Prednáška 12 - Sémantická analýza

Ing. Viliam Hromada, PhD.

C-510
Ústav informatiky a matematiky
FEI STU

viliam.hromada@stuba.sk

Štruktúra kompilátora

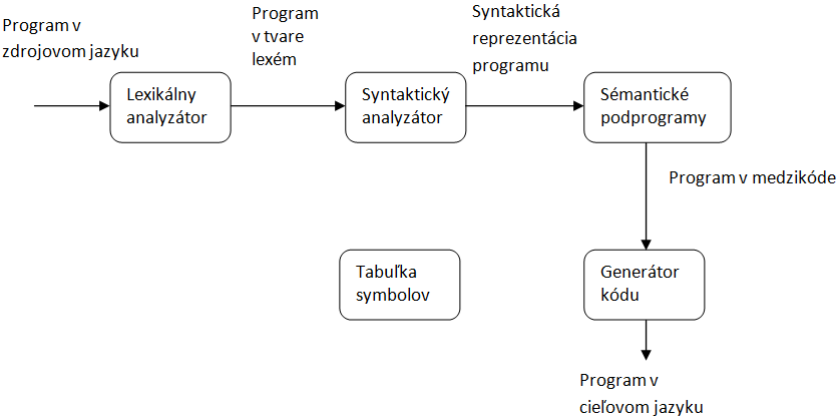


Figure: Prevzaté z Dedera, Ľ.: Počítačové jazyky a ich spracovanie.

Syntax [1]

1. pravidlá na vytváranie prípustných kombinácií symbolov, používaných **bez ohľadu na význam** symbolov a ich kombinácií

Sémantika [1]

1. náuka o **význame** jazykových jednotiek
2. **významová** stránka jazykových jednotiek

[1] Slovník cudzích slov (akademický). 2., doplnené a prepracované vyd. Spracoval kolektív autorov pod vedením V. Petráčkovej a J. Krausa. Preklad Ľ. Balážová, J. Bosák, J. Genzor, I. Ripka, J. Skladaná. Ed. Ľ. Balážová – J. Bosák. Bratislava: Slovenské pedagogické nakladateľstvo – Mladé letá 2005. 1054 s. ISBN 80-10-00381-6.

Cieľ sémantickej analýzy

- Preklad počítačových programov alebo spracovanie dokumentov vytvorených v nejakom formálnom jazyku je väčšinou riadené syntaktickou štruktúrou spracovávaného programu (dokumentu).
- V angličtine sa tento prístup označuje ako syntax-directed translation.
- Syntax programovacieho jazyka / štruktúra dokumentu atď. je teda popísaná nejakou (bezkontextovou) gramatikou, pričom jej terminálne symboly predstavujú lexikálne elementy.
- Na základe syntaxe sa potom vykonáva sémantické spracovanie vstupného dokumentu - t.j. čo majú jednotlivé jazykové konštrukcie / inštrukcie vykonávať.
- Zjednodušene: syntax = štruktúra, sémantika = interpretácia.

Prístup k sémantickej analýze

1. Syntaktický analyzátor vytvorí reprezentáciu programu alebo v tvare derivačného (syntaktického) stromu, alebo v tvare **abstraktného syntaktického stromu (AST)**.
2. Vrcholom stromu sa pridelia **sémantické atribúty** vyhodnocované počas sémantického spracovania.
3. Po vyhodnotení všetkých atribútov sa alebo priamo generuje cieľový kód, alebo sa generuje tzv. **medzikód.**. Teda vo všeobecnosti sa generuje nejaká cieľová reprezentácia informácie zapísanej v zdrojovom jazyku.

1. abstraktný syntaktický strom

- Vychádza z derivačného (syntaktického stromu), avšak abstrahuje od konkrétnej syntaxe (konkrétnych elementov gramatiky) a zachytáva len konceptuálne informácie dôležité z hľadiska sémantického spracovania.
- Napríklad pri gramatikách popisujúcich podmienený príkaz abstrahuje od použitých kľúčových slov; od toho, či sa časť *then vetvy* začína kľúčovým slovom **then**, atď.
- Pri gramatikách popisujúcich matematické výrazy môže abstrahovať od názvov jednotlivých neterminálov a zachytáva len použité operácie a operandy, atď.

2. sémantické atribúty

- Vrcholy AST sú ohodnotené tzv. **sémantickými atribútmi**, ktoré sa počas sémantickej analýzy vyhodnocujú.
- Pri preklade programov nimi môžu byť napr. názov, adresa a typ premennej; hodnota a typ konštanty; návestie pre skokovú inštrukciu; časti skompilovaných programov, atď.
- Na začiatku sémantického spracovania sú známe len sémantické atribúty listov AST, keďže tie väčšinou dodá lexikálny analyzátor.

Príklad *AST* č. 1

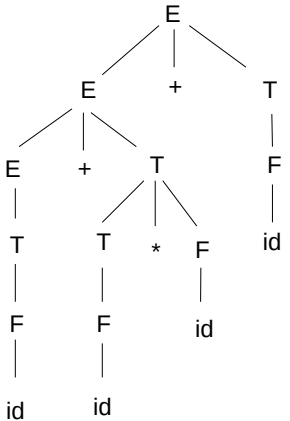
Nech je daná gramatika $G = (\{E, T, F\}, \{+, *, (,), id\}, P, E)$ s pravidlami P:

1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow id$
6. $F \rightarrow (E)$

Terminál *id* je lexéma predstavujúca identifikátor nejakej premennej, pre jednoduchosť uvažujme, že identifikátory premenných sú reťazce písmen. Terminály $+, *, (,)$ zodpovedajú príslušným symbolom.

Príklad *AST* č. 1

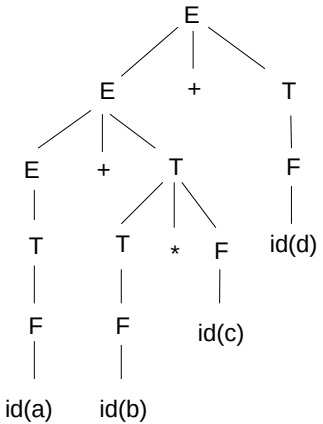
Reťazec $a + b * c + d$ je po lexikálnej analýze reprezentovaný ako postupnosť terminálnych symbolov $id + id * id + id$ a jeho derivačný strom:

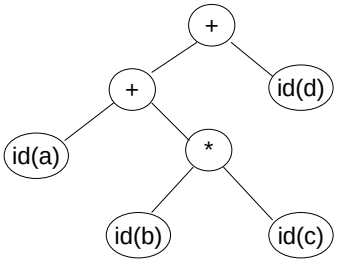


Príklad *AST* č. 1

- Problémom takéhoto derivačného stromu je, že momentálne nie je zrejmé, ktorý list *id* predstavuje ktorú premennú a, b, c, d.
- V praxi sa táto informácia dopĺňa do derivačného stromu a listy sa **ohodnotia** menom príslušnej premennej, aby sa táto informácia nestratila.
- Tieto vnesené informácie do vrcholov derivačného stromu sú príkladmi **sémantických atribútov**.
- Upravený derivačný strom, v ktorom sú lexémy ohodnotené príslušnými reťazcami, je na ďalšom slajde vľavo.
- Taktiež na ďalšom slajde - vpravo - je príslušný AST. Všimnite si abstrahovanie od mien neterminálov a zachytenie len informácie o mene premennej a použitej operácii.

Príklad *AST* č. 1





Príklad *AST* č. 1

- Priradenie mien premenných do príslušných vrcholov AST je príkladom sémantických atribútov.
- To, aké sémantické atribúty budú prislúchať **listom**, t.j. že jednotlivé lexémy *id* budú ohodnotené a, b, c, d podľa toho, akú premennú predstavujú, priraďuje napr. lexikálny analyzátor.

Príklad *AST* č. 1

Ďalšie sémantické atribúty, ktoré by sa v strome mohli nachádzať, sú:

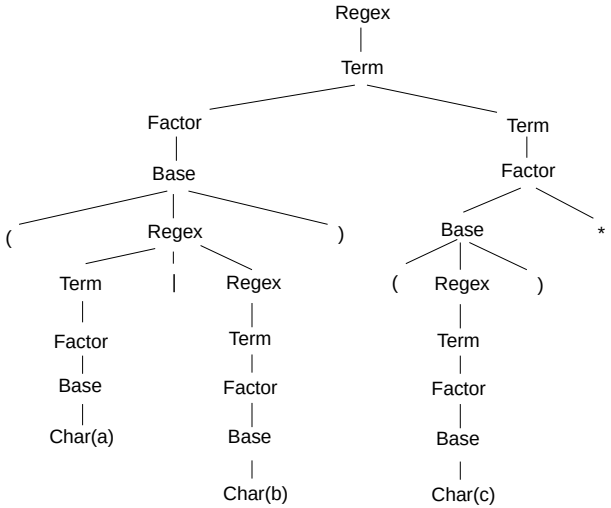
1. Keďže a, b, c, d sú premenné, z hľadiska generovania kódu je potrebné vediet informácie o adresách premenných v pamäti a ich typoch.
   - Tieto informácie sa nachádzajú v tzv. **tabuľke symbolov (TS)**.
   - Spravidla sa tam dostanú vtedy, ak sú premenné v programe pred ich použitím**deklarované** - vtedy dochádza k vytvoreniu záznamu o ich adrese a type do tabuľky symbolov.
2. Vyhľadanie identifikátorov v TS a ich typu a adresy.
3. Kontrola typu operandov a určenie typu výsledku - t.j. napr. overenie, či b a c sú rovnakého typu, aby ich násobenie $b * c$ dávalo v príslušnom jazyku zmysel.

Príklad *AST* č. 2

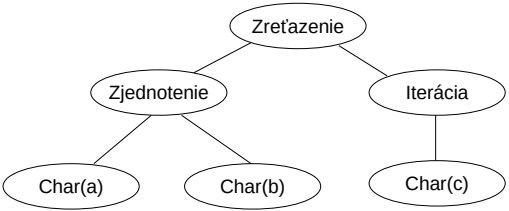
Uvažujme gramatiku pre parsovanie regulárnych výrazov, Regex je počiatočný neterminál, Char je terminál / lexéma reprezentujúca malé písmená abecedy:

1. Regex $\rightarrow$ Term
2. Regex $\rightarrow$ Term | Regex
3. Term $\rightarrow$ Factor Term
4. Term $\rightarrow$ Factor
5. Factor $\rightarrow$ Base
6. Factor $\rightarrow$ Base $*$
7. Base $\rightarrow$ Char
8. Base $\rightarrow$ (Regex)

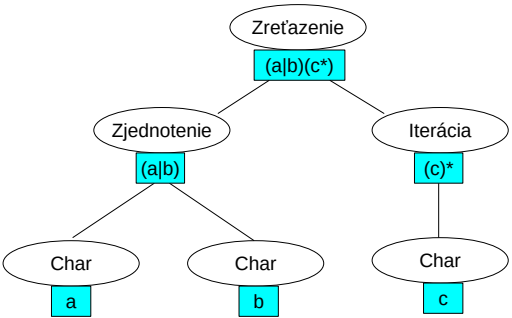
Výraz $(a|b)(c)^*$ má derivačný strom (terminály Char sú už ohodnotené sémantickými atribútmi podľa príslušného písmena abecedy):



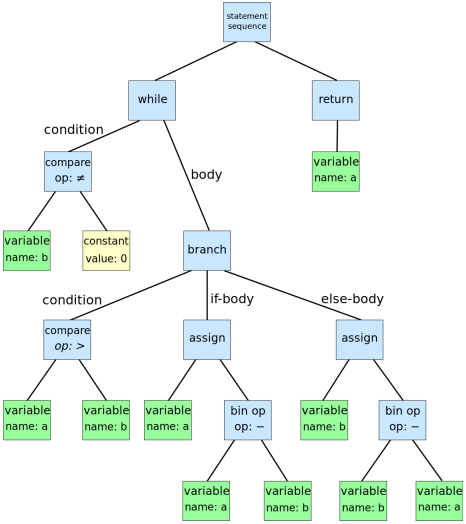
Príslušný abstraktný syntaktický strom by mohol vyzerať nasledovne:



Iný abstraktný syntaktický strom by mohol vyzerať nasledovne (sémantické atribúty sú teraz znázornené v modrých obdĺžnikoch a predstavujú regulárny výraz, ktorý je asociovaný s príslušným vrcholom AST).



Príklad *AST* č. 3 - GCD algoritmus



while b ≠ 0
  if a > b
    a := a – b
  else
    b := b – a
return a

Figure: Prevzaté z https://commons.wikimedia.org/wiki/File:Abstract_syntax_tree_for_Euclidean_algorithm.svg, Autor: Cmglee

Sémantická analýza

- Po spracovaní všetkých sémantických atribútov sa v prípade prekladačov pristúpi ku generovaniu kódu.
  1. Generuje sa alebo kód vo výslednom cieľovom jazyku.
  2. Alebo sa generuje tzv. medzikód.

Medzikód

- **Medzikód** je forma reprezentácie zdrojového kódu, resp. programu.
- Umožňuje pri kompilácii oddeliť spracovanie na vyššej úrovni (lexikálna, syntaktická, sémantická analýza) od spracovania na nižšej úrovni (generovanie cieľového kódu).
- Abstrahuje od platformovo-závislých operácií (práca s konkrétnymi fyzickými registrami procesora, konkrétnymi absolútnymi adresami v pamäti, atď.)
- Rozďeluje preklad (kompiláciu) na 2 hlavné časti - tzv. front-end a back-end.

Medzikód

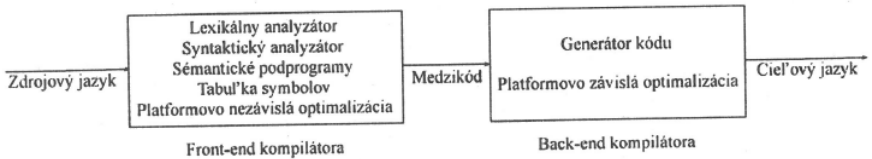


Figure: Prevzaté z Dedera, Ľ.: Počítačové jazyky a ich spracovanie

- Front-end - transformácia zdrojového kódu do medzikódu (a jeho prípadná optimalizácia)
- Back-end - transformácia medzikódu do cieľového jazyka (a prípadná optimalizácia na platformovej úrovni)

Medzikód

Jeho použitie má 2 hlavné výhody:

- Možnosť optimalizácie na úrovni medzikódu t.j. nezávislej od cieľového jazyka / platformy. Preto je táto optimalizácia ľahko prenositeľná medzi platformami a zväčša aj jednoduchšia ako optimalizácia na úrovni strojového jazyka.
- Jednoduchšia prenositeľnosť kompilátorov medzi jazykmi / platformami.

Medzikód

Vďaka použitiu medzikódu a rozdeleniu kompilátora na front-end a back-end je možné:

- Realizovať preklad zdrojového jazyka do medzikódu **bez ohľadu** na cieľovu platformu,
- realizovať preklad z medzikódu do cieľového jazyka **bez ohľadu** na zdrojový jazyk.

T.j. je možné vytvoriť kompilátor:

- existujúceho jazyka pre novú platformu - vytvorí sa **len** nový back-end kompilátora,
- nového jazyka pre existujúcu cieľovú platformu - vytvorí sa **len** nový front-end, ktorý vytvorí reprezentáciu v existujúcom medzikóde.

3 často uvažované formy medzikódu sú:

1. Abstraktný syntaktický strom
2. Postfixový zápis
3. Jazyk štvoríc (niekedy nazývaný aj trojadresný kód, resp. three-address code)

Postfixový zápis

- Získame ho, ak vypíšeme vrcholy AST v usporiadaní POSTORDER (t.j. rekurzívne najprv vrcholy ľavého podstromu, potom pravého podstromu a nakoniec koreň).
- Pre AST výrazu $a + b * c + d$ zo slajdu č. 11 by postfixový zápis bol $abc * +d+$.
- Pre AST regulárneho výrazu $(a|b)(c)*$ zo slajdu č. 16 by postfixový zápis bol (teoreticky) $ab|c*$, resp.
  a b Zjednotenie c Iterácia Zreťazenie

Postfixový zápis

- Jeho výhodou je, že umožňuje vyhodnotenie výrazu len s pomocou zásobníka.
- Postfixový zápis sa spracúva zľava-doprava, pričom ak je na vstupe **operand**, vloží sa do zásobníka a ak je na vstupe **operátor**, zo zásobníka sa vyberie príslušný počet operandov, operácia sa vyhodnotí a výsledok sa vloží do zásobníka.

Jazyk štvoríc

- Je tvorený inštrukciami, ktoré su charakterovo-blízke strojovému jazyku.
- Rozdiel je v tom, že operandy sú alebo konštanty, alebo premenné, nie registre.
- Používajú tzv. dočasné premenné (temporaries) na ukladanie medzivýsledkov - ich meno začína písmenom *t*.

Jazyk štvoríc - typické štvorice

Štvorica	Popis
(*operátor*, *op1*, *op2*, *výsledok*)	*výsledok* ← *op1* *operátor* *op2*, pričom *operátor* ∈ {+, −, *, /, AND, OR, =, ≠, <, <=, >, >=} (binárne aritmetické, logické a relačné operátory)
(−, *op1*, *výsledok*) (NOT, *op1*, *výsledok*)	*výsledok* ← −*op1* (unárne mínus) *výsledok* ← not(*op1*) (negácia)
(:=, *op1*, *op2* ,*výsledok*)	*výsledok* ← *op1*, *op2* voliteľne obsahuje počet prenesených bajtov (presun/priradenie)
(FLOAT, *op1*,*výsledok*)	*výsledok* ← Float(*op1*) (konverzia na typ Float)
(INTEGER, *op1*,*výsledok*)	*výsledok* ← Integer(*op1*) (konverzia na typ Integer)
(LABEL, *návestie*)	*návestie* sa použije ako návestie nasledujúcej inštrukcie
(JUMP, *návestie*)	nepodmienený skok na inštrukciu s návestím *návestie*
(JUMPT, *op1*, *návestie*)	podmienený skok na inštrukciu s návestím *návestie*, ak *op1* obsahuje logickú hodnotu TRUE
(JUMPF, *op1*, *návestie*)	podmienený skok na inštrukciu s návestím *návestie*, ak *op1* obsahuje logickú hodnotu FALSE
(RANGETEST, *dol*, *hor*, *op1*)	testuje, či $dol \leq op1 \leq hor$
(INDEX, *pole*, *i*, *prvok*)	*prvok* ← adresa prvku poľa *pole*[*i*]

Jazyk štvoríc - príklad

Výraz $a + b * c + d$ by zapísaný v jazyku štvoríc vyzeral:

$(*, b, c, t1)$

$(+, a, t1, t2)$

$(+, t2, d, t3)$

V dočasnej premennej $t3$ sa tak nachádza výsledná hodnota výrazu (v závislosti od hodnôt v premenných a, b, c, d. Predpokladali sme, že sú všetky rovnakého typu.

Jazyk štvoríc - príklad č. 2

Výraz $D := b * b - 4 * a * c$, kde D, b, c sú typu Float, a je typu Integer, 4 je konštanta typu Int, by zapísaný v jazyku štvoríc vyzeral:

$(*, b, b, t1)$

$(\text{FLOAT}, 4, t2)$

$(-, t2, t3)$

$(\text{FLOAT}, a, t4)$

$(*, t3, t4, t5)$

$(*, t5, c, t6)$

$(+, t1, t6, t7)$

$(:=, t7, \text{sizeof(Float)}, D)$

Sémantické podprogramy

- Uvažujme konštrukciu **jednoprechodového kompilátora**.
- Jeho ústrednou časťou je syntaktický analyzátor, ktorý podľa potreby volá lexikálny analyzátor a sémantické podprogramy.
- Sémantické podprogramy sú programy, ktorých cieľom je interpretovať syntaktickú štruktúru programu a na základe interpretácie generovať medzikód alebo cieľový kód.
- Napríklad v prípade, že syntaktický analyzátor rozpoznal podmienený príkaz, kompilátor zavolá sémantický podprogram na jeho spracovanie.
- Jednotlivé sémantické podprogramy by mali medzi sebou komunikovať a vymieňať si dáta.

Sémantické podprogramy

- Každý syntaktický analyzátor sa riadi gramatikou, pre ktorú bol zostrojený.
- Vyvolanie **sémantických podprogramov** vo vhodnom mieste sa docieli vkladaním tzv. **akčných symbolov** do pravidiel gramatiky.
- **Akčné symboly** nie sú ani terminálne, ani neterminálne symboly gramatiky - budeme ich preto označovač prefixom #.
- Ak syntaktický analyzátor narazí na symbol označený #, vyvolá príslušný sémantický podprogram.

Sémantické spracovanie - príklad

Chceme spracovať zjednodušený podmienený príkaz. Nech je gramaticky daný:

<príkaz> → `if`<výraz>`then`<príkaz>

Jeho spracovanie by prebehlo nasledovne:

1. Vyhodnotí sa podmienka, t.j. výraz generovaný z neterminálu <výraz>
2. Na základe hodnoty výrazu (uloží sa do nejakej dočasnej premennej `t1`) sa - ak nie je splnená - generuje skok za celý príkaz (výraz generovaný z neterminálu <príkaz> za `then`) na nejaké jedinečné návestie (label) `n1`
3. Vygeneruje sa kód pre príkaz za `then`.
4. Po spracovaní celého podmieneného príkazu nasleduje definícia návestia `n1`.

Sémantické spracovanie - príklad

Ekvivalentný zápis s využitím jazyku štvoríc:

1. Kód vyhodnocujúci podmienku, pričom do premennej `t1` uloží jej výslednú hodnotu.
2. (JUMPF, `t1`, `n1`)
3. Kód príkazu za `then`
4. (LABEL, `n1`)

Sémantické spracovanie - príklad

Konkrétny príklad v assembleri:

C

```c
// Type your code here, or load an example.
int main()
{
    int a = 1;
    if (a == 0) a = a+1;
    a = 2;
}
```

Assembler

```
main:
        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR [rbp-4], 1
        cmp     DWORD PTR [rbp-4], 0
        jne     .L2
        add     DWORD PTR [rbp-4], 1
.L2:
        mov     DWORD PTR [rbp-4], 2
        mov     eax, 0
        pop     rbp
        ret
```

Figure: Zdroj: `https://godbolt.org/`

Sémantické spracovanie - príklad

- Ak by sme teda chceli preložiť program, v ktorom je príslušný podmienený príkaz, tak vidíme, že:
  1. Po vyhodnotení podmienky musíme vyvolať sémantický podprogram (ozn. ho `#IF_PODMIENKA`), ktorý zabezpečí generovanie podmieneného skoku za časť programu reprezentovanú neterminálom <príkaz> za `then`.
  2. Po spracovaní celého podmieneného príkazu potrebujeme vyvolať druhý sémantický podprogram (ozn. ho `#IF_KONIEC`), ktorý zabezpečí vloženie návestia, od ktorého sa pokračuje v prípade, že podmienka nie je splnená.

Sémantické spracovanie - príklad

Príslušné pravidlo gramatiky teda upravíme vložením akčných symbolov:

<príkaz> → `if`<výraz>`#IF_PODMIENKA then`<príkaz>`#IF_KONIEC`

- Keď teraz syntaktický analyzátor narazí na akčný symbol `#IF_PODMIENKA` alebo `#IF_KONIEC`, vyvolá príslušný sémantický podprogram.
- Sémantický podprogram `#IF_PODMIENKA` musí vedieť, kde sa nachádza výsledná hodnota podmienky výraz (t.j. premenná `t1` na slajdoch 33,34).
- Oba podprogramy musia medzi sebou komunikovať v tom, kde `#IF_PODMIENKA` zanechá identifikátor návestia, na ktoré generoval skok v prípade nepravdivej podmienky a ktoré musí `#IF_KONIEC` do cieľového kódu zaviesť (t.j. návestie `n1` na slajdoch 33,34).

Sémantické podprogramy

- Komunikáciu sémantických podprogramov, t.j. odovzdávanie hodnôt sémantických atribútov medzi nimi, sa zabezpečuje pomocou tzv. **sémantických záznamov** prislúchajúcich symbolom zo syntaktického zásobníka, resp. derivačného stromu.
- **Sémantický záznam (SZ)** je údajová štruktúra obsahujúca údaje (sémantické atribúty) ľubovoľného typu potrebné pre komunikáciu medzi sémantickými podprogramami.

Sémantické záznamy

Definícia

Sémantický záznam zodpovedajúci ľavej strane pravidla označujeme symbolom $$ *a sémantické záznamy terminálnych a neterminálnych symbolov pravej strany pravidla označujeme zľava-doprava symbolmi* $1, $2, ..., $*n*, *pričom n je dĺžka pravej strany pravidla.*

Sémantické spracovanie - Príklad

Uvažujme nasledovnú gramatiku:

1. <program> → `begin`<príkazy>`end`
2. <príkazy> → <príkaz>`;`<príkazy>
3. <príkazy> → ε
4. <príkaz> → `p`

Nech úloha sémantického spracovania je zistiť, koľko *príkazov* obsahuje *program* - t.j. spočítať výskyt terminálu `p` vo vstupnom reťazci.

Sémantické spracovanie - Príklad

Doplníme do gramatiky akčné symboly `#PROGRAM`, `#PRÍKAZY0`, `#PRÍKAZY1` nasledovne:

1. <program> → `begin`<príkazy>`end #PROGRAM`
2. <príkazy> → <príkaz>`;`<príkazy> `#PRÍKAZY1`
3. <príkazy> → ε `#PRÍKAZY0`
4. <príkaz> → `p`

V sémantickom zázname budeme ukladať informáciu o počte volaní príkazov `p`.

Sémantické podprogramy v Príklade - PROGRAM

Sémantický podprogram PROGRAM sa vyvolá na konci spracovaného reťazca a vráti výsledný počet príkazov p, ktorý sa nachádza v SZ asociovanom s <príkazy>, t.j. v $2:

funkcia PROGRAM
    vráť ”Program obsahuje” $2 ”príkazov.”
koniec funkcia

Sémantické podprogramy v Príklade - PRÍKAZY1

Sémantický podprogram PRÍKAZY1 sa vyvolá na konci 2. pravidla reťazca. V rámci tohoto pravidla sa generuje nový príkaz, preto sémantický podprogram zvýši počet príkazov generovaných z <príkazy> na pravej strane pravidla ($3) o 1 a uloží ho v SZ asociovanom s neterminálom <príkazy> na ľavej strane, t.j. v $$:

funkcia PRÍKAZY1
    $$ \leftarrow $3 + 1;
koniec funkcia

Sémantické podprogramy v Príklade - PRÍKAZY0

Sémantický podprogram PRÍKAZY0 sa vyvolá na konci 3. pravidla reťazca. V rámci tohoto pravidla sa negeneruje nový príkaz, preto sémantický podprogram uloží **nulu** do SZ asociovaného s neterminálom <príkazy> na ľavej strane pravidla, t.j. $$:

funkcia PRÍKAZY0
    $$ ← 0;
koniec funkcia

Sémantické spracovanie - príklad

Ako bude prebiehať spracovanie vstupu: `begin p ; p ; end`

Znázornenie sémantického spracovania

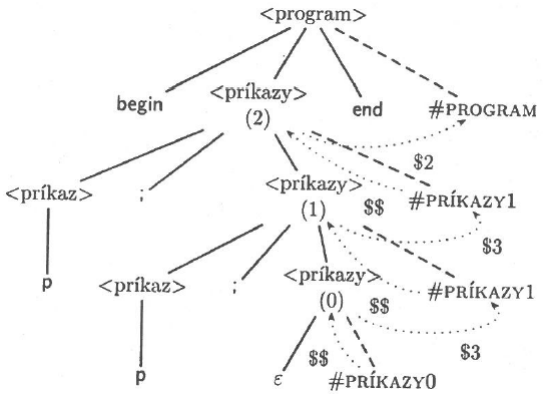


Figure: Prevzaté z Dedera, Ľ.: Počítačové jazyky a ich spracovanie

Výsledok príkladu

- Je dôležité si uvedomiť, že sémantiku sme si zvolili my, t.j. chceli sme spočítať, koľkokrát sa vo vstupnom reťazci ”nachádza volanie programu”, t.j. terminál *p*.
- Na to slúžili sémantické podprogramy PROGRAM, PRÍKAZY1, PRÍKAZY0.
- Všimnite si, že symboly gramatiky v sémantickom zásobníku medzi sebou komunikujú prostredníctvom svojich sémantických záznamov!
- Napr. pri redukcii podľa pravidla: <príkazy> $\rightarrow$ <príkaz>; <príkazy> sú vlastne príslušné SZ označené ako \$\$ $\rightarrow$ \$1 \$2 \$3.

Výsledok príkladu

- Preto ak je v sémantickom podprograme PRÍKAZY1 napísané, že $$ = $3 + 1, znamená to, že do SZ pre symbol $$ sa vloží hodnota SZ pre symbol $3 zvýšená o 1.

Sémantické spracovanie - Kalkulačka

Nech je daná gramatika:

$$S \to E$$
$$E \to E + T \mid E - T \mid T$$
$$T \to T * F \mid T/F \mid F$$
$$F \to (E) \mid k$$

Lexikálny analyzátor rozpozná ako terminál k číselnú konštantu, ktorej hodnotu vie vrátiť. Príslušná gramatika je $SLR(1)$.

Sémantické spracovanie - Kalkulačka

- Úlohou sémantickej analýzy bude v tomto príklade **vyhodnotiť výraz**.
- T.j. zadaný výraz sa rozparsuje a pre jednotlivé vrcholy derivačného stromu budú pridelené sémantické atribúty znamenajúce hodnotu podvýrazu.
- Hodnota sémantického atribútu pri koreni derivačného stromu bude potom hodnota celého výrazu.
- Výraz sa bude vyhodnocovať pomocou sémantických podprogramov.
- Akčné symboly sú vložené na **konci každého pravidla**, pričom im prislúchajú nasledovné sémantické akcie:

Sémantické spracovanie - Kalkulačka

Č.	Pravidlo	Sémantická akcia
1	$S \to E$	Výsledok výrazu je \$1.
2	$E \to E + T$	$\$\$ \leftarrow \$1 + \$3$
3	$E \to E - T$	$\$\$ \leftarrow \$1 - \$3$
4	$E \to T$	$\$\$ \leftarrow \$1$
5	$T \to T * F$	$\$\$ \leftarrow \$1 * \$3$
6	$T \to T/F$	$\$\$ \leftarrow \$1/\$3$
7	$T \to F$	$\$\$ \leftarrow \$1$
8	$F \to (E)$	$\$\$ \leftarrow \$2$
9	$F \to k$	\$\$ ← konštanta vrátená lexikálnym analyzátorom

Sémantické spracovanie - Kalkulačka

Simulujme výpočet pre vstupný reťazec $5 + 3 * 6$ predstavujúci matematický výraz.

- Lexikálny analyzátor (LexA) rozdelí reťazec $5 + 3 * 6$ na postupnosť lexém $k + k * k$.
- Syntaktický analyzátor (SynA) nájde derivačný strom postupnosti $k + k * k$.
- Sémantický analyzátor (SemA) vyhodnotí príslušné sémantické atribúty a nájde hodnotu vstupného výrazu (t.j. mal by vrátiť hodnotu 23).

Obrázok sémantického spracovania podľa derivačného stromu:

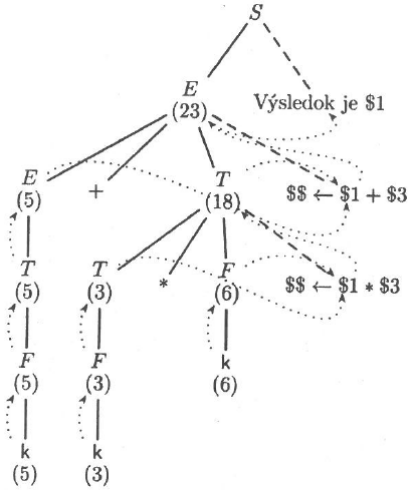


Figure: Prevzaté z Dedera, Ľ.: Počítačové jazyky a ich spracovanie

Tvorba jazykových procesorov pomocou nástrojov Flex a Bison

- Jazykový procesor, resp. kompilátor so syntaktickým analyzátorom ako ústrednou časťou, je možné vytvárať aj pomocou špeciálnych programov - generátorov jazykového procesora.
- Generátor jazykového procesora vytvorí jazykový procesor, ktorý obsahuje:
  - Lexikálny analyzátor na základe popisu lexém,
  - Syntaktický analyzátor na základe popisu bezkontextovej gramatiky,
  - Správu volaní sémantických podprogramov,
  - Správu sémantického zásobníka.
- Typický nástroj pre konštrukciu lexikálneho analyzátora je **Flex** (viď. prednáška č. 6).
- Typický generátor jazykového procesora je nástroj **Bison.**

Nástroj Bison

- Bison predstavuje generátor jazykového procesora vychádzajúci z nástroja Yacc.
- Generuje syntaktický analyzátor LALR(1), konflikty rieši pomocou priority asociatívnosti operátorov, resp. aj GLR prístupom.
- Predstavuje teda príklad sémantického spracovania pri analýze zdola-nahor.
- Spolupracuje s nástrojom Flex.
- Jeho vstupný súbor (zvyčajne s príponou `.y`) obsahuje popis gramatiky, ku ktorej sa jazykový procesor tvorí, spolu s časťami kódu v jazyku C, ktoré sa vykonajú, keď sa rozpozná pravá strana príslušného pravidla (sémantické podprogramy).

Nástroj Bison

- Tieto fragmenty kódu (sémantické podprogramy) používajú symboliku $$, $1, ... na prístup k sémantickým záznamom sémantického zásobníka.
- Výstupom z nástroja Bison je kód jazykového procesora v jazyku C v 2 súboroch s koncovkami: `.tab.c` a `.tab.h`. Tie sa dajú skompilovať kompilátorom jazyka C a prilinkovať k ďalším programom.

Nástroj Bison - vstupný súbor

Vstup nástroja Bison predstavuje súbor s koncovkou `.y` obsahujúci tieto 3 časti:

1. Sekciu deklarácií
2. Sekciu pravidiel
3. Sekciu podprogramov

Tieto sekcie sú navzájom oddelené %%. Ich obsah si priblížime na príklade.

Nástroj Bison - príklad

- Zostrojme jazykový procesor, ktorý vykoná spracovanie gramatiky zo slajdu č. 40 s rovnakou sémantickou analýzou, t.j. spočíta, koľko ”príkazov” `p` obsahuje ”program”, t.j. vstupný reťazec. Gramatika:
  1. <program> $\rightarrow$ `begin`<príkazy>`end`
  2. <príkazy> $\rightarrow$ <príkaz>`;`<príkazy>
  3. <príkazy> $\rightarrow \varepsilon$
  4. <príkaz> $\rightarrow$ `p`

Bison - sekcia deklarácií

Sekcia deklarácií vstupu nástroja Bison môže obsahovať:

- Kód (v jazyku C), ktorý sa vloží do výsledného súboru na jeho začiatok.
- Definíciu údajového typu sémantického záznamu - konštanta `YYSTYPE`. Ak sa nedefinuje, implicitne je typu `int`.
- Definície terminálnych symbolov gramatiky cez klauzulu `%token`
- Špecifikáciu počiatočného neterminálu gramatiky cez klauzulu `%start`
- Definíciu externej premennej `yytext` ktorá obsahuje reťazec získaný na vstupe zodpovedajúci príslušnej vrátenej lexéme.

Bison - sekcia deklarácií

```
%{
#include<stdio.h>
#define YYSTYPE int
extern char *yytext;
void vypis(YYSTYPE argument);
%}

%token _BEGIN
%token _END
%token _SEMICOLON
%token _P
%token NEZNAME

%start PROGRAM
%%
```

Bison - sekcia pravidiel

- Pravidlá sa zadávajú v tvare:

```
A: B1 ...  BN
 | C1 ...  CN
...
 ;
```

- pričom `A` je ľavá strana pravidiel, `B1 ... BN` a `C1 ... CN` sú pravé strany pravidla s rovnakou ľavou stranou. Všimnite si zvislú čiaru pri novom pravidle s rovnakou ľavou stranou a bodkočiarku, ktorá ukončuje pravidlá s rovnakou ľavou stranou.
- Všetko, čo nebolo deklarované ako `%token` v predchádzajúcej časti, je neterminál.
- Symboly **musia** byť oddelené medzerami.

Bison - sekcia pravidiel

- Pravidlá sa zadávajú v tvare:

```
A: B1 ...  BN
 | C1 ...  CN
...
 ;
```

- pričom `A` je ľavá strana pravidiel, `B1 ... BN` a `C1 ... CN` sú pravé strany pravidla s rovnakou ľavou stranou. Všimnite si zvislú čiaru pri novom pravidle s rovnakou ľavou stranou a bodkočiarku, ktorá ukončuje pravidlá s rovnakou ľavou stranou.
- Všetko, čo nebolo deklarované ako `%token` v predchádzajúcej časti, je neterminál.
- Symboly **musia** byť oddelené medzerami.

Bison - sekcia pravidiel

- Do pravidiel sa môžu vkladať kusy kódu v jazyku C uzatvorené do $\{...\}$, ktorý sa vykoná, keď bude rozpoznaná časť pravej strany pravidla pred blokom ohraničeným $\{...\}$.
- T.j. ak sa takýto blok kódu vloží na koniec pravidla, tak sa vykoná v prípade redukcie podľa príslušného pravidla.
- V podstate to nahrádza **akčné symboly**, resp. príslušné sémantické podprogramy.
- V sémantických podprogramoch je možné využívať:
  - Premenné `yytext` a `yyleng` z lexikálneho analyzátora (rozpoznaný reťazec pre danú lexému, resp. jeho dĺžka
  - Premenné $\$\$, \$1, ...$ pre symboly v pravidle, resp. príslušné sémantické záznamy v sémantickom zásobníku.
  - Volania funkcií definovaných v sekcii podprogramov.

Bison - sekcia pravidiel

```
PROGRAM:  _BEGIN PRIKAZY _END  {vypis($2);}
 ;
PRIKAZY: PRIKAZ _SEMICOLON PRIKAZY {$$ = $3 + 1;}
 | /**/ {$$ = 0;}
 ;
PRIKAZ: _P
 ;
%%
```

Bison - sekcia podprogramov

- Tretia sekcia vstupného súboru nástroja Bison obsahuje definície funkcií, ktoré bude obsahovať výsledný jazykový procesor (výstup nástroja Bison).
- Sú tu teda aj definované funkcie, ktoré môžu byť volané v rámci sémantických podprogramov použitých v sekcii pravidiel.
- Samotný syntaktický analyzátor sa volá pomocou funkcie `yyparse()`, ktorá realizuje kompletnú syntaktickú a sémantickú analýzu. V prípade úspešnej syntaktickej analýzy vráti hodnotu 0 (nula), v prípade syntaktickej chyby vráti hodnotu 1.
- Taktiež je vhodné tu vložiť funkciu main(), v ktorej sa môže zavolať samotný analyzátor cez volanie `parse()`.

Bison - sekcia podprogramov

```
void vypis(YYSTYPE argument)
{
 printf("Program obsahuje %d prikazov\n",argument);
}

void main()
{
 if (yyparse() == 0)
  printf("Vyraz bol syntakticky spravny!\n");
 else
  printf("Vyraz nebol syntakticky spravny!\n");
 return ;
}
```

Bison - vstup a výstup

- Vstupný súbor teda pozostáva z uvedených troch častí - ak ich spojíme dokopy a výsledok nazveme `program.y`, tak spracovanie vstupného súboru pomocou nástroja Bison vykonáme:

  ```
  Bison -d -v program.y
  ```

- Prepínače `-d` a `-v` vygenerujú hlavičkový súbor `.tab.h`, resp. generujú viacero výpisov.
- Výsledkom sú súbory `program.tab.c` a `program.tab.h` obsahujúce kód výsledného analyzátora v jazyku C.
- Bison taktiež vygeneruje súbor `program.output` obsahujúci popis stavov automatu syntaktického analyzátora, ktorý rozpoznáva pravé strany pravidiel (viď. $LR(0)$-automat).

Použitie Flex a Bison

- Následne môžeme vygenerovaný súbor `program.tab.h` použiť vo vstupnom súbore pre program Flex, ktorý generuje lexikálny analyzátor. Súbor `program.tab.h` totiž obsahuje definície tokenov (lexém), ktoré bude lexikálny analyzátor rozpoznávať.

Flex - vstupný súbor využívajúci výstup Bisonu

```
%{
#include "program.tab.h"
%}
%%

[\t ]+
";" return (_SEMICOLON);
"p" return (_P);
[b][e][g][i][n] return (_BEGIN);
[e][n][d] return (_END);
[\n] yyterminate();
. return(NEZNAME);
%%
```

Flex - vstupný súbor využívajúci výstup Bisonu

- Vidíme, že do kódu lexikálneho analyzátora bude vložený príkaz `#include "program.tab.h"`, ktorým sa zabezpečí spojenie so syntaktickým analyzátorom, resp. tokenmi (lexémami) v ňom definovanými
- riadok `[\n] yyterminate()` spôsobí, že keď lexikálny analyzátor natrafí na koniec riadku (alebo je stlačený enter pri manuálnom zadávaní), tak v tom momente končí lexikálnu analýzu

Bison a Flex

- Spustíme Flex so vstupným Flex súborom (nech sa volá `program.l`)

  `Flex program.l`
- Flex vygeneruje kód lexikálneho analyzátora `lex.yy.c`
- Súbory `lex.yy.c, program.tab.c, program.tab.h` spolu tvoria výsledný C-kód jazykového procesora.
- Podobných generátorov jazykových procesorov je viacero, pekný prehľad s použitými syntaktickými analyzátormi a s programovacím jazykom výstupu je na `https://en.wikipedia.org/wiki/Comparison_of_parser_generators`.

Použitá literatúra

Aho, A., Lam, M., Sethi, R., Ullman, J.: *Compilers: Principles, techniques and tools.*

Dedera, Ľ: *Počítačové jazyky a ich spracovanie*.

Linz, P.: *An Introduction to Formal Languages and Automata*.