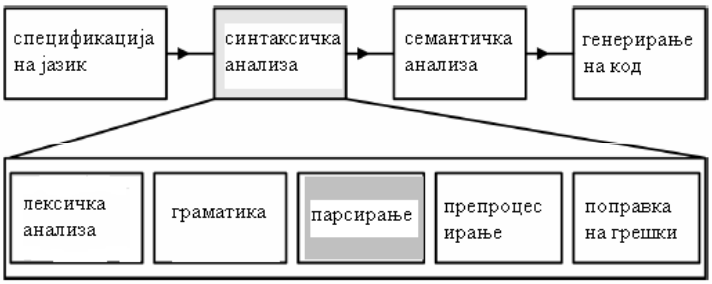
Парсирање

Вовед



Вовед

- Конструиравме граматики за формален јазик
  - За генерирање на валидна реченица треба да се напишат синтаксни дрва и синтаксни дијаграми
  - Компајлерите не работат врз основа на овие дијаграми туку врз основа на разбирање (парсирање) на реченициите
- Програмата која ги парсира реченициите се нарекува парсер.
  - Некој парсери градат синтаксни дрва за реченициите и така ги разбираат
  - Тие се генерираат од конкретна реченица, а не од синтаксен дијаграм

Вовед

- Парсерот е програма која го чита внесениот текст и кажува дали тој ги следи правилата на граматиката или не
  - Ако не треба да каже зошто
- Друг начин на работа на паресерот е да каже дали реченицата коже да биде генерирана од граматиката.
- Само посложените компајлери градат синтаксно дрво во нивната меморија
  - градбата на дрвата е илустративно за работата на компајлерот
- Поедноставување на синтаксно дрво со креирање на апстрактно синтаксно дрво (покомпактно)
  - Апстракното синтаксно дрво е битно за фазата на семантичка анализа и генерирање на код

Префиксен код

- Префиксната нотација е воведена од Полскиот научник J. Lukasiewicz.
- Нотација за која не требаат загради
  - Перфектна за излез од компајлерот
- Пример: 1 + 2 * 3 станува + 1 * 2 3
  1 / 2 - 3 станува - / 1 2 3

- При пишување на граматиката треба да се води сметка за асоцијативноста или за предимство на оператори
- Понатамошните фази не треба да водат сметка за тоа
  - Парсерот треба да го конвертира текстот во некој меѓу формат (меѓу јазик) во кој тоа ќе биде вметнато anymore{the parser should convert the input text to an intermediate format
  - Ова се решава со недвосмислено синтаксно дрво или со префиксна нотација

Префиксен код

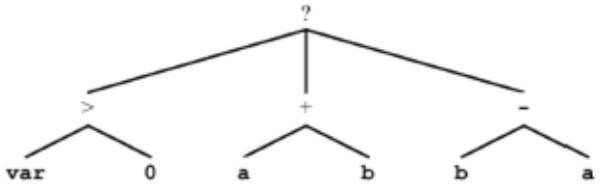
- Со префиксната нотација може да се претстават и покомплексни конструкции како *if . . . then* и *while .. do*
- Пример

  if (var > 0) { a + b } else { b – a} станува ?>var'0'+a'b'-b'a'

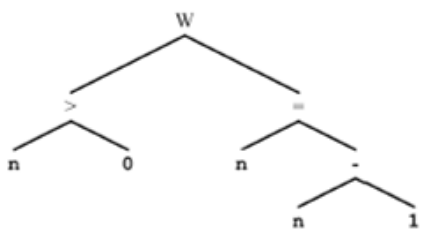
  while (n > 0) { n = n – 1} станува W>n'0'=n'-'n'1'
  - Апострофите (') се користат за да се одвојат променливите
- Префиксната нотација всушност е слично на дрво и со користење на рекурзивни функции може да се претвори во дрво
- Може да се конструира многу едноставен компајлер кој би превел математички изрази во префиксен код
- Потоа може да се направи втора програма-евалуатор-кој може да го интерпретира овој код и да го прорачуна резултатот.

Префиксен код

- if (var > 0) { a + b } else { b – a}



- while (n > 0) { n = n – 1}



Префиксен код

- Преискната нотација исто така помага во елиминирање на празни места и коментари без да се загуби смислата
- Програмата за евалуација работи рекурзивно
  - Почнува да го чита пефиксниот стринг од лево кон десно и за секој оператор кој ќе го изброи се повикува самата себе за да ги добие операндите
  - Рекурзијата завршува кога че се најдат константите (иминја на променливи или литерални врености).
- Евалуаторот работи на принцип што рекурзивната функција може да ја користи како стек во кој ги чува операнторите кои сеуште не се евалуирани
- Значајно својство на префиксниот код и соодветното дрво е тоа што сега операторите веќе не се листови во дрвото туку внатрешни јазли
  - Со ова се штеди простор за зачувување на дрвото

Теорија на прасирање

- Како се конструира синтаксното (парсирачко) дрво од даден оригинален влез?
- Гледаме леви граматики
- Два вида на парсери: top-down и bottom-up.
  - Top-down парсерот почнува од стартниот симбол од граматиката и работи над конкретната реченица со замена на правилата на граматиката при што ги заменува нетерминалите до нетерминали
    - Овој метод е полесен но бара големи рестрикции на граматиката
  - Bottom-up парсерот почнува со реченицата која ја земанува со нетерминалите од левата страна на правилата.
    - Помоќен но потежок да се напише на рака, па за таа цел постојат алатки за конструкција на вакви bottom-up парсери од дадена граматика

Top-down парсирање

- Го зема најлевиот нетерминал и го заменува со тој од десната стран
  - Ако има повеќе избори ќе треба да се види со што да се замени, зависно од наредниот симбол кој следи
  - Ако има повеќе можни избори треба да се пробува по сите (постојат вакви парсери) и ако со некоја проба се успее сите нетерминали да се заменат со терминали, тигаш реченицата е добра. Ако во сите можности не се добие сите нетерминали да се заменат, треба компајлерот да јави грешка.
- За контекстно слободна граматика не е битно кој нетерминал ќе се замени, но обично се заменува најлевиот

Top-down парсирање

expression: factor restexpression .
restexpression : e.
restexpression : + factor restexpression .
restexpression : - factor restexpression .
factor : 0.
factor : 1.
factor : 2
factor : 3.
factor : 4.
factor : 5.
factor : 6.
factor : 7.
factor : 8.
factor : 9.

1 +2 +3

expression => factor restexpression
=> 1 restexpression
=> 1 +factor restexpression
=> 1 +2 restexpression
=> 1 +2 +factor restexpression
=> 1 +2 +factor restexpression
=> 1 +2 +3 restexpression
=> 1 +2 +3

Top-down парсирање

- Со оваа граматика не може да се испарсира изразот 1 + 2 * 3
- Парсирањето ќе падне кога ќе се појави знакот *

  expression $=>_L$ 1 +2 restexpression

- Парсерот ќе бара замена која почнува со *, и бидејќи не постои таква нема со што да направи замена и реченицата не може да се распознае.

Лево рекурзивни граматики

- Контекстно слободна граматика е (V;Σ; S; P) лево рекурзивна ако има правило од облик Xα =>Xβ
- X може бесконечно да се заменива
- Пример:

  expression: expression + term.

  expression: expression - term.

  expression: term.

Лево рекурзивни граматики

- Левата рекурзија може да се отстрани
- Пример

  expression: term + expression.
  expression: term - expression.
  expression: term.
- Операторите + и – сега се третираат како да се десно асоцијативни,
  - Кај одземање изразот погрешно да се испарсира
- Но сега не може да се избере дали со кој од овие правила да се направи замена, дали со +, дали со – или без знак. Може да се гледа неколку чекори наназад-но до каде?
- Може да искористиме лева факторизација
  - term да се отстрани од почетокот и да се стави како посебно правило. Ова се нарекува “factoring out“. Се добива граматика со еден поглед на назад и е лево асоцијативна

    expression: term restexpression .
    restexpression : + term restexpression.
    restexpression : - term restexpression.
    restexpression : e.

Bottom-up парсирање

- Bottom-up парсерите работат обратно од top-down
- Почнуваат со целосниот внес на реченицата и работат со замена на терминалите со нетерминали користејќи ги правилата на граматиката во обратен правец.

Bottom-up парсирање

expression: expression + expression.
expression: expression - expression.
expression: factor .
factor : 0.
factor : 1.
factor : 2
factor : 3.
factor : 4.
factor : 5.
factor : 6.
factor : 7.
factor : 8.
factor : 9.

1 +2 +3

1 +2 + 3 => factor +2 +3
=> expression +2 +3
=> expression +factor +3
=> expression +expression +3
=> expression +3
=> expression +factor
=> expression +expression
=> expression

Множествата FIRST и FOLLOW

- Множеството FIRST на нетерминалот A е множество од сите терминали со кои може да почнува A.

  $FIRST(X) := \cup_{i=1}^{n} PFIRST(Xi)$

- FIRST множеството на factor е {0; 1; 2; 3; 4; 5; 6; 7; 8; 9}.
- Множеството FOLLOW на нетерминалот A е множество од сите терминали кои може да следуваат директно после A.
  - Ова множество се бара само за нетерминалите кои се јавуваат од десната страна на правилата.

Множествата FIRST и FOLLOW

- Множеството FIRST на нетерминалот A е множество од сите терминали со кои може да почнува A.

  $FIRST(X) := \cup_{i=1}^{n} PFIRST(Xi)$

- FIRST множеството на factor е {0; 1; 2; 3; 4; 5; 6; 7; 8; 9}.
- Множеството FOLLOW на нетерминалот A е множество од сите терминали кои може да следуваат директно после A.
  - Ова множество се бара само за нетерминалите кои се јавуваат од десната страна на правилата.

Множествата FIRST и FOLLOW

expression:	factor restexpression .
restexpression:	ϵ .
	| + factor restexpression
	| − factor restexpression .
factor:	**0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

FOLLOW(expression) = $\{\perp\}$[1]
FOLLOW(restexpresson) = $\{\perp\}$
FOLLOW(factor) = $\{\perp, +, -\}$

Production		PFIRST
program:	statementlist **RESULT**=implication.	$\{A \ldots Z\}$
statementlist :	ϵ.	$\emptyset$
statementlist :	statement statementlist.	$\{A \ldots Z\}$
statement:	identifier =implication ;.	$\{A \ldots Z\}$
implication :	conjunction restimplication .	$\{\sim, (, 0, 1, A \ldots Z\}$
restimplication :	ϵ.	$\emptyset$
restimplication :	−> conjunction restimplication .	{ -> }
restimplication :	<− conjunction restimplication .	{ <- }
restimplication :	<−>conjunction restimplication.	{ <-> }
conjunction:	negation restconjunction.	$\{\sim, (, 0, 1, A \ldots Z\}$
restconjunction :	ϵ.	$\emptyset$
restconjunction :	&negation restconjunction.	$\{\&\}$
restconjunction :	| negation restconjunction.	$\{\|\}$
negation:	~ negation.	$\{\sim\}$
negation:	factor .	$\{(, 0, 1, A \ldots Z\}$
factor :	(implication) .	$\{(\}$
factor :	identifier .	$\{A \ldots Z\}$
factor :	1.	$\{1\}$
factor :	0.	$\{0\}$
identifier :	A.	$\{A\}$
identifier :	Z.	$\{Z\}$

Table 6.1: PFIRST Sets for Logic Language

Множествата FIRST и FOLLOW

Nonterminal	FIRST	FOLLOW
program	$\{A \ldots Z\}$	$\{\perp\}$
statementlist	$\{A \ldots Z\}$	$\{RESULT\}$
statement	$\{A \ldots Z\}$	$\{A \ldots Z, RESULT\}$
implication	$\{\sim, (, 0, 1, A \ldots Z\}$	$\{;, \perp,)\}$
restimplication	{ ->, <-, <-> }	$\{;, \perp,)\}$
conjunction	$\{\sim, (, 0, 1, A \ldots Z\}$	$\{$->, <-, <->, $;, \perp,)\}$
restconjunction	$\{\&, \|\}$	$\{$->, <-, <->, $;, \perp,)\}$
negation	$\{\sim, (, 0, 1, A \ldots Z\}$	$\{\&, \|,$ ->, <-, <->, $;, \perp,)\}$
factor:	$\{(, 0, 1, A \ldots Z\}$	$\{\&, \|,$ ->, <-, <->, $;, \perp,)\}$
identifier:	$\{A \ldots Z\}$	$\{=, \&, \|,$ ->, <-, <->, $;, \perp,)\}$

Table 6.2: FIRST and FOLLOW Sets for Logic Language