
%t2 = mul i32 2, %t1
%t3 = load i32, i32* %b
%t4 = sub i32 %t3, 3
%t5 = add i32 %t2, %t4
```

Zgodnie z zapowiedzią pełny opis kodu LLVM nie będzie prezentowany, jednak warto wspomnieć o kilku najważniejszych aspektach.
- zmienne nie mogą być bezpośrednio argumentami rozkazów obliczeniowych, przechowywane nich dane trzeba najpierw załadować do rejestrów (rozkaz load)
- operacje są oznaczane mnemonikami, a nie symbolami (mul, sub, add)
- "wszędzie" są informacje o typach (tu i32) - co jest uciążliwe dla czytelnika, ale wygodne dla kompilatora

Natomiast kod CIL powyższego wyrażenia jest następujący:

```
ldc.i4 2
ldloc a
mul
ldloc b
ldc.i4 3
sub
add
```

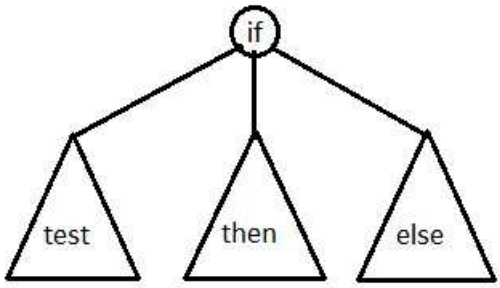
Ten kod również nie będzie szczegółowo omawiany, należy jednak wspomnieć o tym, że
- rozkaz ldc to ładowanie na stos stałej (.i4 oznacza, że ta stała to 4 bajtowa liczba całkowita)
- ldloc to załadowanie na stos wartości wskazanej zmiennej
- mul, sub, add to operacje arytmetyczne - pobierają (i usuwają) ze stosu dwa argumenty, a na stosie umieszczają wynik obliczeń.

Instrukcja warunkowa if

Przyjmijmy, że instrukcja warunkowa if opisana jest przez następującą produkcję:

ifstmt -> if (exp) stmt | if (exp) stmt else stmt

Takiej instrukcji warunkowej odpowiada następujący fragment drzewa struktury:



Węzłowi if mogłaby odpowiadać klasa (Statement i Expression to odpowiednio klasy odpowiadające ogólnym pojęciom instrukcji i wyrażenia)

```
class IfStmt : Statement
    {
    Expression test;
    Statement thenStmt;
    Statement elseStmt;
    ....
    }
```

W kodzie pośrednim tą strukturę drzewiastą należy zlinearyzować. Robi się to przy pomocy rozkazów skoków (warunkowych i bezwarunkowych).
Szczegóły różnią się dla kodu LLVM i CIL.

Kod LLVM

Instrukcji warunkowej if odpowiada następujący schemat kodu LLVM

```
[kod obliczający warunek do t]
br i1 %t, label %truelab, label %falselab
truelab:
[kod sekcji then]
br label %endlab
falselab:
[kod sekcji else]
br label %endlab
endlab:
```

Kod odpowiadający temu schematowi może być wygenerowany przez następującą funkcję

```
string GenCode()
    {
    string t, truelab, falselab, endlab;
    t = test.GenCode();
    truelab  = NewTemp();
    falselab = NewTemp();
    EmitCode($"br i1 %{t}, label %{truelab}, label %{falselab}");
    EmitCode($"{truelab}:");
    thenStmt.GenCode();
    endlab  = NewTemp();
    EmitCode($"br label %{endlab}");
    EmitCode($"{falselab}:");
    if ( elseStmt!=null )
        elseStmt.GenCode();
    EmitCode($"br label %{endlab}");
    EmitCode($"{endlab}:");
    return null;
    }
```

Poniżej przedstawione jest kilka uwag wyjaśniających najważniejsze (i nieoczywiste) elementy powyższego kodu.

Uwaga 1:
Funkcja GenCode formalnie zwraca string, ponieważ technicznie zapewne byłaby to funkcja wirtualna zdefiniowana dla każdego z typów węzłów w drzewie struktury i oczywiście dla wszystkich tych węzłów musi mieć taką samą sygnaturę. Dla węzłów odpowiadających operacjom zwracającym wartość funkcja GenCode zwraca nazwę zmiennej roboczej (pseudorejestru) zawierającej wynik obliczeń (widać to w wywołaniu funkcji GenCode dla warunku instrukcji if), instrukcja if nie zwraca wartości zatem funkcja GenCode zwraca null.

Uwaga 2:
Funkcja NewTemp zwraca unikalny identyfikator, jest wykorzystywana do generowania etykiet i nazw zmiennych roboczych (pseudorejestrów).

Uwaga 3:
Funkcja EmitCode generuje pojedynczą operację kodu pośredniego, nie należy jej mylić z funkcją GenCode, która generuje kod dla całej konstrukcji języka wysokiego poziomu (a ten kod zazwyczaj składa się z wielu operacji kodu pośredniego).

Uwaga 4:
W przypadku instrukcji bez frazy else referencja na odpowiednie poddrzewo jest równa null i kod sekcji else nie jest generowany.

Uwaga 5:
Rozkaz skoku do etykiety znajdującej się w następnej linii (dwie ostatnie linie schematu kodu) jest niezbędny. To specyficzna cecha kodu LLVM - do etykiety można jedynie przejść za pomocą rozkazu skoku, nie można po prostu przejść jako do kolejnej tekstowo instrukcji.

Kod CIL

Instrukcji warunkowej if odpowiada następujący schemat kodu CIL

```
[kod obliczający warunek]
brfalse falselab
[kod sekcji then]
br endlab
falselab:
[kod sekcji else]
endlab:
```

Kod odpowiadający temu schematowi może być wygenerowany przez następującą funkcję

```
void GenCode()
    {
    string falselab, endlab;
    test.GenCode();
    falselab = NewTemp();
    EmitCode($"brfalse {falselab}");
    thenStmt.GenCode();
    endlab  = NewTemp();
    EmitCode($"br {endlab}");
    EmitCode($"{falselab}:");
    if ( elseStmt!=null )
        elseStmt.GenCode();
    EmitCode($"{endlab}:");
    }
```

Poniżej przedstawione jest kilka uwag wyjaśniających najważniejsze (i nieoczywiste) elementy powyższego kodu.

Uwaga 1:
Ponieważ w kodzie CIL wyniki obliczeń zapisywane są na stosie i nie ma konieczności korzystania ze zmiennych roboczych funkcja, GenCode formalnie nie zwraca wartości (jest typu void).

Uwaga 2:
Znaczenie funkcji NewTemp i EmitCode jest takie same jak dla kodu LLVM.

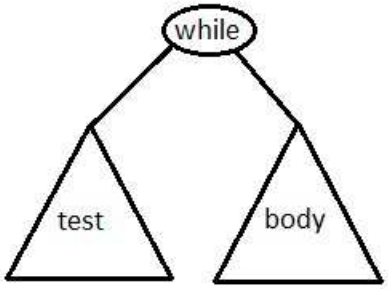
Uwaga 3:
Idea reprezentowania w drzewie oraz generowania kodu CIL dla instrukcji if bez frazy else jest identyczna jak dla kodu LLVM.

Pętla while

Przyjmijmy, że pętla while opisana jest przez następującą produkcję

loop -> while (exp) stmt

Takiej pętli odpowiada następujący fragment drzewa struktury:



Węzłowi while mogłaby odpowiadać klasa (Statement i Expression to odpowiednio klasy odpowiadające ogólnym pojęciom instrukcji i wyrażenia)

```
class While : Statement
    {
    Expression test;
    Statement body;
    ....
    }
```

Podobnie jak dla instrukcji warunkowej w kodzie pośrednim tą strukturę drzewiastą należy zlinearyzować. Szczegóły różnią się dla kodu LLVM i CIL.

Kod LLVM

Pętli while odpowiada następujący schemat kodu LLVM

```
br label %startlab
startlab:
[kod obliczający warunek do t]
br i1 %t, label %innerlab, label %endlab
innerlab:
[kod wnętrza]
br label %startlab
endlab:
```

Kod odpowiadający temu schematowi może być wygenerowany przez następującą funkcję

```
string GenCode()
    {
    string t, startlab, innerlab, endlab;
    startlab = NewTemp();
    innerlab = NewTemp();
    endlab   = NewTemp();
    EmitCode($"br label %{startlab}");
    EmitCode($"{startlab}:");
    t = test.GenCode();
    EmitCode($"br i1 %{cond}, label %{innerlab}, label %{endlab}");
    EmitCode($"{innerlab}:");
    body.GenCode();
    EmitCode($"br label %{startlab}");
    EmitCode($"{endlab}:");
    return null;
    }
```

Kod CIL

Pętli while odpowiada następujący schemat kodu CIL

```
startlab:
[kod obliczający warunek]
brfalse endlab
[kod wnętrza]
br startlab
endlab:
```

Kod odpowiadający temu schematowi może być wygenerowany przez następującą funkcję

```
void GenCode()
    {
    string startlab, endlab;
    startlab = NewTemp();
    EmitCode($"{startlab}:");
    test.GenCode();
    endlab = Compiler.NewTemp();
    EmitCode($"brfalse {endlab}");
    body.GenCode();
    EmitCode($"br {startlab}");
    EmitCode($"{endlab}:");
    }
```

Zauważmy, że kod dla pętli while zaczyna i kończy się etykietą. Dzięki temu łatwo jest zaimplementować instrukcje break (skok do etykiety endlab) i continue (skok do etykiety startlab). Uwaga ta dotyczy zarówno kodu CIL, jak i LLVM.

Obliczenia skrócone

Warto wspomnieć również o realizacji operatorów logicznych || i && jako obliczeń skróconych. Polega to na tym, że dla wyrażenia

```
log1 || log2
```

jeśli wartość wyrażenia log1 jest równa true, to wyrażenie log2 nie jest w ogóle obliczane. Analogicznie dla wyrażenia

```
log1 && log2
```

jeśli wartość wyrażenia log1 jest równa false, to wyrażenie log2 nie jest w ogóle obliczane. Oczywiście w kodzie pośrednim nie ma operacji "wykonaj obliczenia skrócone" - trzeba to oprogramować przy pomocy serii innych operacji (w tym rozkazów skoków - na niskim poziomie nie da się programować bez skoków). Tak naprawdę kod obliczeń skróconych podobny jest do kodu instrukcji warunkowej if (szczegóły pozostawiamy jako ćwiczenie domowe).

Załączniki

W załączniku do tego opracowania znajdują się dwa kompletne przykłady (można je skompilować i uruchomić) generatorów kodu dla sekwencji wyrażeń arytmetycznych z predefiniowanymi zmiennymi całkowitoliczbowymi i zmiennopozycyjnymi.
Pierwszy generator generuje kod LLVM na podstawie drzewa struktury, a do stworzenia drzewa używa metody zejść rekurencyjnych.
Drugi generator jest jednoprzebiegowy, nie tworzy jawnego drzewa struktury, korzysta z generatora parserów GPPG, kod generowany jest bezpośrednio w akcjach parsera.

12. Załączniki

Plik zalaczniki.zip zawiera kody źródłowe przykładów będących uzupełnieniem tego opracowania.
W archiwum znajdują się dwa katalogi
- kalkulator_jednoliniowy
- kalkulator_wieloliniowy

Katalog kalkulator_jednoliniowy zawiera dwie implementacje kalkulatora pojedynczych wyrażeń arytmetycznych. Obie wersje kalkulatora do analizy składniowej korzystają z metody zejść rekurencyjnych, bazują jednak na inaczej zapisanej gramatyce (ale obie wersje rozpoznają ten sam język).
W podkatalogu recursive1 znajduje się wersja kalkulatora bazująca na gramatyce zapisanej w notacji EBNF, natomiast w podkatalogu recursive2 znajduje się wersja kalkulatora bazująca na gramatyce zapisanej w klasycznej notacji, ale z wyeliminowaną lewostronną rekursją. Różne są jedynie pliki Parser.cs w obu podkatalogach, pliki Main.cs oraz Scanner.cs są identyczne.
Obie wersje kalkulatora generują takie same drzewa struktury i na ich podstawie obliczają wartość wyrażeń.

Katalog kalkulator_wieloliniowy zawiera dwie implementacje kalkulatora sekwencji wyrażeń arytmetycznych z predefiniowanymi zmiennymi. Obie wersje kalkulatora rozpoznają ten sam język, natomiast jedna z nich generuje kod LLVM, a druga kod CIL.

W podkatalogu LLVM znajduje się generator kodu LLVM. W fazie analizy składniowej wykorzystuje on metodę zejść rekurencyjnych. Najpierw tworzone jest drzewo struktury, a następnie generowany jest kod LLVM.
W podkatalogu CIL znajduje się generator kodu CIL. Korzysta on z narzędzi Gardens Point. Jest to generator jednoprzebiegowy, kod CIL generowany jest bezpośrednio w akcjach parsera (bez tworzenia jawnego drzewa struktury).

Wszystkie przykłady są to kompletne kody źródłowe, które można skompilować i uruchomić (w przypadku generatora kodu CIL niezbędne jest wykorzystanie narzędzi Gardens Point).

13. Bibliografia

1) Ullman J., Aho A.V., Sethi R., Lam M.S. - Kompilatory. Reguły, metody i narzędzia, WNT 2019
2) Waite W.M., Goos G. - Konstrukcja kompilatorów, WNT 1989
3) Louden K.C. - Compiler Construction: Principles and Practice, PWS 1997
4) ECMA-335 standard - Common Language Infrastructure (CLI), https://www.ecma-international.org/publications-and-standards/standards/ecma-335/
5) dokumentacja projektu LLVM, https://llvm.org/docs/Reference.html
6) dostępna w internecie dokumentacja narzędzi FLEX i Bison, np: https://westes.github.io/flex/manual/ https://www.gnu.org/software/bison/manual/

Fundusze Europejskie Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Politechnika Warszawska
Unia Europejska Europejski Fundusz Społeczny

Projekt „NERW 2 PW. Nauka – Edukacja – Rozwój – Współpraca” współfinansowany jest ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego.

Zadanie 10 pn. „Modyfikacja programów studiów na kierunkach prowadzonych przez Wydział Matematyki i Nauk Informacyjnych”, realizowane w ramach projektu „NERW 2 PW. Nauka – Edukacja – Rozwój – Współpraca”, współfinansowanego ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego.