

Opracowanie do egzaminu
Teoria komplikacji i kompilatory 2020

Spis treści

1 Trochę informacji teoretycznych na dobry początek...	3
1.1 Translator, kompilator, interpreter	3
1.2 Proces tworzenia kodu maszynowego - preprocesor, assembler	5
1.3 Przerywnik - zebranie powyższych pytań	7
1.4 Budowa kompilatora	8
1.4.1 Analiza leksykalna	9
1.4.2 Przerywnik - zebranie powyższych pytań	11
1.4.3 Analiza składniowa	12
1.4.4 Analiza semantyczna	14
1.4.5 Generator kodu pośredniego	15
1.4.6 Optymalizator	15
1.4.7 Generator kodu	15
1.4.8 Przerywnik - zebranie powyższych pytań	17
1.5 Rodzaje błędów i sposoby radzenia sobie z nimi	21
2 Przypomnienie z gramatyk	23
2.1 Przerywnik - zebranie powyższych pytań	26
3 Analizatory składniowe - parsery	27
3.1 Przerywnik - zebranie powyższych pytań	28
4 Rekurencja lewostronna i lewostronna faktoryzacja	29
4.1 Metoda 1 - zamiana problematycznej produkcji	29
4.2 Metoda 2 - dla dowolnej liczby produkcji rekurencyjnie lewostronnych (z wykładu)	31
4.3 Lewostronna faktoryzacja	32
5 Przerywnik - zebranie powyższych pytań	35
6 Wstęp do analizatorów składniowych LL(k)	36
7 Zbiory $FIRST(\alpha)$ oraz $FOLLOW(A)$	38
7.1 Zbiór $FIRST$	38
7.2 Zbiór $FOLLOW$	39
8 Przerywnik - zebranie powyższych pytań	40
9 Gramatyka LL(1)	41
10 Przerywnik - zebranie powyższych pytań	44
11 Budowa parsera LL(1)	45
11.1 Budowa tabelki sterującej M dla parsera LL(1)	46
11.1.1 Struktura tabeli	46
11.2 Wypełnianie tabelki sterującej	48

11.2.1 Obliczanie $FIRST(\alpha)$	49
12 Proces parsowania	52
13 Wstęp do analizatorów składniowych LR	55
13.1 Uchwyty i ich przycinanie	55
13.2 Analiza metodą przesunięcie-redukcja	57
13.3 Gramatyki dla których można stosować analizę przesunięcie-redukcja	58
14 Sytuacje $LR(0)$	60
15 Funkcja CLOSURE	61
16 Funkcja GOTO	62
17 Tabelka sterująca parsera LR	64
18 Proces parsowania LR	68

Opracowanie na podstawie:

- Alfred V. Aho, Jeffrey Ullman, Monica S. Lam, Sethi Ravi Kompilatory - Reguły, Metody, Narzędzia, PWN 2019
- Wykłady z przedmiotu Teoria kompilacji i kompilatorów - w mniejszym stopniu

1 Trochę informacji teoretycznych na dobry początek...

Wiadomości z tego rozdziału pokrywają dwa pierwsze wykłady.

1.1. Translator, kompilator, interpreter

Pytanie

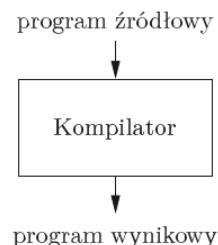
Czym jest translator ?

To program przekształcający program źródłowy na program docelowy.

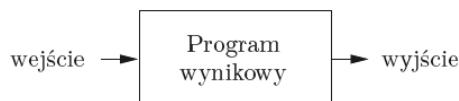
Pytanie

Czym jest kompilator ?

Mówiąc w uproszczeniu, kompilator jest programem, który potrafi przeczytać program sformułowany w jednym języku – języku źródłowym – i przełożyć go na równoważny program w innym języku – języku wynikowym (wykonywalnym). Ważną rolą kompilatora jest zgłoszenie wykrytych w czasie tłumaczenia dowolnych błędów w programie źródłowym. Można powiedzieć, że kompilator jest szczególnym przypadkiem translatora, który zgłasza błędy.



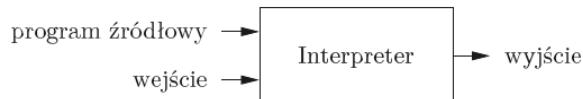
Jeśli program wynikowy jest programem wykonywalnym w języku maszynowym, może zostać uruchomiony przez użytkownika w celu przetworzeniu wejścia i wygenerowania wyjścia:



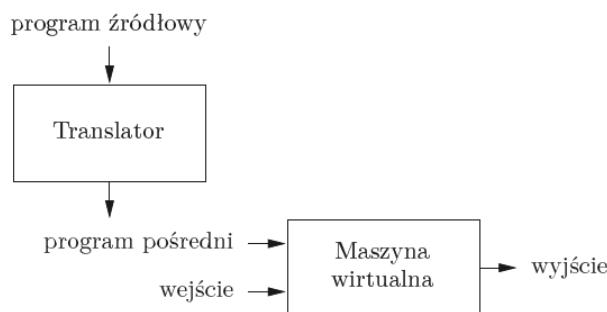
Pytanie

Czym jest interpreter ?

Innym powszechnie spotykanym rodzajem translatora jest interpreter. Zamiast tworzenia programu wynikowego jako efektu tłumaczenia, interpreter wydaje się bezpośrednio wykonywać operacje wyspecyfikowane w programie źródłowym względem danych wejściowych dostarczanych przez użytkownika.



Ciekawostka dotyczący Javy: Translatory języka Java łącza komplikacje i interpretacje, co pokazuje rysunek 1.4. Program źródłowy Java może zostać najpierw skompilowany do formy pośredniej nazywanej kodem bajtowym (bytecode). Kod bajtowy jest następnie interpretowany przez maszynę wirtualną. Zaleta tego podejścia jest to, że kod bajtowy – skompilowany na jednej maszynie – może być interpretowany na innym komputerze, być może również przez sieć.



Pytanie

Jakie są zalety interpretera wobec kompilatora i kompilatora wobec interpretera ?

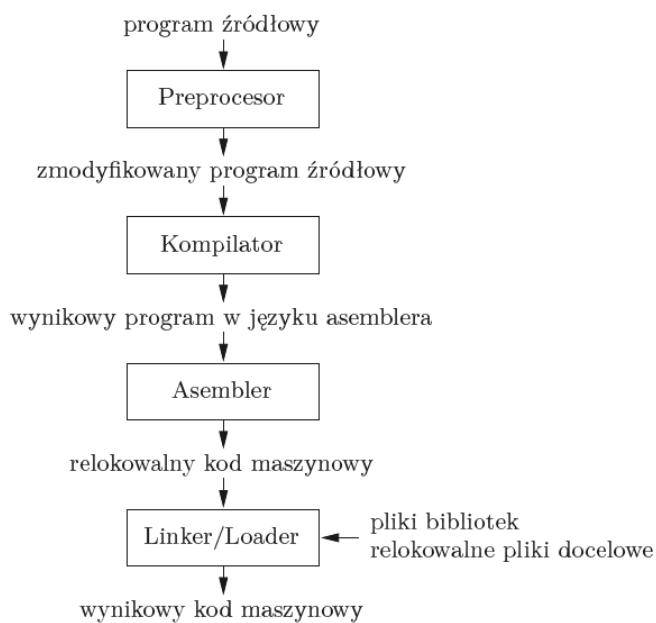
Z jednej strony program wynikowy w języku maszynowym tworzony przez kompilator jest zazwyczaj znacznie szybszy od interpretera przy przechodzeniu od wejścia do wyjścia. Z drugiej strony interpreter zazwyczaj udostępnia lepszą diagnostykę błędów niż kompilator, gdyż wykonuje program źródłowy instrukcja po instrukcji.

1.2. Proces tworzenia kodu maszynowego - preprocessor, assembler

Pytanie

Czy sam kompilator wystarczy do wytworzenia kodu maszynowego ? Jakie inne programy poza kompilatorem mogą być potrzebne do utworzenia kodu maszynowego ?

W większości przypadków sam kompilator nie wystarczy i potrzebne jest więcej programów. Translacja kodu programu bezpośrednio na kod maszynowy byłaby conajmniej trudnym przedsięwzięciem i dlatego kod jest zazwyczaj tłumaczony najpierw na assemblera. Te dodatkowe programy przedstawione są na rysunku poniżej.



- Preprocesor - odpowiada za zbieranie elementów programu źródłowego. Może on również rozwijać skróty nazywane makrami do pełnych wyrażeń języka źródłowego (tak jak np. w języku C).

Pytanie

Czy kompilator tworzy kod maszynowy ? Czym jest assembler ?

Zmodyfikowany program źródłowy jest następnie przekazywany do kompilatora. Ten może utworzyć jako swoje wyjście program w języku asemblera, gdyż tworzenie kodu asemblera jest łatwiejsze do wykonania, a ponadto łatwiejsze do debugowania.

Uzyskany kod asemblera jest następnie przetwarzany przez kolejny program nazywany po prostu assemblerem, który generuje relocowalny kod maszynowy jako swoje wyjście.

Pytanie

Jaka jest rola linkera (konsolidatora) i loadera ?

Obszerne programy są często kompilowane w częściach, zatem uzyskane fragmenty kodu maszynowego mogą wymagać złączenia z innymi relokowalnymi plikami wynikowymi i plikami bibliotek w kod, który ostatecznie może zostać uruchomiony na komputerze.

Konsolidator (nazywany również linkerem) rozwiązuje zewnętrzne adresy pamięci, dzięki czemu kod w jednym pliku może odnosić się do lokalizacji w innym pliku. Następnie program ładowający (loader) umieszcza wszystkie wykonywalne pliki obiektowe w pamięci w celu wykonania.

1.3. Przerywnik - zebranie powyższych pytań

Pytanie

Czym jest translator ?

Pytanie

Czym jest kompilator ?

Pytanie

Czym jest interpreter ?

Pytanie

Jakie są zalety interpretera wobec kompilatora i kompilatora wobec interpretera ?

Pytanie

Czy sam kompilator wystarczy do wytworzenia kodu maszynowego ? Jakie inne programy poza kompilatorem mogą być potrzebne do utworzenia kodu maszynowego ?

Pytanie

Czy kompilator tworzy kod maszynowy ? Czym jest assembler ?

Pytanie

Jaka jest rola linkera (konsolidatora) i loadera ?

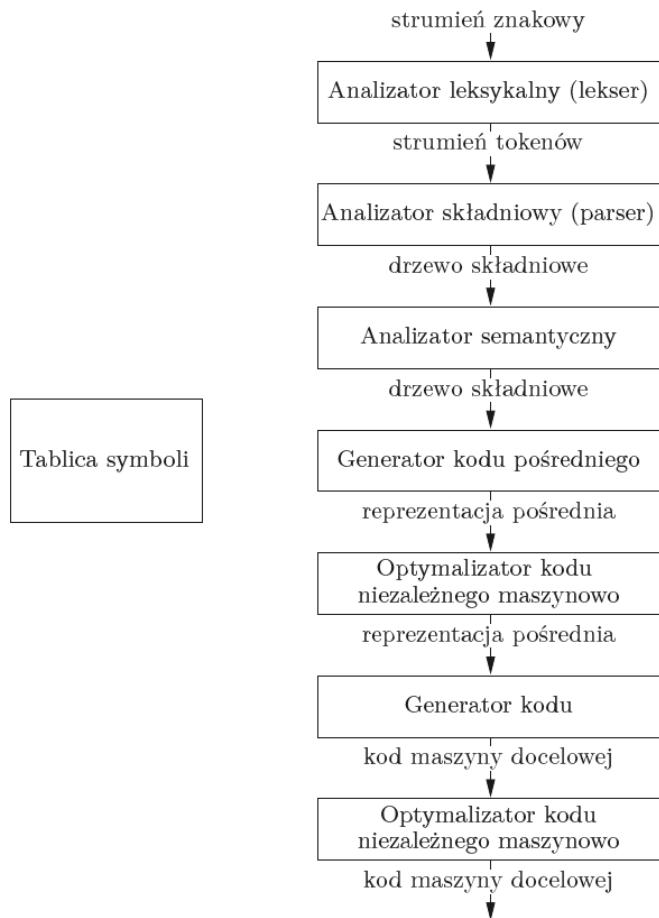
1.4. Budowa kompilatora

Teraz bardziej zagłębimy się w budowę samego kompilatora.

Pytanie

Jak ogólnie zbudowany jest kompilator ?

- Kompilator składa się z kilku modułów, z których każdy ma inne zadanie i odpowiada innej fazie komplikacji. W całym procesie reprezentacja programu zmienia się pomiędzy poszczególnymi fazami. Moduły kompilatora są niezależne od siebie.
- Wejściem całego procesu jest strumień znaków programu, a wyjściem kod maszyny docelowej. Poszczególne fazy realizowane są przez komponenty kompilatora sekwencyjnie jak na rysunku poniżej.
- Tablica symboli to struktura danych, która przechowuje informacje o programie źródłowym.
- Lekser nazywany jest również czasami skanerem.
- Niektóre z modułów są opcjonalne, np. optymalizator



Pytanie

Czym jest faza analizy, a czym faza syntezy kompilatora ?

Patrząc na te komponenty możemy wyróżnić dwie ogólne fazy działania kompilatora, są to:

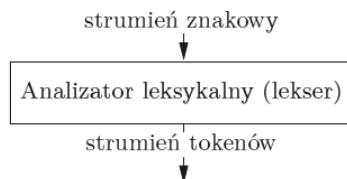
- faza analizy - za realizację tej fazy odpowiedzialne są komponenty analizatorów: analizator leksykalny czyli lexer (skaner) oraz analizator składniowy czyli parser. Część analityczna dzieli program źródłowy na części składowe i stosuje do nich strukturę gramatyczną. Następnie używa tej struktury do utworzenia pośredniej reprezentacji programu źródłowego. Jeśli część analityczna odkryje, że program źródłowy jest błędnie sformułowany pod względem składniowym lub niejednoznaczny semantycznie, musi zwrócić komunikaty informujące o tych niedociągnięciach, aby użytkownik mógł podjąć działania naprawcze. W części analitycznej gromadzone są również informacje o programie źródłowym, umieszczane w strukturze danych nazywanej tablica symboli, która jest przekazywana wraz z reprezentacją pośrednią do fazy syntezy.
- faza syntezy - jest realizowana przez pozostałe komponenty kompilatora, konstruuje pożądany program wynikowy z reprezentacji pośredniej i informacji zawartych w tablicy symboli.

1.4.1. Analiza leksykalna

Teraz omówimy sobie pierwszą fazę działania kompilatora, czyli analizę leksykalną. Komponentem kompilatora odpowiedzialnym za tę fazę jest analizator leksykalny nazywany również lekserem lub skanerem.

Pytanie

Co jest wejściem, a co wyjściem fazy analizy leksykalnej ?



Pytanie

Za co odpowiedzialny jest lekser (skaner) ?

- odczytuje strumień znaków budujących program źródłowy i grupuje te znaki w znaczące sekwencje nazywane leksemami. Dla każdego leksemu analizator leksykalny tworzy wyjście w postaci tokenu (inaczej atomu)
- usuwa znaki nie mające wpływu na sam program (np. odstępy, komentarze)

Pytanie

Czym jest leksem, a czym token ?

- Leksem to ciąg znaków, które stanowią semantycznie niepodzielną całość
- Token ma format (*nazwa-tokenu, wartość-atrybutu*)

UWAGA: Każdy leksem jest zamieniany na token.

Przykład

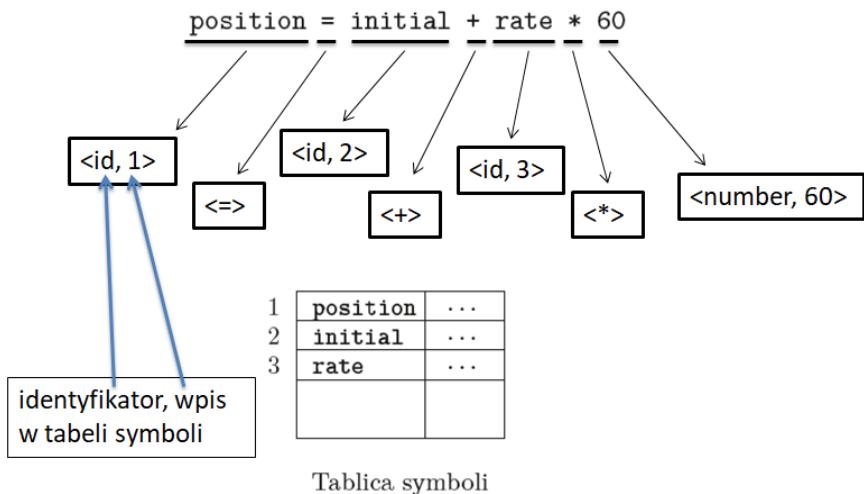
Mamy dany kod:

$$\text{position} = \text{initial} + \text{rate} * 60$$

Najpierw skaner wygeneruje 8 różnych leksemów, każdy z nich został oznaczony podkreśleniem poniżej:

position = initial + rate * 60

Następnie skaner będzie zamieniał leksemy na tokeny:



Zauważmy, że niektóre z tokenów nie mają wartości. Dla niektórych leksemów nie jest ona konieczna.

Wartość dodajemy dla leksemów, które są użyte w kontekście programu jako taka sama struktura (np. identyfikatory zmiennych), ale mogą mieć różne wartości (identyfikator może mieć różne nazwy).

1.4.2. Przerywnik - zebranie powyższych pytań

Pytanie

Jak ogólnie zbudowany jest kompilator ?

Pytanie

Czym jest faza analizy, a czym faza syntezy kompilatora ?

Pytanie

Co jest wejściem, a co wyjściem fazy analizy leksykalnej ?

Pytanie

Za co odpowiedzialny jest lekser (skaner) ?

Pytanie

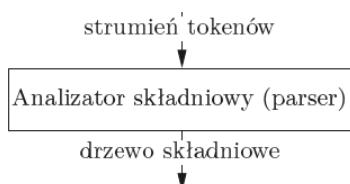
Czym jest leksem, a czym token ?

1.4.3. Analiza składniowa

Teraz omówimy sobie drugą fazę działania kompilatora, czyli analizę składniową. Komponentem kompilatora odpowiedzialnym za tą fazę jest analizator składniowy nazywany również parserem.

Pytanie

Co jest wejściem, a co wyjściem fazy analizy składniowej ?



Pytanie

Za co odpowiedzialny jest parser ?

- dokonuje analizy składniowej programu na podstawie jego postaci pośredniej otrzymanej jako wynik analizy leksykalnej
- sprawdza poprawność składniową (syntaktyczną) poprzez:
 - dokonanie rozbioru na części składowe
 - zbudowanie odpowiedniego drzewa składniowego

Pytanie

Jak można podzielić parsery ?

Parsery można podzielić na:

- analizatory syntaktyczne (składniowe) - dokonują ścisłej analizy składniowej
- analizatory semantyczne - sprawdzają pewne aspekty poprawności semantycznej, np. zastosowanie tego samego identyfikatora wiele razy

UWAGA: my przedstawiliśmy sobie analizator semantyczny jako oddzielny komponent kompilatora, ale czasami jest on uznawany za rodzaj parsera, w istocie jest to również parser, tyle, że sprawdza inne aspekty poprawności niż analizator składniowy

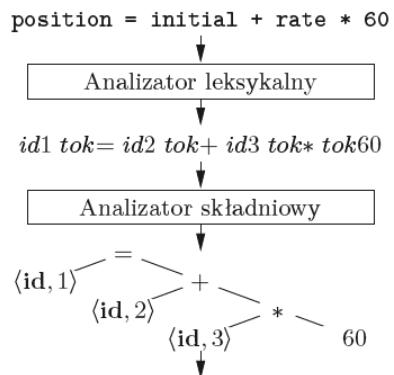
Pytanie

Od czego zależy sposób działania analizatora składniowego (syntaktycznego) ?

Sposób działania zależy w największym stopniu od tego w jaki sposób została zdefiniowana gramatyka. Analizatory składniowe będziemy konstruować w zależności od własności zdefiniowanej gramatyki. Ogólnie rzecz ujmując, proces analizy składniowej jest procesem dość złożonym.

Przykład

Drzewo składniowe dla rozważanego przez nas przykładu mogłoby mieć taką postać:



1.4.4. Analiza semantyczna

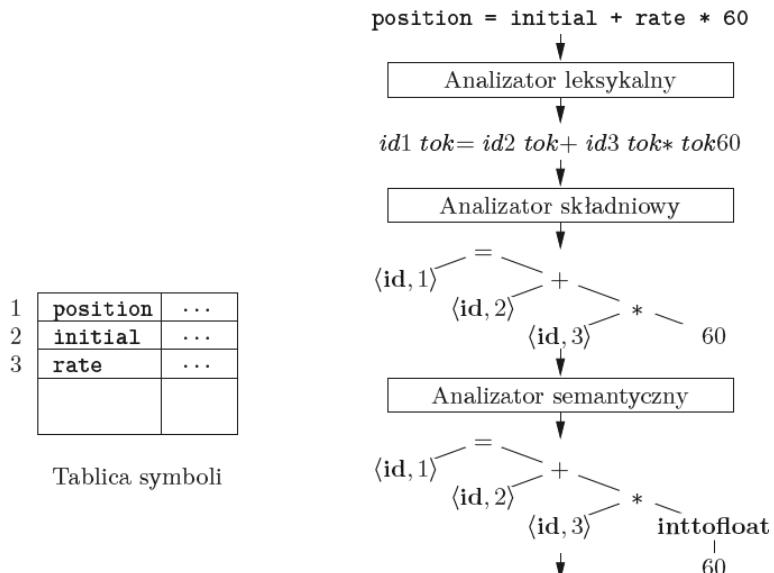
Pytanie

Za co odpowiedzialny jest analizator semantyczny ?

Analizator semantyczny wykorzystuje drzewo składniowe oraz informacje z tablicy symboli do sprawdzenia programu źródłowego pod katem spójności semantycznej programu z definicją języka. Ponadto gromadzi on informacje o typach i zapisuje je albo w drzewie składniowym, albo w tablicy symboli do późniejszego użycia podczas generowania kodu pośredniego.

Przykład

Rozwijając dalej rozważany przykład:



1.4.5. Generator kodu pośredniego

Pytanie

Za co odpowiedzialny jest generator kodu pośredniego ?

Po analizie składniowej i semantycznej programu źródłowego wiele kompilatorów generuje jawną ni-skopoziomową reprezentację pośrednią, zbliżoną do kodu maszynowego, o której możemy myśleć jako o programie dla maszyny abstrakcyjnej. Ta postać pośrednia powinna mieć dwie ważne właściwości: powinna być łatwa do utworzenia i łatwa do przetłumaczenia na kod maszyny docelowej.

1.4.6. Optymalizator

Pytanie

Za co odpowiada optymalizator kodu ?

Faza niezależnej od architektury maszynowej optymalizacji kodu ma na celu ulepszenie kodu pośredniego, dzięki czemu lepszy będzie również kod wynikowy. Zazwyczaj “lepszy” oznacza “szybszy”, ale możliwe są też inne pożądane cele, takie jak mniejsza długość lub mniejsze zużycie energii przez kod wynikowy. Optymalizator jest elementem opcjonalnym kompilatora.

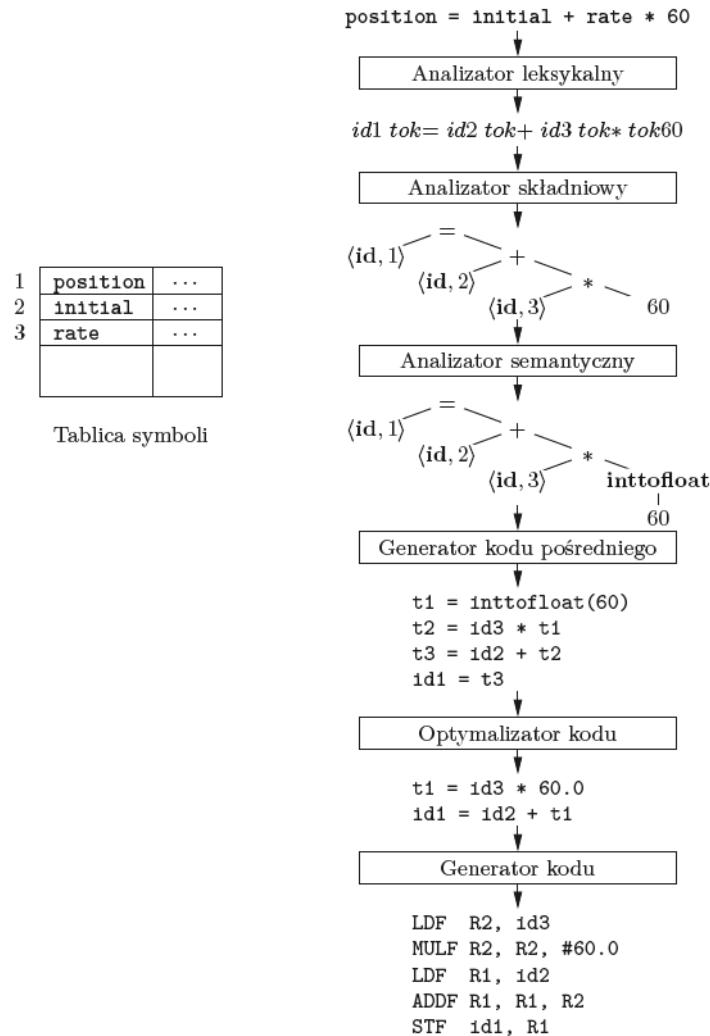
1.4.7. Generator kodu

Pytanie

Za co odpowiedzialny jest generator kodu ?

Generator kodu przyjmuje jako wejście reprezentacje pośrednią programu źródłowego i odwzorowuje ją na język wynikowy. Jeśli językiem tym jest kod maszynowy, dla każdej zmiennej używanej przez program wybierane są rejestrów lub lokalizacje w pamięci. Następnie instrukcje pośrednie są tłumaczone na sekwencje instrukcji maszynowych wykonujących to samo zadanie. Krytycznym aspektem generowania kodu jest rozważne przypisanie rejestrów do przechowywanych zmiennych.

Przykład Wracając do naszego przykładu cały proces wyglądałby tak



1.4.8. Przerywnik - zebranie powyższych pytań

Pytanie

Co jest wejściem, a co wyjściem fazy analizy składniowej ?

Pytanie

Za co odpowiedzialny jest parser ?

Pytanie

Jak można podzielić parsery ?

Pytanie

Od czego zależy sposób działania analizatora składniowego (syntaktycznego) ?

Pytanie

Za co odpowiedzialny jest analizator semantyczny ?

Pytanie

Za co odpowiedzialny jest generator kodu pośredniego ?

Pytanie

Za co odpowiada optymalizator kodu ?

Pytanie

Za co odpowiada optymalizator kodu ?

Pytanie

Za co odpowiedzialny jest generator kodu ?

Spójrzmy jeszcze na przykład z wykładu 31.03.2020:

Prosty proces komplikacji na przykładzie ciągu znaków:

$$ALFA := (BETA + GAMMA) * 2.5$$

Podział na tokeny:

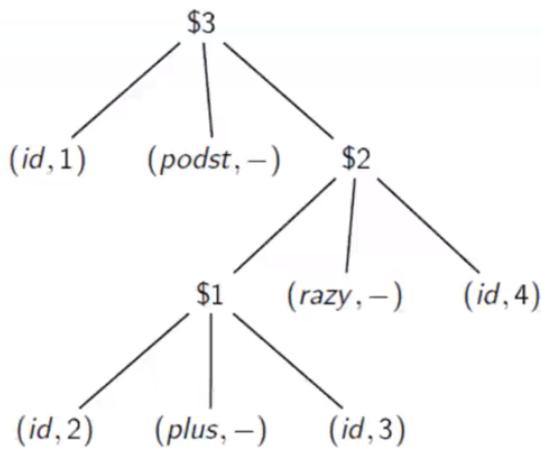
`ALFA` `:=` `(` `BETA` `+` `GAMMA` `)` `*` `2.5`

Tworzenie par oraz generowanie tablic:

- Ciąg par:
 $(id, 1)$; $(podst, -)$; $(Inawias, -)$;
 $(id, 2)$; $(plus, -)$; $(id, 3)$;
 $(pnawias, -)$; $(razy, -)$; $(id, 4)$; $(eof, -)$.

- Tablica identyfikatorów:

Lp.	Identyfikator	Typ	Adres/wartość
1	ALFA	zmienna	
2	BETA	zmienna	
3	GAMMA	zmienna	
4	brak	stała	2.5



Drzewo syntaktyczne pewnego wyrażenia.

Przykład c.d.

Na tym etapie mamy drzewo syntaktyczne. Teraz chcemy dokonać translacji na kod assemblera.

LOAD — pobranie do akumulatora zawartości wskazanej komórki;

STOR — zapamiętanie we wskazanej komórce pamięci zawartości akumulatora;

ADD — dodanie do zawartości akumulatora wartości wskazanej komórki pamięci, wynik umieszczany w akumulatorze;

MPY — mnożenie zawartości akumulatora przez zawartość wskazanej komórki pamięci, wynik umieszczany w akumulatorze.

Wierzchołek typu podstawienie („:=“)

Generujemy kolejno:

- jeśli prawy wierzchołek potomny jest liściem,
to wtedy " LOAD prawy liść ",
w p.przypadku kod wygenerowany dla prawego wierzchołka
potomnego;
- dla lewego wierzchołka potomnego " STOR lewy liść ".

Notacja symboliczna (assemblerowa)

- Węzeł \$1

LOAD GAMMA

STOR \$1

LOAD BETA

ADD \$1

Notacja symboliczna (assemblerowa) cd.

- Węzeł \$2

LOAD 2.5

STOR \$2

```
LOAD GAMMA
STOR $1
LOAD BETA
ADD $1
```

MPY \$2

Notacja symboliczna (assemblerowa) cd.

- Węzeł \$3

LOAD 2.5

STOR \$2

```
LOAD GAMMA
STOR $1
LOAD BETA
ADD $1
```

MPY \$2

STOR ALFA

1.5. Rodzaje błędów i sposoby radzenia sobie z nimi

Pytanie

Jakie rodzaje błędów mogą występuwać w procesie komplikacji ?

- błędy leksykalne (tzw. ortograficzne)
- syntaktyczne (składniowe)
- semantyczne

Definicja

Leksykalne - błędy w budowie jednostek leksykalnych (tokenów), np. nielegalne znaki lub słowa, błędy te nazwane są czasami błędami ortograficznymi;

Definicja

Syntaktyczne - błędy w budowie struktur syntaktycznych, np. brakujące operatory, niedomknięcie nawiasów, itd.;

Definicja

Semantyczne - ogólnie rozumiane błędy niezgodności, np. niezgodność typów, wielokrotne deklaracje lub brak deklaracji symboli, zbyt głębokie zagnieżdżanie pętli oraz instrukcji warunkowych, itd.

- Błędy leksykalne mogą być wykrywane już na etapie pracy skanera.
- Błędy syntaktyczne zasadniczo w trakcie pracy parsera, choć można sobie wyobrazić iż także w trakcie analizy leksykalnej, szczególnie np. gdy wiadomo jakiego symbolu (słowa kluczowego) w danym momencie możemy się spodziewać, inna rzecz że takie przerzucanie zadań na skaner wydaje się wątpliwe i niepotrzebne.
- Błędy semantyczne wykrywane są już tylko przez parser.

Pytanie

Czym jest domykanie analizy ?

Przyjmijmy że w dowolnym punkcie rozbioru program źródłowy ma postać:

$xTt \quad |$

gdzie x oznacza przetworzoną już część programu, T symbol kolejny do sprawdzenia, a t pozostała część programu.

- Wystąpienie błędu w analizie generacyjnej oznacza iż po wyprowadzeniu x nie jesteśmy w stanie wyprowadzić symbolu T .
- W analizie redukcyjnej natomiast, że np. koniec (sufiks) symbolu x nie tworzy argumentu redukcji.
- Są to oczywiście sytuacje błędne generujące odpowiedni komunikat zależny od kontekstu (zaawansowania) analizy.
- Jeżeli chcemy proces translacji kontynuować dalej, to musimy w tym miejscu domknąć jakoś analizę, tzn. jakby na własną rękę „naprawić błąd”.

Pytanie

Jakie znasz sposoby domykania analizy ?

Można to uczynić na kilka sposobów:

- 1 usunąć symbol T i kontynuować analizę;
- 2 pomiędzy x a T wstawić odpowiedni ciąg symboli podstawowych q , co da nam wtedy $xqTt$, i kontynuować analizę teraz od qTt ;
- 3 tak jak w poprzednim punkcie wstawić odpowiedni ciąg symboli q , ale dalszą analizę kontynuować od symbolu T ;
- 4 usunąć odpowiednie symbole z końca napisu x .

2 Przypomnienie z gramatyk

Pytanie

Czym jest gramatyka ? Z czego się składa ?

Gramatyka służy do specyfikowania składni języka. Gramatyka składa się z:

- zbiór symboli terminalnych (w skrócie terminali) - symbole terminalne są elementarnymi symbolami języka definiowanego przez te gramatykę
- zbiór symboli nieterminalnych (w skrócie nieterminali) - każdy symbol nieterminalny oznacza zbiór ciągów utworzonych z symboli terminalnych w sposób, który zamierzamy opisać
- zbiór produkcji - każda produkcja składa się z symbolu nieterminalnego nazywanego nagłówkiem lub lewą stroną produkcji, strzałki oraz sekwencji terminali lub nieterminali, nazywanej ciałem lub prawą stroną produkcji. Intuicyjnym celem produkcji jest określenie jednej z możliwych form konstrukcji językowej; jeśli nagłówkowy nieterminal reprezentuje pewna konstrukcje, wówczas ciało produkcji przedstawia formę zapisu tej konstrukcji
- jeden wyróżniony symbol nieterminalny jest traktowany jako symbol startowy

Specyfikowanie gramatyki wykonujemy, wyliczając jej produkcje, zaczynając od produkcji dla symbolu startowego.

Przykład

Gramatyka dla list cyfr rozdzielanych znakami plus lub minus:

$$\begin{aligned}list &\rightarrow list + digit \\list &\rightarrow list - digit \\list &\rightarrow digit \\digit &\rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9\end{aligned}$$

Moglibyśmy też korzystać z symbolu "lub" oznaczanego |, wtedy trzy z naszych produkcji moglibyśmy zastąpić przez:

$$list \rightarrow list + digit \mid list - digit \mid digit$$

Pytanie

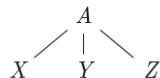
Co nazywamy "wyprowadzeniem" ?

Mówimy, że gramatyka wyprowadza ciągi symboli, zaczynając od symbolu startowego i kolejno zastępując symbol nieterminalny ciałem produkcji dla tego nieterminala. Wszystkie ciągi symboli terminalnych, które mogą zostać wyprowadzone z symbolu startowego, tworzą język zdefiniowany przez te gramatykę.

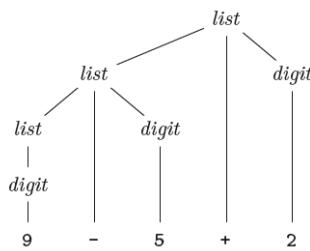
Pytanie

Czym jest drzewo rozbioru ?

Drzewo rozbioru (parse tree) graficznie pokazuje, jak z symbolu startowego gramatyki wywodzi się ciąg występujący w języku. Jeśli nieterminalne A ma produkcje $A \rightarrow XYZ$, wówczas drzewo analizy może zawierać pośredni węzeł oznaczony etykietą A z trzema potomnymi węzłami opisanymi jako X, Y oraz Z, od lewej do prawej:



Np. wyprowadzenie zapisu 9-5+2 dla gramatyki z powyższego przykładu prezentuje poniższe drzewo:



Pytanie

Kiedy mówimy, że gramatyka jest niejednoznaczna ?

Gramatyka może dopuszczać więcej niż jedno drzewo rozbioru generujące zadany ciąg terminali. Taka gramatyka nazywana jest niejednoznaczna. Formułując to inaczej, niejednoznaczna gramatyka to taka, w której pewne zdanie ma więcej niż jedno lewostronne (lub prawostronne) wyprowadzenie.

Pytanie

Czym jest wyprowadzenie lewostronne, a czym wyprowadzenie prawostronne ?

- W wyprowadzeniach lewostronnych zawsze jest wybierany “najbardziej lewy” nieterminal i jest on przekształcany zgodnie z produkcją
- W wyprowadzeniach prawostronnych zawsze wybierany jest nieterminal znajdujący się najbardziej po prawej stronie i jest on przekształcany zgodnie z produkcją

Np. mamy taką gramatykę:

$$E \rightarrow E + E \mid E * E \mid - E \mid (E) \mid \text{id}$$

Wyprowadzenie lewostronne dla tej gramatyki to:

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(id+E) \Rightarrow -(id+id)$$

TEORIA KOMPILACJI I KOMPILATORY

Wyprowadzenie prawostronne dla tej gramatyki to:

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E + E) \Rightarrow -(E + \text{id}) \Rightarrow -(\text{id} + \text{id})$$

2.1. Przerywnik - zebranie powyższych pytań

Pytanie

Jakie rodzaje błędów mogą występować w procesie komplikacji ?

Pytanie

Czym jest domykanie analizy ?

Pytanie

Jakie znasz sposoby domykania analizy ?

Pytanie

Czym jest gramatyka ? Z czego się składa ?

Pytanie

Co nazywamy "wyprowadzeniem" ?

Pytanie

Czym jest drzewo rozbiórku ?

Pytanie

Kiedy mówimy, że gramatyka jest niejednoznaczna ?

Pytanie

Czym jest wyprowadzenie lewostronne, a czym wyprowadzenie prawostronne ?

3 Analatory składniowe - parsery

Na wykładzie największa część poświęcona jest działaniu analizatorów składniowych. Rozważamy różne algorytmy, które można zastosować do realizacji analizy składniowej. Na nich skupimy pozostałą część tego opracowania.

Pytanie

Czym jest analiza składniowa ?

Analiza składniowa (parsing) to proces ustalania, czy i jak dany ciąg symboli terminalnych może zostać wygenerowany przez gramatykę. W omawianiu tego problemu pomocne będzie myślenie o budowaniu drzewa rozbioru, nawet jeśli w praktyce kompilator może takowego nie konstruować. W każdym przypadku jednak analizator składniowy (parser) zasadniczo musi być w stanie skonstruować drzewo; w przeciwnym razie nie będzie można zagwarantować poprawności tłumaczenia.

Pytanie

Jakie znasz trzy ogólne typy analizatorów składniowych ?

Rozróżniamy trzy ogólne typy analizatorów składniowych dla gramatyk:

- uniwersalne - np. Cock'a-Youngera-Kasamiego lub Earleya, mogą przetwarzarć dowolną gramatykę, są złożone obliczeniowo, za mało wydajne do rozwiązań produkcyjnych
- zstępujące - budują drzewa rozbioru od korzenia w dół do liści
- wstępujące - zaczynają od liści i budują drzewa wyprowadzenia w górę do korzenia

W każdym jednak przypadku dane wejściowe parsera są skanowane od lewej do prawej, po jednym symbolu naraz.

Najbardziej wydajne metody zstępujące i wstępujące działają tylko dla pewnych podklas gramatyk, ale wiele z tych klas, a w szczególności gramatyki LL i LR, jest wystarczająco wyrazistych, aby opisać większość konstrukcji składniowych występujących w nowoczesnych językach programowania.

Pytanie

Jakie znasz dwa podstawowe typy analizatorów składniowych ze względu na kierunek analizy tekstu i użytą metodę wyprowadzenia ?

- LL (top-down, left-to-right) - analizuje tekst od lewej do prawej i produkuje lewostronne wyprowadzenie metodą zstępującą
- LR (bottom-up, left-to-right) - analizuje tekst od lewej do prawej i produkuje prawostronne wyprowadzenia metodą wstępującą

Pytanie

Co jest wejściem do parsera ?

Wejściem do parsera są tokeny.

Pytanie

W oparciu o co pracuje parser ?

Parser pracuje w oparciu o gramatykę.

3.1. Przerywnik - zebranie powyższych pytań

Pytanie

Jakie znasz trzy ogólne typy analizatorów składniowych ?

Pytanie

Jakie znasz dwa podstawowe typy analizatorów składniowych ze względu na kierunek analizy tekstu i użytą metodę wyprowadzenia ?

Pytanie

Co jest wejściem do parsera ?

Pytanie

W oparciu o co pracuje parser ?

4 Rekurencja lewostronna i lewostronna faktoryzacja

Jeżeli gramatyka jest lewostronnie rekurencyjna, to nie możemy do niej zastosować metody zstępującej rozbioru składniowego. Zanim przejdziemy do opisu algorytmu parserów LL(k) (które korzystają z metody zstępującej) omówimy sobie rekurencje lewostronne.

Pytanie

Kiedy w gramatyce występuje lewostronna rekurencja ?

Gramatyka jest lewostronnie rekurencyjna, jeśli zawiera taki nieterminal A, że istnieje wyprowadzenie

$$A \implies A\alpha$$

dla pewnego ciągu α . Przy czym wyprowadzenie takie nie musi być bezpośrednie (może być wynikiem zastosowania jednej lub większej liczby produkcji). Przy takich gramatykach istnieje ryzyko wejścia w nieskończoną pętlę.

Przykład

Prosta gramatyka z rekurencją lewostronną:

$$A \rightarrow Aa \mid b$$

Przykład

Tutaj gramatyka, która ma dodatkowo rekurencje lewostronną, która jest efektem kilku kroków:

$$\begin{aligned} S &\rightarrow Aa \mid b \\ A &\rightarrow Ac \mid Sd \epsilon \end{aligned}$$

Możliwe wyprowadzenie to:

$$S \implies Aa \implies Sda$$

co daje rekurencję lewostronną.

W dalszej części omówimy sobie sposoby na usuwanie lewostronnej rekurencji z gramatyki.

4.1. Metoda 1 - zamiana problematycznej produkcji

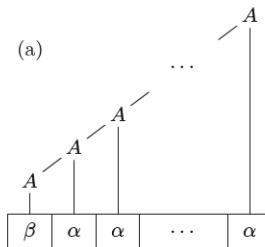
Pytanie

Omów metodą usuwania lewostronnej rekurencji z gramatyki poprzez zamianę problematycznej produkcji.

Jedną z najprostszych metod usunięcia lewostronnej rekurencji jest zamiana kłopotliwej produkcji poprzez wprowadzenie nowego symbolu nieterminalnego. Założymy, że mamy taką gramatykę:

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

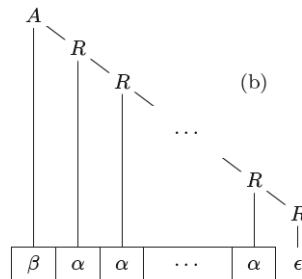
α oraz β oznaczają dowolny ciąg symboli terminalnych i nieterminalnych, które nie zaczyna się od A . Taka gramatyka jest niewątpliwie lewostronnie rekurencyjna. Drzewo wyprowadzenia dla takiej gramatyki ma taką postać:



Aby usunąć lewostronną rekurencję zamieniamy problematyczną produkcję używając nowego symbolu nieterminalnego R w następujący sposób:

$$\begin{aligned} A &\rightarrow \beta R \\ R &\rightarrow \alpha R \mid \epsilon \end{aligned}$$

Taka zamiana nie powoduje zmiany języka opisywanego przez gramatykę. Ale teraz drzewo wyprowadzenia będzie wyglądało tak:



Usuń z podanej gramatyki rekurencje lewostronną:

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow (E) \mid \text{id} \end{aligned}$$

Najpierw stosujemy wzór dla pierwszej, a potem dla drugiej produkcji:

$$\begin{aligned} E &\rightarrow T E' \\ E' &\rightarrow + T E' \\ T &\rightarrow F T' \\ T' &\rightarrow * F T' \\ F &\rightarrow (E) \mid \text{id} \end{aligned}$$

4.2. Metoda 2 - dla dowolnej liczby produkcji rekurencyjnie lewostronnych (z wykładow)

Pytanie

Omów metodę usuwania lewostronnej rekurencji dla dowolnej liczby produkcji rekurencyjnie lewostronnych (z wykładow).

Na początek grupujemy produkcje jak poniżej

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \dots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$$

gdzie żadne β_i nie zaczyna się od A . Następnie zastępujemy produkcje przez wprowadzenie nowego symbolu nieterminalnego jak poniżej:

$$\begin{aligned} A &\rightarrow \beta_1 \mid \dots \mid \beta_n \mid \beta_1 A' \mid \dots \mid \beta_n A' \\ A' &\rightarrow \alpha_1 \mid \dots \mid \alpha_m \mid \alpha_1 A' \mid \dots \mid \alpha_m A' \end{aligned}$$

Ta procedura eliminuje wszystkie rekurencje lewostronne z produkcji A i A' (pod warunkiem, że żadne z α_i nie jest ciągiem pustym), ale nie usuwa rekurencji lewostronnej występującej w wyprowadzeniach w dwóch i więcej krokach. Informacja z wykładów:

- W przypadku rekursji lewostronnej pośredniej nie ma prostego sposobu pozbycia się jej, choć można łatwo sprawdzić czy gramatyka taka zawiera symbol pośrednio rekurencyjny.

Przykład

Dla rozważanej przez nas w metodzie pierwszej gramatyki:

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow (E) \mid \text{id} \end{aligned}$$

Stosujemy metodę 2 i otrzymujemy

$$\begin{aligned} E &\rightarrow T \mid TE' \\ E' &\rightarrow +T \mid +TE' \\ T &\rightarrow F \mid FT' \\ T' &\rightarrow *F \mid *FT' \\ F &\rightarrow (E) \mid a \end{aligned}$$

Troszczek inny algorytm jest przedstawiony w książce, sprowadza on się jednak do tego samego. Stosujemy taką zamianę:

$$\begin{array}{c} A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \dots \mid \beta_n A' \\ A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \dots \mid \alpha_m A' \mid \epsilon \end{array}$$

Stosując ten wzór nasza gramatyka po przekształceniu miałaby taką postać:

$$\begin{array}{l} E \rightarrow TE' \\ E' \rightarrow +TE' \mid \epsilon \\ T \rightarrow FT' \\ T' \rightarrow *FT' \mid \epsilon \\ F \rightarrow (E) \mid a \end{array}$$

Istnieje także algorytm, który radzi sobie z niebezpośrednią rekurencją lewostronną, ale jest dość złożony (jest opisany w książce).

4.3. Lewostronna faktoryzacja

Pytanie

Po co stosowana jest lewostronna faktoryzacja ?

Lewostronna faktoryzacja jest przekształceniem gramatyki do takiej postaci, która nadaje się do analizy predykcyjnej (zstępującej). Nie każda gramatyka nadaje się do stosowania z analizatorem typu LL, który korzysta z analizy zstępującej. Lewostronna faktoryzacja pozwala na takie przekształcenie gramatyki aby mogła ona zostać zastosowana z analizatorem LL.

Pytanie

Kiedy stosujemy lewostronną faktoryzację ? Jaki jest algorytm działania ?

Jeśli wybór między dwiema alternatywnymi A-produkcjami nie jest oczywisty. Możemy być w stanie przepisać te produkcje, tak aby odłożyć decyzje do chwili, gdy będziemy mieli przeczytaną dostateczną liczbę znaków z wejścia, aby móc dokonać właściwego wyboru. Z przykładu:

Tak otrzymana gramatyka nie jest jednak $LL(k)$.

Konieczna jest tu tzw. *lewostronna faktoryzacja*, będąca jakby wyłączeniem przed nawias

$$A \rightarrow \alpha\beta_1 \mid \dots \mid \alpha\beta_n \Rightarrow \begin{cases} A \rightarrow \alpha A' \\ A' \rightarrow \beta_1 \mid \dots \mid \beta_n \end{cases}$$

W ogólności, jeśli mamy dwie A produkcje takie, że

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$$

i dane wejściowe zaczynają się od niepstego ciągu wyprowadzonego z α nie możemy wiedzieć czy mamy rozwijać A na $\alpha\beta_1$ czy na $\alpha\beta_2$. Jednak możemy opóźnić tę decyzję rozwijając A do $\alpha A'$. Wówczas po ujrzeniu wejścia wyprowadzonego z α rozwiniemy A' do β_1 albo β_2 .

Przykład

Kontynuujemy nasz przykład dla którego przeprowadziliśmy usunięcie lewostronnej rekursji. Startowaliśmy od gramatyki:

$$\begin{array}{lcl} E & \rightarrow & E + T \mid T \\ T & \rightarrow & T * F \mid F \\ F & \rightarrow & (E) \mid \text{id} \end{array}$$

Po usunięciu lewostronnej rekursji:

$$\begin{array}{l} E \rightarrow T | TE' \\ E' \rightarrow +T | +TE' \\ T \rightarrow F | FT' \\ T' \rightarrow *F | *FT' \\ F \rightarrow (E) | a \end{array}$$

Po zastosowaniu lewostronnej faktoryzacji:

$$\begin{array}{l} E \rightarrow TT'' \\ T'' \rightarrow \varepsilon | E' \\ E' \rightarrow +TT'' \\ T \rightarrow FF'' \\ F'' \rightarrow \varepsilon | T' \\ T' \rightarrow *FF'' \\ F \rightarrow (E) | a \end{array}$$

Zwróćmy uwagę, że aż cztery pierwsze produkcje wymagają lewostronnej faktoryzacji. Nie dla każdej jednak wygenerowaliśmy nowy symbol nieterminalny. Czasami zdarza się, że jeden nowy symbol będzie nas służył dla kilku produkcji. Np. symbol T'' został użyty zarówno do faktoryzacji pierwszej jak i drugiej produkcji, a symbol F'' do trzeciej i czwartej.

Dopiero tak przekształcona gramatyka mogłaby być użyte z parserem typu LL, ponieważ taka gramatyka jest klasy $LL(k)$.

Moglibyśmy także zastosować drugi ze sposobów na eliminację lewostronnej rekursji, zauważmy, że w tym przypadku nie jest konieczne stosowanie lewostronnej faktoryzacji, bo od razu mamy gramatykę w poprawnej postaci:

$$\begin{array}{lcl} E & \rightarrow & E + T \mid T \\ T & \rightarrow & T * F \mid F \\ F & \rightarrow & (E) \mid \text{id} \end{array}$$

Po usunięciu lewostronnej rekursji:

$$\begin{array}{l} E \rightarrow TE' \\ E' \rightarrow +TE' \mid \epsilon \\ T \rightarrow FT' \\ T' \rightarrow *FT' \mid \epsilon \\ F \rightarrow (E) \mid a \end{array}$$

5 Przerywnik - zebranie powyższych pytań

Pytanie

Kiedy w gramatyce występuje lewostronna rekurencja ?

Pytanie

Omów metodą usuwania lewostronnej rekurencji z gramatyki poprzez zamianę problematycznej produkcji.

Pytanie

Omów metodę usuwania lewostronnej rekurencji dla dowolnej liczby produkcji rekurencyjnie lewostronnych (z wykładu).

Pytanie

Po co stosowana jest lewostronna faktoryzacja ?

Pytanie

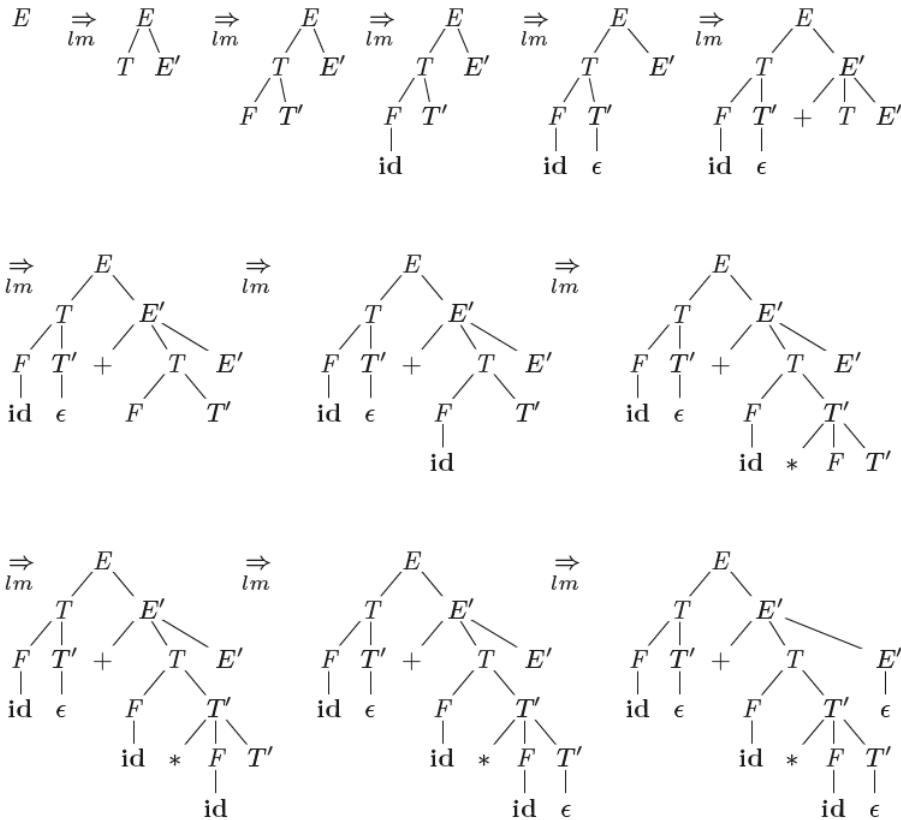
Kiedy stosujemy lewostronną faktoryzację ? Jaki jest algorytm działania ?

6 Wstęp do analizatorów składniowych LL(k)

Analizę zstępującą można traktować jako wyszukiwanie lewostronnego wyprowadzenia ciągu wejściowego. Analiza zstępująca zgodna z gramatyką:

$$\begin{aligned} E &\rightarrow TE' \\ E' &\rightarrow +TE'|\epsilon \\ T &\rightarrow FT' \\ T' &\rightarrow *FT'|\epsilon \\ F &\rightarrow (E)|a \end{aligned}$$

dla wejścia $id+id*id$ jest przedstawiona w postaci drzewa na poniższym rysunku:



Pytanie

Co jest kluczowym problemem w każdym kroku analizy składniowej?

Kluczowym problemem jest wybór produkcji. Jako przykład, na rysunku powyżej mamy węzły, które zawierają nieterminal E' . Skąd mamy wiedzieć, którą z produkcji należy zastosować dla takich węzłów? Są przecież dwie możliwe:

$$E' \rightarrow +TE'|\epsilon$$

Pytanie

Czym jest analiza zejść rekurencyjnych, a czym jest analiza predykcyjna ?

Są to typy analizy składniowej podzielone ze względu na sposób w jaki wybierana jest produkcja do zastosowania:

- Analiza zejść rekurencyjnych - może wymagać wycofania ruchów w celu znalezienia właściwej do zastosowania produkcji
- Analiza predykcyjna - wycofanie nie jest wymagane, analiza predykcyjna wybiera właściwą produkcję przez sprawdzenie pewnej ustalonej liczby symboli wejściowych do przodu przy czym typowo trzeba sprawdzić tylko jeden (czyli następny symbol wejściowy). W przykładzie powyżej w jednym z węzłów E' jest wybierana produkcja na $+TE'$, a w drugim na ϵ . Parser może wybrać właściwą z nich dzięki podejrzeniu następnego symbolu wejściowego

Pytanie

Czym jest klasa gramatyk $LL(k)$?

Klasa gramatyk, dla których można skonstruować parsery predykcyjne wymagające sprawdzenia nie więcej niż k symboli w przód, jest nazywana klasą $LL(k)$.

Pytanie

Czym są zbiory $FIRST$ i $FOLLOW$?

Przy ożyciu zbiorów $FIRST$ i $FOLLOW$ dla gramatyki można skonstruować "tablice analizy predykcyjnej", które sprawiają, że wybieranie produkcji podczas analizy zstępującej staje się oczywiste. Zbiory te są również użyteczne podczas analizy wstępnej, co zobaczymy później.

7 Zbiory $FIRST(\alpha)$ oraz $FOLLOW(A)$

Konstrukcje zarówno zstępującego, jak i wstępującego parsera wspierają dwie funkcje, $FIRST$ oraz $FOLLOW$, powiązane z gramatyką G . Podczas analizy zstępującej $FIRST$ i $FOLLOW$ pozwalają wybrać produkcje do zastosowania na podstawie kolejnego symbolu wejściowego.

7.1. Zbiór $FIRST$

Pytanie

Czym jest funkcja $FIRST(\alpha)$?

Funkcję $FIRST(\alpha)$, gdzie α jest dowolnym ciągiem symboli gramatycznych, definiujemy jako:

- zbiór terminali rozpoczynających ciągi, które można wyprowadzić z α

UWAGI:

- Jeżeli $\alpha \implies \epsilon$ (w jednym lub więcej kroków) to wtedy także ϵ należy do $FIRST(\alpha)$

Pytanie

Jak można użyć $FIRST$ w analizie predykcyjnej ?

Rozważmy dwie A-produkcje:

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

gdzie $FIRST(\alpha)$ i $FIRST(\beta)$ są zbiorami rozłącznymi. Możemy wówczas wybrać między tymi produkcjami, sprawdzając kolejny symbol wejściowy a, gdyż a może należeć co najwyżej do jednego ze zbiorów $FIRST(\alpha)$ albo $FIRST(\beta)$, ale nie do obu.

Przykładowo, jeśli a należy do $FIRST(\beta)$ wybieramy produkcję $A \rightarrow \beta$.

Pytanie

Czym jest funkcja $PRFX_k(\alpha)$?

Jest to funkcja analogiczna do $FIRST(\alpha)$, ale zamiast brać pierwszy znak z ciągu wyprowadzalnego, bierze prefiks o długości k . Klimek używa tej funkcji na wykładach w kilku miejscach. Ogólnie jest to po prostu uogólnienie funkcji $FIRST(\alpha)$ na prefiksy o długości k . Z wykładu:

Przykładowo, rozważając produkcje

$$A \rightarrow aB \mid bB \mid cD$$

określimy

$$PRFX_1(A) = \{a, b, c\}$$

7.2. Zbiór FOLLOW

Pytanie

Czym jest funkcja $FOLLOW(A)$?

Dla nieterminala A funkcje $FOLLOW(A)$ definiujemy jako zbiór terminali a, które mogą wystąpić bezpośrednio na prawo od A w pewnej formie zdaniowej.

Inaczej mówiąc, jest to zbiór terminali a takich, że istnieje wyprowadzenie w postaci

$$S \implies \alpha A a \beta$$

w pewnej liczbie kroków. Zauważmy, że mogą też istnieć jakieś symbole między A i a w pewnym momencie podczas wyprowadzenia. Ale jeśli tak, to w pewnym momencie one wyprowadzają ϵ i ostatecznie znikają.

W wyznaczaniu $FOLLOW$ mogą pomóc nam poniższe dwie reguły:

Jeśli istnieje produkcja $A \rightarrow \alpha B \beta$, wówczas cała zawartość $FIRST(\beta)$ z wyjątkiem ϵ należy do $FOLLOW(B)$.

Jeśli istnieje produkcja $A \rightarrow \alpha B$ lub produkcja $A \rightarrow \alpha B \beta$, gdzie $FIRST(\beta)$ zawiera ϵ , wówczas każdy element $FOLLOW(A)$ należy do $FOLLOW(B)$.

W dalszej części (przy budowaniu parsera LL(1)) będziemy wyznaczać zbiory $FIRST$ i $FOLLOW$ dla konkretnej gramatyki.

8 Przerywnik - zebranie powyższych pytań

Pytanie

Czym jest kluczowym problemem w każdym kroku analizy składniowej ?

Pytanie

Czym jest analiza zejść rekurencyjnych, a czym jest analiza predykcyjna ?

Pytanie

Czym jest klasa gramatyk $LL(k)$?

Pytanie

Czym są zbiory $FIRST$ i $FOLLOW$?

Pytanie

Czym jest funkcja $FIRST(\alpha)$?

Pytanie

Jak można użyć $FIRST$ w analizie predykcyjnej ?

Pytanie

Czym jest funkcja $PRFX_k(\alpha)$?

Pytanie

Czym jest funkcja $FOLLOW(A)$?

9 Gramatyka LL(1)

Pytanie

Dla jakiej klasy gramatyk można skonstruować parsery LL ?

Parsery predykcyjne, czyli parsery oparte nazejściach rekurencyjnych, które nie wymagają nawracania, można skonstruować dla klasy gramatyk nazywanych LL(k).

Pytanie

Jaką gramatykę nazywamy gramatyką klasy LL(1) ?

Gramatyka G jest klasy LL(1) wtedy i tylko wtedy, gdy dla każdej pary różnych produkcji $A \rightarrow \alpha \mid \beta$ z G spełnione są poniższe warunki:

1. Dla każdego terminala a nie można wyprowadzić z obu α i β ciągów rozpoczynających się od a .
2. Co najwyżej z jednego spośród α i β można wyprowadzić ciąg pusty.
3. Jeżeli $\beta \xrightarrow{*} \epsilon$, wówczas z α nie można wyprowadzić żadnego ciągu zaczynającego się od terminala należącego FOLLOW(A). Analogicznie, jeśli $\alpha \xrightarrow{*} \epsilon$, wówczas z β nie można wyprowadzić żadnego ciągu rozpoczynającego się od terminala należącego do FOLLOW(A).

Pierwsze dwa warunki są równoważne wymaganiu, aby FIRST(α) i FIRST(β) były zbiorami rozłącznymi. Trzeci warunek jest równoważny stwierdzeniu, że jeśli ϵ należy do FIRST(β), wówczas FIRST(α) i FOLLOW(A) są zbiorami rozłącznymi i podobnie, gdy ϵ należy do FIRST(α).

Jeśli zaczynamy więc budowę parsera LL(1), to najpierw sprawdzamy czy gramatyka jest LL(1), w szczególności usuwamy lewostronną rekurencję, a następnie wykonujemy lewostronną faktoryzację jeśli to konieczne.

Powyżej mamy definicję gramatyki LL(1) pochodząą z książki, na wykładzie Klimek podał ogólną definicję gramatyki LL(k), która jest analogiczna do tej dla LL(1), ale poszerza definicję na dowolne k :

Definicja

Jeżeli gramatyka G jest bezkontekstowa oraz dla każdych dwóch wyvodów zachodzi:

$$S \xrightarrow{*I} wA\alpha \xrightarrow{I} w\beta\alpha \xrightarrow{*I} wx$$

$$S \xrightarrow{*I} wA\alpha \xrightarrow{I} w\gamma\alpha \xrightarrow{*I} wy$$

to gramatykę nazywamy gramatyką $LL(k)$ gdy

$$\text{jeżeli } PRFX_k(x) = PRFX_k(y) \text{ to } \beta = \gamma^I$$

gdzie $w, x, y \in V^*$, $\alpha, \beta, \gamma \in (N \cup V)^*$, $A \in N$.

Gramatyka $LL(k)$ w wywodzie lewostronnym ma własność taką, że zawsze k pierwszych symboli słowa końcowego w sposób jednoznaczny określa regułę (produkcję) jaką należy zastosować.

Twierdzenie

Jeżeli $G = < N, V, P, S >$ jest gramatyką bezkontekstową, to G jest gramatyką $LL(k)$ wtedy i tylko wtedy, gdy dla każdej pary produkcji $(A \rightarrow \beta, A \rightarrow \gamma) \in P$ oraz dla $wA\alpha$ takiego że

$S \xrightarrow{*I} wA\alpha$, gdzie $w \in V^*$, $A \in N$ oraz $\alpha \in (N \cup V)^*$, zachodzi że:

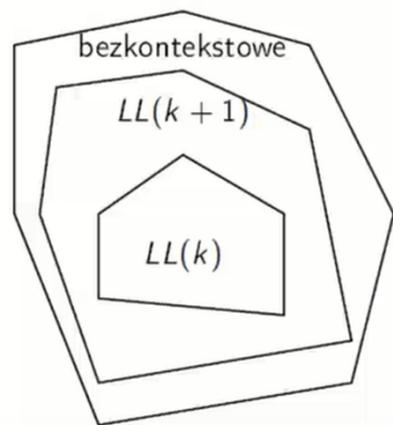
$$PRFX_k(\beta\alpha) \cap PRFX_k(\gamma\alpha) = \emptyset$$

W oparciu o pojęcia $PRFX_k$ i $FOLLOW_k$ możliwe jest także uproszczone zdefiniowanie gramatyki $LL(1)$.

Definicja

Bezkontekstowa gramatyka G jest $LL(1)$ wtedy i tylko wtedy, gdy $\forall A \rightarrow \alpha_1|\alpha_2|...|\alpha_n$ zachodzi, że

- 1 $PRFX_1(\alpha_1) ... PRFX_1(\alpha_n)$ są parami rozłączne;
- 2 jeżeli $\alpha_i \xrightarrow{*} \epsilon$, to $PRFX_1(\alpha_j) \cap FOLLOW_1(A) = \emptyset$ dla $1 \leq j \leq n, i \neq j$.



10 Przerywnik - zebranie powyższych pytań

Pytanie

Dla jakiej klasy gramatyk można skonstruować parsery LL ?

Pytanie

Jaką gramatykę nazywamy gramatyką klasy LL(1) ?

11 Budowa parsera LL(1)

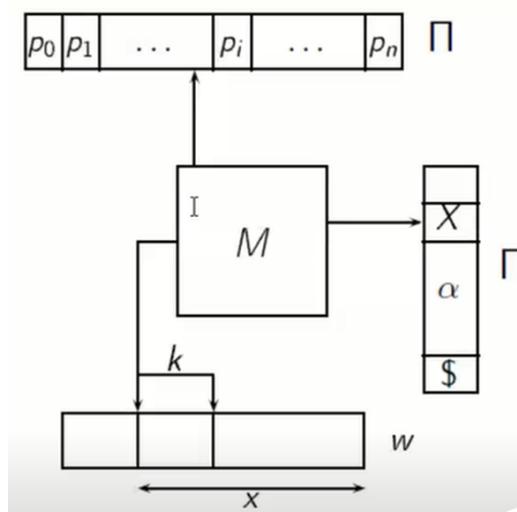
W końcu powiemy sobie o tym jak zbudować parser LL(1).

Pytanie

Jak w ogólności działa / jest zbudowany parser LL(k) ?

Najważniejszym elementem jest tabelka sterująca M . Tabelka mówi nam, którą produkcję mamy zastosować. Oprócz tego mamy wejście oraz stos. Na podstawie tego co znajduje się na stosie i co jest na wejściu, wyszukujemy w tabelce sterującej odpowiednią produkcję do zastosowania.

Na rysunku poniżej wejście oznaczone jest w , stos znajduje się na rysunku po prawej i jest zakończony znakiem $\$$ (który służy do sprawdzania czy stos jest pusty). Na górze znajdują się numery produkcji, które zostały zastosowane w procesie parsowania słowa wejściowego.



Oczywiście pierwszą częścią budowy takiego parsera jest budowa tabelki sterującej M . Przedstawiemy algorytm jej budowy dla gramatyki LL(1). Gdy mamy już taką tabelkę, to możemy wziąć dowolne wejście w i sprawdzić czy należy ono do języka opisywanego przez gramatykę. Po budowie tabelki przedstawiemy algorytm, który to realizuje.

11.1. Budowa tabelki sterującej M dla parsera LL(1)

11.1.1. Struktura tabeli

Pytanie

Jak wygląda struktura tabelki sterującej ? Co wpisujemy w wierszach, a co w kolumnach ?

- W kolejnych wierszach wpisujemy wszystkie symbole, które mogą się pojawić na stosie, a zatem:
 - wszystkie symbole terminalne
 - wszystkie symbole nieterminalne (ale nie ϵ !)
 - znak \$
 - UWAGA: w wierszach nie wpisujemy symbolu ϵ , nie pojawi się on na stosie
- W kolejnych kolumnach wpisujemy wszystkie możliwe symbole terminalne, tym razem z ϵ .

Pokażmy to na przykładzie. Rozpatrujemy gramatykę, którą już wcześniej używaliśmy:

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow (E) \mid a \end{aligned}$$

Na początek zawsze sprawdźmy czy jest LL(1). Widzimy, że jest lewostronna rekursja więc ją usuwamy według jednej z omówionych metod:

$$\begin{aligned} E &\rightarrow TE' \\ E' &\rightarrow +TE' \mid \epsilon \\ T &\rightarrow FT' \\ T' &\rightarrow *FT' \mid \epsilon \\ F &\rightarrow (E) \mid a \end{aligned}$$

Widzimy dalej, że nie potrzebna jest lewostronna faktoryzacja, bo jest ok. Dodatkowo korzystnie jest zapisać każdą z produkcji w osobnej linii i nadać produkcjom numery. Ułatwi nam to budowanie tabelki sterującej:

1. $E \rightarrow TE'$
2. $E' \rightarrow +TE'$
3. $E \rightarrow \epsilon$
4. $T \rightarrow FT'$
5. $T' \rightarrow *FT' \mid \epsilon$
6. $T' \rightarrow \epsilon$
7. $F \rightarrow (E)$
8. $F \rightarrow a$

Gramatyka jest LL(1), możemy przystąpić do tworzenia tabelki sterującej.

W gramatyce mamy 5 różnych nieterminali: E, E', T, T', F oraz 5 różnych terminali, które mogą pojawić się na stosie: $a, (,), +, *$. Te znaki zapisujemy w wierszach i na koniec dokładamy znak końca stosu:

E					
E'					
T					
T'					
F					
a					
(
)					
+					
*					
\$					

Teraz wypełnimy kolumny. W gramatyce mamy 6 różnych terminali: $a, (,), +, *, \epsilon$:

	a	()	+	*	ϵ
E						
E'						
T						
T'						
F						
a						
(
)						
+						
*						
\$						

Mamy strukturę, teraz przejdziemy do wypełniania tabelki według algorytmu.

11.2. Wypełnianie tabelki sterującej

Pierwsze dwie reguły algorytmu z wykładu mówią, że:

- $\forall a \in V : M(a, a) = \text{pop}$
- $M(\$, \varepsilon) = \text{accept};$

A więc po ich zastosowaniu otrzymamy:

	a	()	+	*	ϵ
E						
E'						
T						
T'						
F						
a	pop					
(pop				
)			pop			
+				pop		
*					pop	
\$						acc

Trzecia reguła algorytmu z wykładu mówi, że:

- jeżeli $p_i = (A \rightarrow \alpha) \in P$ to
 $\forall a \in PRFX_1(\alpha), a \neq \varepsilon : M(A, a) = (\alpha, i)$,
 a jeżeli $\varepsilon \in PRFX_1(\alpha)$ to
 $\forall b \in FOLLOW_1(A) : M(A, b) = (\alpha, i)$

O co w niej chodzi ? Przede wszystkim dla każdej prawej strony produkcji $A \rightarrow \alpha$ musimy obliczyć zbiór $FIRST(\alpha)$. Dalej postępujemy w zależności od tego czy $FIRST(\alpha)$ zawiera ϵ czy też nie.

- Jeżeli $FIRST(\alpha)$ nie zawiera ϵ , to sprawia jest dość prosta, bo dla każdego terminala a ze zbioru $FIRST(\alpha)$ wpisujemy w tabelce w miejscu $M[A, a]$ prawą stronę produkcji, czyli α oraz numer produkcji.
- Jeżeli $FIRST(\alpha)$ zawiera ϵ , to musimy dodatkowo obliczyć zbiór $FOLLOW(A)$. Wtedy dla każdego terminala b ze zbioru $FOLLOW(A)$ wpisujemy w tabelce w miejscu $M[A, b]$ prawą stronę produkcji, czyli α oraz numer produkcji.

11.2.1. Obliczanie $FIRST(\alpha)$

W tym miejscu przypomnijmy naszą gramatykę dla wygody:

1. $E \rightarrow TE'$
2. $E' \rightarrow +TE'$
3. $E \rightarrow \epsilon$
4. $T \rightarrow FT'$
5. $T' \rightarrow *FT'$
6. $T' \rightarrow \epsilon$
7. $F \rightarrow (E)$
8. $F \rightarrow a$

Wypełnianie musimy zacząć więc od obliczenia $FIRST(\alpha)$ dla każdej prawej strony produkcji...

$$1. FIRST(TE') = FIRST(FT'E') = FIRST(aT'E') \cup FIRST((E)T'E') = \{ a, (\}$$

Najpierw zamieniliśmy T na FT' , to jedyny wybór w gramatyce. Potem zamieniliśmy F , tym razem są dwa wybory więc uwzględniamy obydwa. Zauważmy, że za każdym razem zamieniamy tylko pierwszy symbol, bo obchodzą nas tylko terminale występujące na początku ciągu.

W momencie gdy otrzymaliśmy takie terminale, kończymy analizę i mamy wyznaczony zbiór $FIRST(\alpha)$.

W naszym zbiorze wynikowym nie ma symbolu ϵ , a więc na podstawie tego obliczenia uzupełniamy w tabelce sterującej komórki: $M[E, a] = TE'$, 1 oraz $M[E, ()] = TE'$, 1.

	a	$($	$)$	$+$	$*$	ϵ
E	$TE', 1$	$TE', 1$				
E'						
T						
T'						
F						
a	pop					
$($		pop				
$)$			pop			
$+$				pop		
$*$					pop	
$$$						acc

i liczymy dalej dla pozostałych produkcji:

$$2. FIRST(+TE') = \{ + \}, \text{ a zatem } M[E', +] = +TE', 2$$

3. $FIRST(\epsilon) = \{\epsilon\}$ - tu mamy inny przypadek, nasz zbiór wynikowy zawiera ϵ , a więc musimy dalej liczyć $FOLLOW(E')$. Ogólnie przy wyznaczaniu $FOLLOW$ musimy być bardzo uważni, czasami nie od razu widać co będzie wynikiem.

- Na pewno musimy wyjść od E , a więc możliwa jest sytuacja w której: $E \rightarrow TE'$, tutaj na prawo od E' mamy ϵ .
- Zauważmy, że E' możemy uzyskać tylko z E i znajduje się ono na końcu ciągu. Stąd możemy wnioskować o tym, że $FOLLOW(E') = FOLLOW(E)$. Na podstawie możliwego wyprowadzenia $F \rightarrow (E) \rightarrow (TE')$ wnioskujemy, że) należy do $FOLLOW(E')$

Podsumowując $FOLLOW(E') = \{ \epsilon, () \}$, a zatem $M[E', \epsilon] = \epsilon, 3$ oraz $M[E', ()] = \epsilon, 3$.

4. $FIRST(FT') = FIRST(aT'E') \cup FIRST((E)T'E') = \{ a, () \}$, zatem $M[T, a] = FT', 4$ oraz $M[T, ()] = FT', 4$

5. $FIRST(*FT') = \{ *\}$, zatem $M[T', *] = *FT', 5$

6. $FIRST(\epsilon) = \{\epsilon\}$ - czyli musimy liczyć $FOLLOW(T')$.

$FOLLOW(T') = \{ +, (), \epsilon \}$, generalnie pokazanie tego nie jest łatwe i wymaga znajomości reguł związanych z wyznaczaniem $FOLLOW$... czyli dostaniemy $M[T', +] = (E), 7$

7. $FIRST((E)) = \{ () \}$, zatem $M[F, ()] = (E), 7$

7. $FIRST((E)) = \{ () \}$, zatem $M[F, ()] = (E), 7$

8. $FIRST(a) = \{ a \}$, zatem $M[F, a] = a, 8$

Na potwierdzenie, te same obliczenia przeprowadzone na wykładzie:

$$\begin{aligned} (1). \quad & PRFX_1(TE') = PRFX_1(FT'E') = \\ & PRFX_1((E)T'E') \cup PRFX_1(aT'E') = \{(), a\} \\ & \text{czyli } M(E, ()) = (TE', 1), M(E, a) = (TE', 1) \end{aligned}$$

$$\begin{aligned} (2). \quad & PRFX_1(+TE') = \{ + \} \\ & \text{czyli } M(E', +) = (+TE', 2) \end{aligned}$$

$$\begin{aligned} (3). \quad & PRFX_1(\epsilon) = \{ \epsilon \} \\ & \text{więc liczymy } FOLLOW_1(E') \\ & \text{czyli } M(E', ()) = (\epsilon, 3), M(E', \epsilon) = (\epsilon, 3) \end{aligned}$$

$$\begin{aligned} (4). \quad & PRFX_1(FT') = PRFX_1((E)T') \cup PRFX_1(aT') = \\ & \{(), a\} \\ & \text{czyli } M(T, ()) = (FT', 4), M(T, a) = (FT', 4) \end{aligned}$$

- (5). $PRFX_1(*FT') = \{*\}$
 czyli $M(T', *) = (*FT', 5)$
- (6). $PRFX_1(\varepsilon) = \{\varepsilon\}$
 więc liczymy $FOLLOW_1(T')$
 czyli $M(T',)) = (\varepsilon, 6)$, $M(T', +) = (\varepsilon, 6)$,
 $M(T', \varepsilon) = (\varepsilon, 6)$
- (7). $PRFX_1((E)) = \{()\}$
 czyli $M(F, ()) = ((E), 7)$
- (8). $PRFX_1(a) = \{a\}$
 czyli $M(F, a) = (a, 8)$

Ostatecznie otrzymujemy taką tabelkę (screen z wykładu):

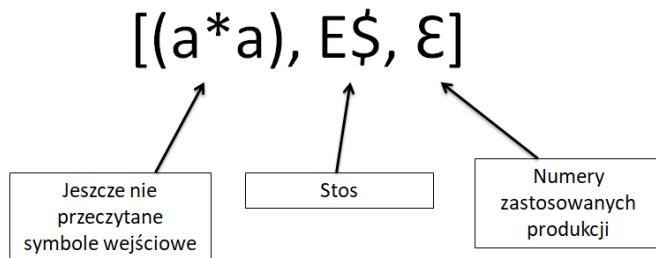
	a	$($	$)$	$+$	$*$	ε
E	I $TE', 1$	$TE', 1$				
E'			$\varepsilon, 3$	$+TE', 2$		$\varepsilon, 3$
T	$FT', 4$	$FT', 4$				
T'			$\varepsilon, 6$	$\varepsilon, 6$	$*FT', 5$	$\varepsilon, 6$
F	$a, 8$	$(E), 7$				
a	<i>pop</i>					
$($		<i>pop</i>				
$)$			<i>pop</i>			
$+$				<i>pop</i>		
$*$					<i>pop</i>	
$\$$						<i>acc</i>

UWAGA: Pusta kratka też ma znaczenie w tablicy. Pusta kratka oznacza błąd (sytuacja gdy słowo wejściowe nie należy do języka). W sytuacji gdybyśmy wpisywali coś w zajętą już kratkę, to oznacza, że nasza gramatyka nie jest klasy LL(k).

12 Proces parsowania

Gdy mamy już tabelkę sterującą, możemy przejść do procesu parsowania, czyli sprawdzania czy dane słowo wejściowe należy do języka opisywanego przez gramatykę.

W procesie parsowania będziemy przekształcać taką tablicę:



Pierwszy jej element, to nieprzeczytane jeszcze wejście. Powyższy rysunek zakłada słowo wejściowe: $(a * a)$. Drugi z elementów to stos, początkowo na stosie jest symbol startowy, a po nim znak końca stosu. Ostatni element to zastosowane produkcje, początkowo to ϵ , bo nie zastosowaliśmy jeszcze żadnej produkcji.

Teraz postępujemy według prostego algorytmu:

- Jeżeli pierwszy symbol na wejściu jest symbolem terminalnym równym symbolowi terminalnemu na wierzchołku stosu, to oba te symbole usuwamy, zarówno ze stosu jak i z wejścia (zgodnie z tabelką, mamy instrukcje pop na przecięciu tych samych terminali)
- Jeżeli na wierzchołku stosu mamy terminal A , a na wejściu mamy symbol wejściowy a , to wyszukujemy w tabelce sterującej komórkę $M[A, a]$, jeżeli jest pusta, to znaczy, że słowo nie należy do języka. A jeżeli nie jest pusta, to jej zawartość dopisujemy na stos w zamian za symbol, który jest na jego wierzchu. Dopisujemy też numer produkcji z tej komórki do naszej tablicy.
- Gdy dojdziemy do momentu w którym na stosie będzie tylko $\$$, a na wejściu nie będzie już nic to mamy instrukcję acc i oznacza to, że słowo należy do języka.

Pokażemy teraz przykład dla zbudowanej przez nas tabelki. Sprawdzamy czy słowo $(a*a)$ należy do języka opisywanego przez gramatykę.

Dla wygody przypominamy tabelkę sterującą:

	<i>a</i>	()	+	*	ε
<i>E</i>	$\text{I } TE', 1$	$TE', 1$				
<i>E'</i>			$\varepsilon, 3$	$+TE', 2$		$\varepsilon, 3$
<i>T</i>	$FT', 4$	$FT', 4$				
<i>T'</i>			$\varepsilon, 6$	$\varepsilon, 6$	$*FT', 5$	$\varepsilon, 6$
<i>F</i>	$a, 8$	$(E), 7$				
<i>a</i>	<i>pop</i>					
(<i>pop</i>				
)			<i>pop</i>			
+				<i>pop</i>		
*					<i>pop</i>	
\$						<i>acc</i>

Rozpoczynamy od:

$$[(a * a), E\$, \varepsilon]$$

Na początku mamy do przeczytania całe słowo ($a * a$), a na stosie jest symbol startowy *E*. Nie zastosowaliśmy jeszcze żadnej produkcji.

Pierwszy symbol na wejściu to (, a terminal na wierzchu stosu to *E*, a więc patrzmy co jest w komórce $M[E, ()]$. Jest tam TE' oraz produkcja numer 1. A więc otrzymujemy:

$$[(a * a), TE'\$, 1]$$

Teraz na wejściu mamy nadal symbol (, ale na wierzchu stosu jest teraz *T*. Patrzmy więc na komórkę $M[T, ()]$. Jest tam FT' oraz produkcja numer 4. Z więc otrzymujemy:

$$[(a * a), FT'E'\$, 14]$$

postępując analogicznie następnie otrzymamy:

$$[(a * a), (E)T'E'\$, 147]$$

Teraz mamy sytuację gdzie na stosie jest terminal (. Taki sam terminal jest na wejściu, oznacza to, że musimy zastosować operację pop, po której otrzymamy:

$$[a * a), E)T'E'\$, 147]$$

i postępujemy analogicznie dalej:

$$\begin{aligned}
 & [a * a), TE')T'E' \$, 1471] \\
 & [a * a), FT'E')T'E' \$, 14714] \\
 & [a * a), aT'E')T'E' \$, 147148] \\
 & [*a), T'E')T'E' \$, 147148] \\
 & [*a), *FT'E')T'E' \$, 1471485] \\
 & [a), FT'E')T'E' \$, 1471485] \\
 & [a), aT'E')T'E' \$, 14714858] \\
 & [], T'E')T'E' \$, 14714858] \\
 & [], E')T'E' \$, 147148586] \\
 & [],)T'E' \$, 1471485863] \\
 & [\epsilon, T'E' \$, 1471485863] \\
 & [\epsilon, E' \$, 14714858636] \\
 & [\epsilon, \$, 147148586363]
 \end{aligned}$$

Jak widać otrzymaliśmy ostatecznie puste wejście oraz pusty stos, a więc gramatyka akceptuje słowo wejściowe. Moglibyśmy na podstawie uzyskanych produkcji wyprowadzić nasze słowo:

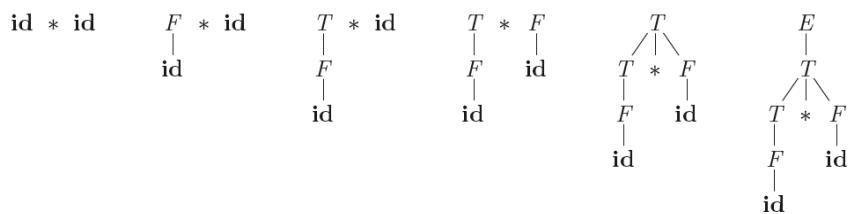
$$\begin{aligned}
 E &\xrightarrow{1} TE' \xrightarrow{4} FT'E' \xrightarrow{7} (E)T'E' \xrightarrow{1} (TE')T'E' \xrightarrow{4} (FT'E)T'E' \xrightarrow{8} \\
 &(aT'E')T'E' \xrightarrow{5} (a * FT'E')T'E' \xrightarrow{8} (a * aT'E')T'E' \xrightarrow{6} \\
 &(a * aE')T'E' \xrightarrow{3} (a * a)T'E' \xrightarrow{6} (a * a)E' \xrightarrow{3} (a * a)
 \end{aligned}$$

13 Wstęp do analizatorów składniowych LR

Pytanie

Czym jest analiza wstępująca ?

Analiza wstępująca (bottom-up) odpowiada budowaniu drzewa rozbioru dla ciągu wejściowego, zaczynając od liści (od dołu) i postępując w kierunku korzenia (w góre).



Uchwyty (handles)

- Uchwyty prawostronnej formy zdaniowej γ to produkcja $A \rightarrow \beta$ i pozycja w γ , na której znajduje się ciąg symboli β , który należy zastąpić przez A , aby otrzymać poprzednią prawostronną formę zdaniową w wyprowadzeniu prawostronnym γ .
- Jeśli

$$\begin{array}{c} S \Rightarrow^* \alpha Aw \\ \quad\quad\quad R \quad\quad\quad R \end{array}$$
to:
 $A \rightarrow \beta$ na pozycji po α jest uchwytem $\alpha\beta w$

Kilka dodatkowych informacji o uchwytnach:

- Ciąg symboli po prawej stronie uchwytu zawiera tylko symbole terminalne
- Dla danej formy zdaniowej γ może istnieć więcej niż jeden uchwyty, ponieważ gramatyka może być niejednoznaczna (może istnieć więcej niż jedno prawostronne wyprowadzenie $\alpha\beta w$)
- Jeśli gramatyka jest jednoznaczna to każda prawostronna forma zdaniowa γ z tej gramatyki ma dokładnie jeden uchwyty

Pytanie

Czym jest przycinanie uchwytów ?

Przycinanie uchwytów

- w – łańcuch (ciąg terminali), który chcemy przeanalizować
- $w = \gamma_n$, gdzie γ_n n-ta prawostronna forma zdaniowa w pewnym wyprowadzeniu prawostronnym
- $S = \gamma_0 \Rightarrow \gamma_1 \Rightarrow \dots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n = w$
- Wyszukujemy β_n w γ_n i zastępujemy lewą stroną produkcji $A_n \rightarrow \beta_n$ w wyniku otrzymujemy γ_{n-1} , itd.
- Jeśli dojdziemy do S to analiza kończy się sukcesem – zapisany od końca ciąg produkcji poszczególnych kroków redukcji jest prawostronnym wyprowadzeniem

W 

narazie nie wiemy jeszcze jak wyznaczać uchwyty.

13.2. Analiza metodą przesunięcie-redukcja

Pytanie

Czym jest analiza metodą przesunięcie-redukcja ? Na czym polega implementacja stosowa analizatora przesunięcie-redukcja ?

Metoda przesunięcie-redukcja jest formą analizy wstępnej, w której stos przechowuje symbole gramatyczne, a bufor wejściowy zawiera pozostałą część ciągu do przeanalizowania. Jak zobaczymy, uchwyt zawsze pojawia się na wierzchołku stosu bezpośrednio przed tym, gdy zostanie zidentyfikowany jako uchwyt.

Implementacja takiego analizatora zakłada, że:

- Stos służy do przechowywania symboli gramatyki
- Bufor wejściowy do przechowywania łańcucha w
- Za pomocą \$ oznaczam dno stosu i prawy koniec bufora
- Konfiguracja początkowa:
 - Stos: \$
 - Wejście: w\$

Analizator przesuwa zero lub więcej symboli z bufora na stos, do momentu kiedy na wierzchołku stosu będzie uchwyt β . Wtedy analizator redukuje β do lewej strony odpowiedniej produkcji. Proces powtarza się do wystąpienia błędu lub do momentu, kiedy na stosie będzie symbol startowy a wejście będzie puste.

UWAGA: Można pokazać, że uchwyt zawsze znajduje się na wierzchołku stosu! (dowód w książce).
Przykład:

STOS	WEJŚCIE	AKCJA
\$	$\mathbf{id}_1 * \mathbf{id}_2 \$$	przesunięcie
\$ \mathbf{id}_1	$* \mathbf{id}_2 \$$	redukacja $F \rightarrow \mathbf{id}$
\$ F	$* \mathbf{id}_2 \$$	redukacja $T \rightarrow F$
\$ T	$* \mathbf{id}_2 \$$	przesunięcie
\$ $T *$	$\mathbf{id}_2 \$$	przesunięcie
\$ $T * \mathbf{id}_2$	\$	redukacja $F \rightarrow \mathbf{id}$
\$ $T * F$	\$	redukacja $T \rightarrow T * F$
\$ T	\$	redukacja $E \rightarrow T$
\$ E	\$	akceptacja

Pytanie

Jakie operacje są możliwe w analizatorze przesunięcie-redukcja ?

- Przesunięcie – pobranie symbolu z bufora wejściowego i wstawienie go na wierzchołek stosu

- Redukcja – wykonywana gdy prawy koniec uchwytu jest na wierzchołku stosu. Odszukanie na stosie lewego końca uchwytu i zastąpienie go właściwym nieterminalem
- Akceptowanie – pomyślne zakończenie analizy
- Błąd – oznacz, że wejście w zawiera błąd składniowy

13.3. Gramatyki dla których można stosować analizę przesunięcie-redukcja

Pytanie

Czy stosowanie analizy przesunięcie-redukcja jest możliwe dla wszystkich gramatyk ?

Nie, jest możliwe tylko dla gramatyk LR(k) (analogicznie jak mieliśmy w przypadku gramatyk LL(k) i parserów LL). Tutaj k oznacza również liczbę symboli podglądanych.

Pytanie

Czym jest gramatyka LR(k) ?

Poniżej mamy formalną definicję z wykładu. Pamiętać musimy, że jest to zawsze gramatyka jednoznaczna i bezkontekstowa.

Definicja gramatyki LR(k)

- GBK $G=(V, \Sigma, P, S)$ dla której $S \xrightarrow{^n} S$ jedynie dla $n=0$ jest gramatyką LR(k) dla $k \geq 0$ wtedy i tylko wtedy jeśli z:

$$(i) S \xrightarrow[R]{*} \alpha A w \xrightarrow[R]{} \alpha \beta w \quad (\alpha, \beta \in (V \cup \Sigma)^*, A \in V, w \in \Sigma^*)$$

$$(ii) S \xrightarrow[R]{*} \alpha' A' x \xrightarrow[R]{} \alpha \beta w' \quad (\alpha' \in (V \cup \Sigma)^*, A' \in V, x, w' \in \Sigma^*)$$

$$(iii) {}^{(k)}W = {}^{(k)}w' \quad (\text{inaczej } FIRST_k(w) = FIRST_k(w'))$$

wynika, że:

$$\alpha A w' = \alpha' A' x \quad (\text{czyli } \alpha = \alpha', A = A', x = w')$$

Pytanie

Czym jest gramatyka wzbogacana ?

Po prostu dodajemy nowy symbol startowy S' i on przechodzi w stary symbol startowy S . To sprawia, że nasz parser wie kiedy zakończyć parsowanie - wtedy gdy stary symbol startowy przejdzie w nowy, czyli gdy parser może wykonać redukcje przy użyciu produkcji $S \rightarrow S'$

**Gramatyka wzbogacona/uzupełniona
(augmented grammar)**

- Dla GBK $G=(V, \Sigma, P, S)$ definiujemy gramatykę wzbogaconą G' :
 $G' = (V \cup \{S'\}, \Sigma, P \cup \{S' \rightarrow S\}, S')$
- Dla gramatyki wzbogaconej spełnione jest wymaganie:
 $S' \Rightarrow^n S'$ jedynie dla $n=0$

14 Sytuacje $LR(0)$

Pytanie

Skąd parser posługujący się metodą przesunięcie-redukcja wie, kiedy wykonać przesunięcie, a kiedy redukcję? Np. na poniższym rysunku: jeśli stos zawiera $\$T$ i kolejny symbol wejściowy to jak parser może stwierdzić, że T na wierzchołku stosu nie jest uchwytem, zatem odpowiednią akcją jest przesunięcie, a nie redukcja T do E ?

STOS	WEJŚCIE	AKCJA
$\$$	$\mathbf{id}_1 * \mathbf{id}_2 \$$	przesunięcie
$\$ \mathbf{id}_1$	$* \mathbf{id}_2 \$$	redukacja $F \rightarrow \mathbf{id}$
$\$ F$	$* \mathbf{id}_2 \$$	redukcja $T \rightarrow F$
$\$ T$	$* \mathbf{id}_2 \$$	przesunięcie
$\$ T *$	$\mathbf{id}_2 \$$	przesunięcie
$\$ T * \mathbf{id}_2$	$\$$	redukacja $F \rightarrow \mathbf{id}$
$\$ T * F$	$\$$	redukacja $T \rightarrow T * F$
$\$ T$	$\$$	redukacja $E \rightarrow T$
$\$ E$	$\$$	akceptacja

Parser LR podejmuje decyzje o przesunięciu lub redukcji, otrzymując stany w celu śledzenia miejsca, w którym znajdujemy się w przebiegu analizy. Stany reprezentują zbiory "sytuacji" (item). Sytuacja $LR(0)$ (w skrócie sytuacja) gramatyki G to produkcja z tej gramatyki z dodanym znacznikiem (kropka) na pewnej pozycji ciała tej produkcji. Produkcji $A \rightarrow XYZ$ odpowiadają zatem cztery sytuacje

$$\begin{aligned} A &\rightarrow \cdot XYZ \\ A &\rightarrow X \cdot YZ \\ A &\rightarrow XY \cdot Z \\ A &\rightarrow XYZ \cdot \end{aligned}$$

Dla produkcji $A \rightarrow \epsilon$ mamy tylko sytuację $A \rightarrow \cdot$. Intuicyjnie sytuacja wskazuje, jak dużą część produkcji zobaczyliśmy już w pewnym punkcie procesu analizy.

Pytanie

Czym jest zbiór sytuacji $LR(0)$?

Pewna kolekcja zbiorów sytuacji $LR(0)$, nazywana kanoniczna kolekcja $LR(0)$, zapewnia podstawy do skonstruowania deterministycznego automatu skończonego, który jest używany do podejmowania decyzji podczas analizy. Automat taki nazywamy automatem $LR(0)$. W szczególności każdy stan automatu $LR(0)$ reprezentuje zbiór sytuacji z kolekcji kanonicznej $LR(0)$.

Aby skonstruować kanoniczną kolekcję $LR(0)$ dla gramatyki, definiujemy gramatykę wzbogaconą oraz dwie funkcje CLOSURE (domknięcie) i GOTO (przejście). Na podstawie tych dwóch funkcji będziemy w stanie wyznaczyć rodzinę $LR(0)$.

15 Funkcja CLOSURE

Pytanie

Jak działa funkcja domknięcia ?

Funkcja ta działa według dwóch reguł:

1. Najpierw dodajemy wszystkie sytuacje z I do $\text{CLOSURE}(I)$.
2. Jeżeli $A \rightarrow \alpha \cdot B\beta$ należy do $\text{CLOSURE}(I)$ i $B \rightarrow \gamma$ jest produkcją, wówczas dodajemy sytuację $B \rightarrow \cdot \gamma$ do $\text{CLOSURE}(I)$, o ile jeszcze jej nie ma w tym zbiorze. Regułę tę powtarzamy, aż żadnych nowych sytuacji nie będzie można dodać do $\text{CLOSURE}(I)$.

Rozważmy uzupełnioną gramatykę wyrażeń:

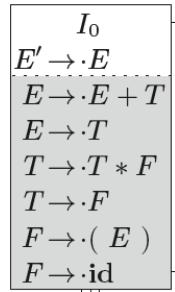
$$\begin{array}{lcl} E' & \rightarrow & E \\ E & \rightarrow & E + T \mid T \\ T & \rightarrow & T * F \mid F \\ E & \rightarrow & (E) \mid \text{id} \end{array}$$

Pamiętajmy aby zawsze najpierw wzbogacić gramatykę !

Na początku zawsze definiujemy zbiór I_0 , który początkowo zawiera tylko jedną sytuację gdzie wstawimy kropkę przed stary symbol startowy:

$$I_0 = \{E' \rightarrow \cdot E\}$$

Teraz stosujemy do tego zbioru regułę numer 2 i otrzymujemy



To pierwszy zbiór sytuacji w naszej rodzinie $LR(0)$. Aby wyznaczyć pozostałe musimy skorzystać z funkcji $GOTO$.

16 Funkcja GOTO

Pytanie

Jak działa funkcja GOTO ?

Funkcja $GOTO(I, X)$, to funkcja gdzie I jest zbiorem sytuacji, a X symbolem gramatycznym.

$GOTO(I, X)$ jest definiowana jako domknięcie zbioru wszystkich sytuacji $[A \rightarrow \alpha X \cdot \beta]$ takich, że $[A \rightarrow \alpha \cdot X \beta]$ należy do I .

Tzn. jeśli w zbiorze I mamy sytuację $[A \rightarrow \alpha \cdot X \beta]$, to do zbioru $GOTO(I, X)$ wpisujemy wszystkie sytuacje z I , które powstają przez przeniesienie kropki o jedno miejsce w prawo, tak aby znalazła się na prawo od symbolu X . Następnie dla tych sytuacji liczymy domknięcie tak jak już umiemy.

W naszym przykładzie mamy aktualnie tylko zbiór I_0 :

I_0
$E' \rightarrow \cdot E$
$E \rightarrow \cdot E + T$
$E \rightarrow \cdot T$
$T \rightarrow \cdot T * F$
$T \rightarrow \cdot F$
$F \rightarrow \cdot (E)$
$F \rightarrow \cdot id$

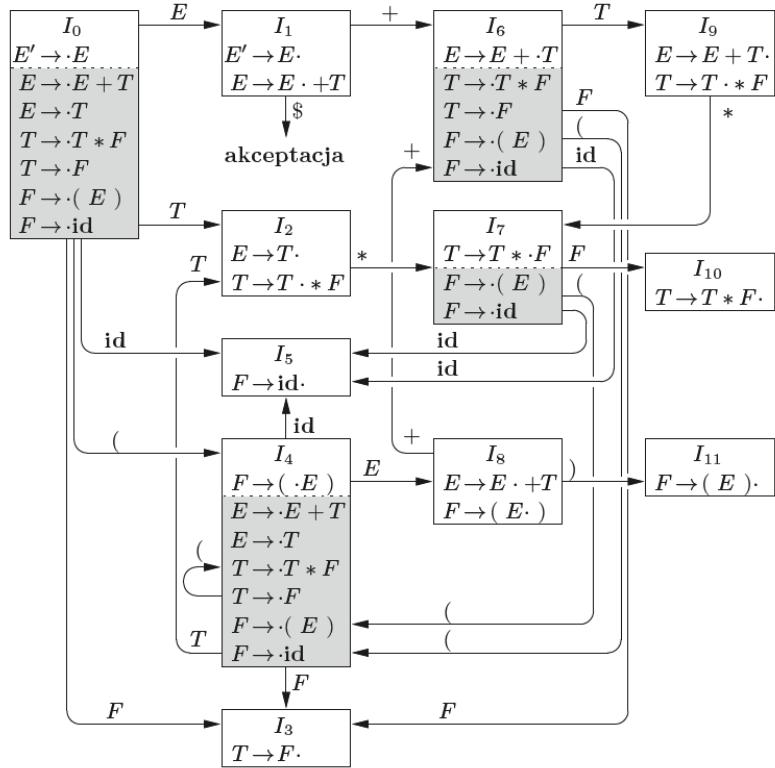
Chcemy policzyć np. $GOTO(I_0, E)$, bo mamy takie sytuacje w I_0 , że kropka znajduje się przed E :

- $[E' \rightarrow \cdot E]$
- $[E \rightarrow \cdot E + T]$

Przenosimy więc kropkę o jeden w prawo i dajemy takie sytuacje do zbioru $GOTO(I_0, E)$:

I_1
$E' \rightarrow E \cdot$
$E \rightarrow E \cdot + T$

Teraz obliczamy domknięcie dla tych sytuacji. W tym akurat przypadku jest tak, że nie ma żadnych możliwych do dodania przez domknięcie sytuacji. Analogicznie postępujemy dla pozostałych symboli. Na końcu dostajemy rodzinę sytuacji $LR(0)$



Szarym kolorem zaznaczono obliczone domknięcia. W efekcie dostaliśmy tzw. automat $LR(0)$, gdzie stany to zbiory sytuacji I_j , a przejścia to funkcje $GOTO$.

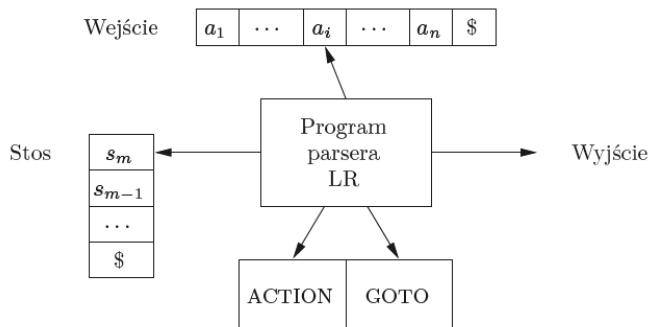
Stanem startowym takiego automatu jest zawsze $CLOSURE([S' \rightarrow \cdot S])$, czyli zbiór I_0 .

17 Tabelka sterująca parsera LR

Pytanie

Jak zbudowany jest parser LR ?

Podobnie o LL, mamy wejście, stos i tabelkę sterującą. Tabelka sterująca składa się z tabelki ACTION oraz GOTO. Omówimy sobie jak skonstruować tabelkę sterującą, a potem przejdziemy do omówienia procesu parsingu za pomocą LR.



Pytanie

Jakie znasz sposoby na tworzenia tabelki sterującej dla parserów LR ?

- SLR - tą sobie omówimy
- Canonical LR
- LALR

18 Proces parsowania LR

Pytanie

Jak przebiega proces konstruowania tabelki sterującej metodą SLR ?

1. Wzbogacamy gramatykę
2. Wyznaczamy rodzinę $LR(0)$ jak pokazaliśmy wcześniej
3. Wyznaczamy $FOLLOW(A)$ dla każdego nieterminala z gramatyki
4. Następnie wypełniamy tabelkę action i goto według reguł:
 2. Stan i konstruujemy z I_i . Akcje parsingu dla stanu i są ustalane jako poniżej:
 - (a) Jeżeli $[A \rightarrow \alpha \cdot a \beta]$ należy do I_i i $GOTO(I_i, a) = I_j$, wówczas ustawiamy $ACTION[i, a]$ jako „shift j ”. W tym przypadku a musi być terminalem.
 - (b) Jeżeli $[A \rightarrow \alpha \cdot]$ należy do I_i , ustawiamy $ACTION[i, a]$ jako „reduce $A \rightarrow \alpha$ ” dla wszystkich a należących $FOLLOW(A)$; w tym przypadku A nie może być S' .
 - (c) Jeżeli $[S' \rightarrow S \cdot]$ należy do I_i , ustawiamy $ACTION[i, \$]$ jako „accept”.

Jeśli z powyższych reguł wynikają jakiekolwiek konfliktowe akcje, mówimy, że gramatyka nie jest klasy SLR(1). W takim przypadku algorytm nie jest w stanie utworzyć parsera.
 3. Przejścia (GOTO) dla stanu i są konstruowane dla wszystkich nieterminali A przy użyciu reguły: jeżeli $GOTO(I_i, A) = I_j$, wówczas $GOTO[i, A] = j$.

Jak widzimy reguły te nie są wcale takie proste, a sam proces tworzenia parsera wymaga dużej uwagi. Nie będziemy więc po kolej przehodzić przez proces tworzenia parsera. Pokażemy jedynie wynikową tabelkę, a potem sposób jak na jej podstawie sprawdzić czy słowo należy do języka opisywanego przez gramatykę.

Wynikowa tabelka ma taką postać:

STAN	AKCJA					GOTO		
	id	+	*	()	\$	E	T	F
0	s5			s4		1	2	3
1		s6			acc			
2		r2	s7		r2	r2		
3		r4	r4		r4	r4		
4	s5			s4		8	2	3
5		r6	r6		r6	r6		
6	s5			s4			9	3
7	s5			s4				10
8		s6			s11			
9		r1	s7		r1	r1		
10		r3	r3		r3	r3		
11		r5	r5		r5	r5		

oznaczenia w tabelce:

$$\begin{array}{ll}
 (1) & E \rightarrow E + T \\
 (2) & E \rightarrow T \\
 (3) & T \rightarrow T * F \\
 (4) & T \rightarrow F \\
 (5) & F \rightarrow (E) \\
 (6) & F \rightarrow \mathbf{id}
 \end{array}$$

Kody dla poszczególnych akcji to:

1. si oznacza przesunięcie (*shift*) i umieszczenie na stosie stanu i ,
2. ri oznacza redukcję według produkcji o numerze j ,
3. acc oznacza akceptację,
4. puste oznacza błąd.

Pytanie

Jak działa algorytm parsingu w oparciu o tabelkę sterującą i gramatykę ?

Formalne reguły mówią, że:

Zachowanie parsera LR

- Jeżeli $\text{ACTION}[s_m, a_i] = \text{shift } s$, parser wykonuje przesunięcie przechodząc do konfiguracji:
 $(s_0 X_1 s_1 X_2 s_2 \dots X_m s_m a_i s, a_{i+1} \dots a_n \$)$
- Jeżeli $\text{ACTION}[s_m, a_i] = \text{reduce } A \rightarrow \beta$, parser wykonuje redukcję przechodząc do konfiguracji:
 $(s_0 X_1 s_1 X_2 s_2 \dots X_{m-r} s_{m-r} As, a_i a_{i+1} \dots a_n \$)$
gdzie $r = |\beta|$, $s = \text{GOTO}[s_{m-r}, A]$
Następuje wypisanie numeru produkcji π na wyjście

Mniej formalnie:

- Zawsze na początku mamy na stosie tylko stan 0, a w buforze wejściowym mamy całe słowo do przeanalizowania
- Patrzymy na stan s , który jest na wierzchu stosu i symbol a , który jest pierwszy w buforze wejściowym, odczytujemy z tabeli akcję $M[s, a]$
- Jeśli akcja to si , to na stos przesuwamy najpierw pierwszy symbol z bufora wejściowego a , a potem dopisujemy stan i
- Jeśli akcja to ri , to patrzymy na produkcję o numerze i , założymy, że produkcja ma postać $A \rightarrow \beta$. Z wierzchu stosu usuwamy tyle stanów i tyle symboli, jaką długość ma prawa strona produkcji, czyli jaką długość ma β . W efekcie uzyskujemy na wierzchu stosu symbol terminalny, a przed nim numer stanu. Dla tego numeru stanu podglądamy $GOTO[s, A]$ i dodajemy odczytany stan na wierzch stosu.

TEORIA KOMPILACJI I KOMPILATORY

Działanie parsera SLR zgodne z powyższymi regułami:

			Action						GOTO		
			id	+	*	()	\$	E	T	F
id+id*id	(1)	$E \rightarrow E + T$	s5				s4		1	2	3
	(2)	$E \rightarrow T$		s6				acc			
	(3)	$T \rightarrow T * F$	r2	s7		r2	r2				
	(4)	$T \rightarrow F$	r4	r4		r4	r4				
	(5)	$F \rightarrow (E)$			s4				8	2	3
	(6)	$F \rightarrow id$	s5		s4					9	3
											10
								s11			
									r1	r1	
									r1	r1	
									r3	r3	
									r3	r3	
									r5	r5	

	Stos	Wejście	Akcja
1	0	id+id\$id	przesunięcie
2	0 id 5	+id\$id	redukacja zgodnie z (6)
3	0 F 3	+id\$id	redukacja zgodnie z (4)
4	0 T 2	+id\$id	redukacja zgodnie z (2)
5	0 E 1	+id\$id	przesunięcie
6	0 E 1 + 6	id\$id	przesunięcie
7	0 E 1 + 6 id 5	*id\$	redukacja zgodnie z (6)
8	0 E 1 + 6 F 3	*id\$	redukacja zgodnie z (4)
9	0 E 1 + 6 T 9	*id\$	przesunięcie
10	0 E 1 + 6 T 9 * 7	id\$	przesunięcie
11	0 E 1 + 6 T 9 * 7 id 5	\$	redukacja zgodnie z (6)
12	0 E 1 + 6 T 9 * 7 F 10	\$	redukacja zgodnie z (3)
13	0 E 1 + 6 T 9	\$	redukacja zgodnie z (1)
14	0 E 1	\$	acc