Kompilátory

RNDr. Ján Šturc, CSc.

Katedra informatiky MFF UK

marec 1998 (ver 1.0)

Literatúra:

A.V.Aho, R.Sethi a J.D.Ullman:
Compilers Principles, Techniques and Tools.
Addison Wesley, Reading Mass. 1986.

A.V.Aho, J.D.Ullman:
Compiler Design.
Addison Wesley 1977.

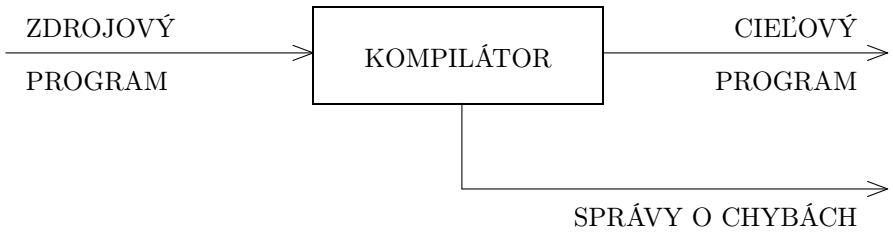
A.V.Aho, J.D.Ullman:
The theory of Parsing, Translation and Compiling.
Vol. I. Parsing Prentice Hall 1972
Vol. II. Compiling Englewood Cliffs N.J. 1973

C. N. Fisher, R. J. Leblanc:
Crafting a Compiler
The Benjamin / Cummings Publishing Company, Inc, 1988.

Ľ. Molnár, M. Češka, B. Melichar:
Gramatiky a jazyky.
Alfa, SNTL Bratislava 1987.

Upozornenie: *Nasledujúci text nie je učebnica ani učebný tex, ale len poznámky k prednáške. Obsahuje väčšinou len to, čo počas prednášky napíšem na tabuľu alebo premietnem. Hlavným jeho účeľom je ušetriť zdržujúce opisovanie počas prednášky.*

1 Úvod



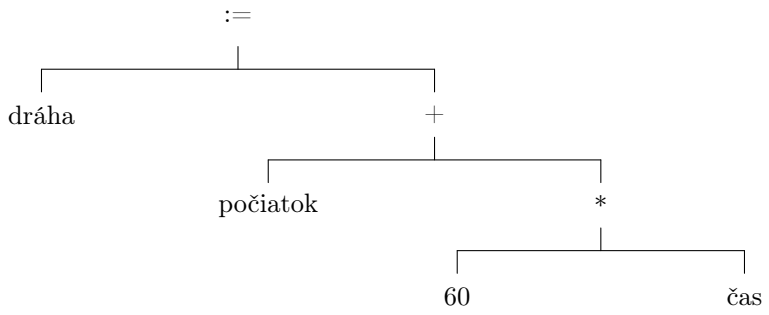
Prvá predstava kompilátora.

Prečo?

N1: Všetky kompilátory sú už hotové, je to zbytočné.
O1: Celý rad aplikácii používa nejakú formu analýzy.
Napr.:
- štrukturálne editory
- tlačové programy (type-seter)
- statické kontrolóry
- formátory textu
- kremíkové kompilátory
- interpretátory databázových dotazov

atď.

N2: Aj to si väčšinou kupujeme.
O2: Navyše kompilátor je exemplárny príklad veľkého programu, pre ktorý sa našli efektívne techniky. Realizácia kompilátorov prvých jazykov FORTRAN, Algol 60, COBOL trvala roky veľkým pracovným týmom. Vďaka pokroku teórie dnes môže realizovať aj študent ako ročníkovú prácu. Je asi užitočné osvojiť si techniky, ktoré k tomu viedli.

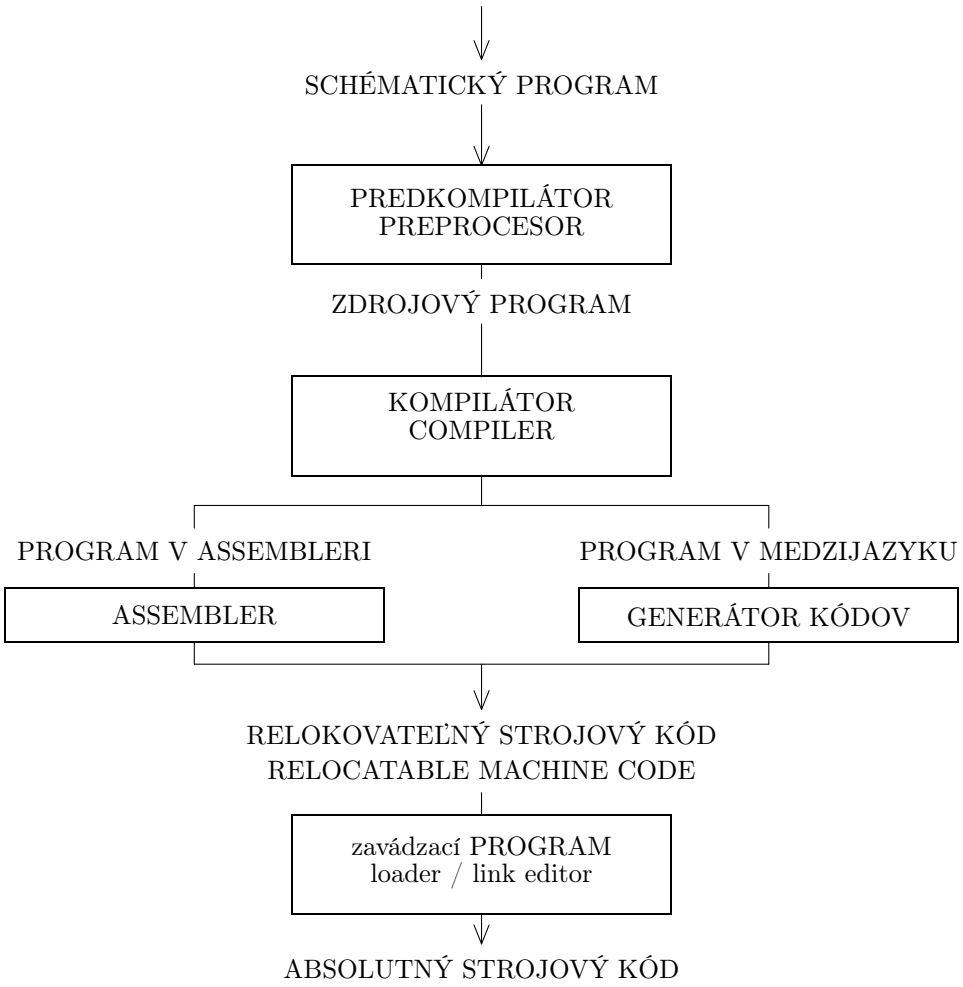


Analýza štruktúry (vytváranie stromu) je základ kompilácie.

Predpoklady:

I. Formálne jazyky.
II. Praktická znalosť nejakého programovacieho jazyka.
III. Assembler.
(model počítača) RAM.
RASP.

1.1 Spracovanie jazyka



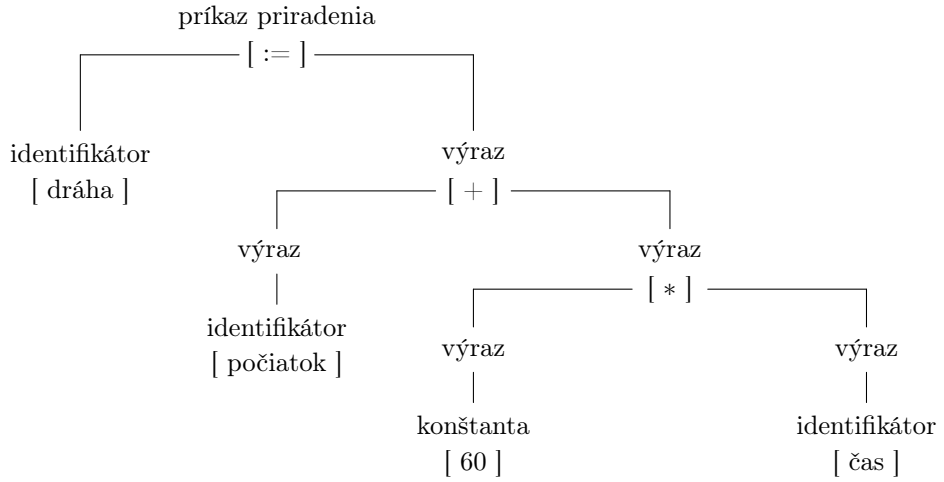
Zdroje príkladov:

- Klasické programovacie jazyky: FORTRAN, ALGOL 60, PL/I
- Moderné programovcacie jazyky: Pascal, C, Modula
- Databázové a logické jazyky: SQL, Prolog, Trilogy
- Tlačové systémy a editory: TEX, LATEX, EQN

Príklad: Ako pracuje kompilátor.

1. identifikátor	*dráha*	lexikálna analýza lexikálny analyzátor – scanner
2. symbol priradenia	*:=*	
3. identifikátor	*počiatok*	
4. operátor sčítania	*+*	
5. konštanta (číslo)	*60*	
6. operátor násobenia	***	
7. identifikátor	*čas*	

Syntaktická analýza (parsing)



Bezkontextová gramatika:

$$\begin{aligned} &< \text{výraz} > \quad ::= \quad && < \text{identifikátor} > \mid < \text{konštanta} > \mid \\ &&& < \text{výraz} > + < \text{výraz} > \mid < \text{výraz} > * < \text{výraz} > \end{aligned}$$

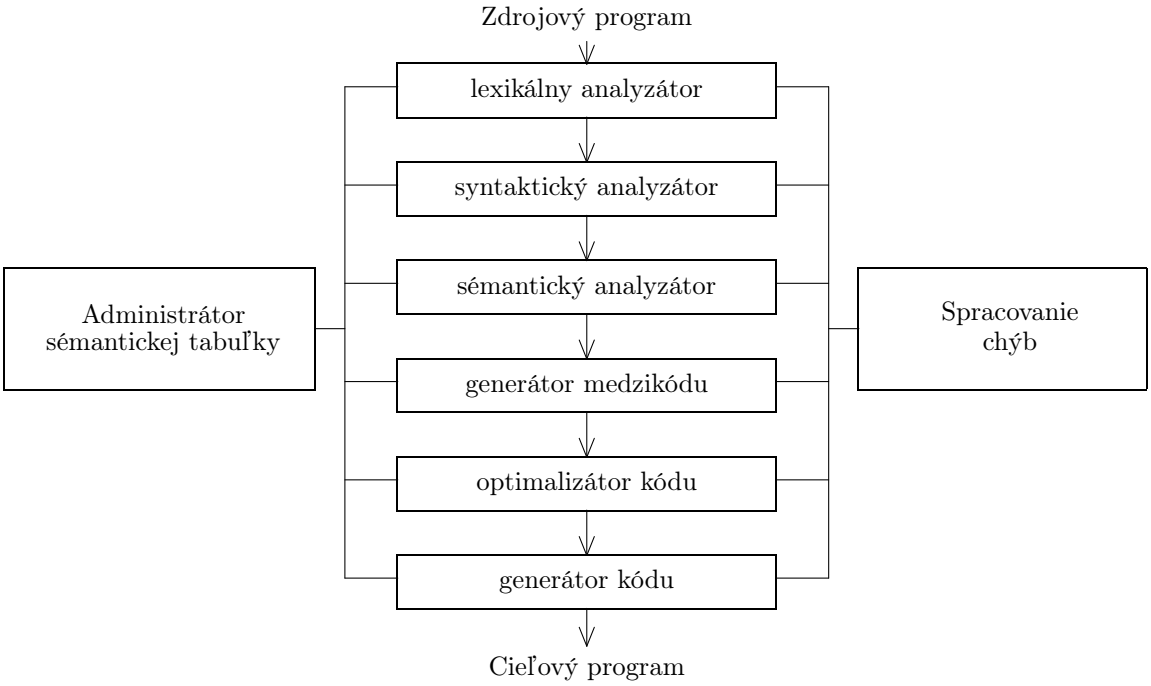
Rozdelenie práce medzi lexikálnu a syntaktickú analýzu nie je presne určené.
Spravidla: scanner – regulárna gramatika
parser – bezkontextová gramatika

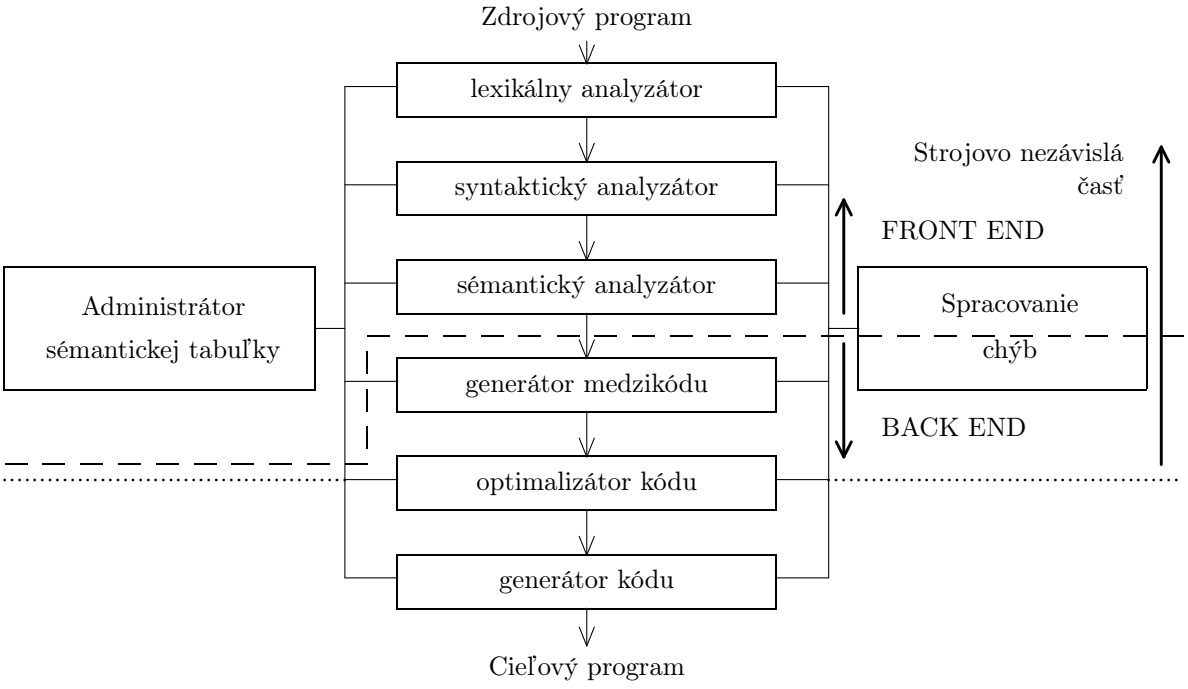
Sémantická analýza.

Kontroluje sémantické chyby.
Zbiera informácie o typoch. Typová analýza.

Príklad: V našom výraze konštanta 60 má byť reálna (vzhľadom na typy ostatných identifikátorov).
Rozdelenie práce medzi syntaktickú a sémantickú analýzu je trochu ľubovolné.
V zásade sémantika je to, čo sa nedá popísať bezkontextovou gramatikou.

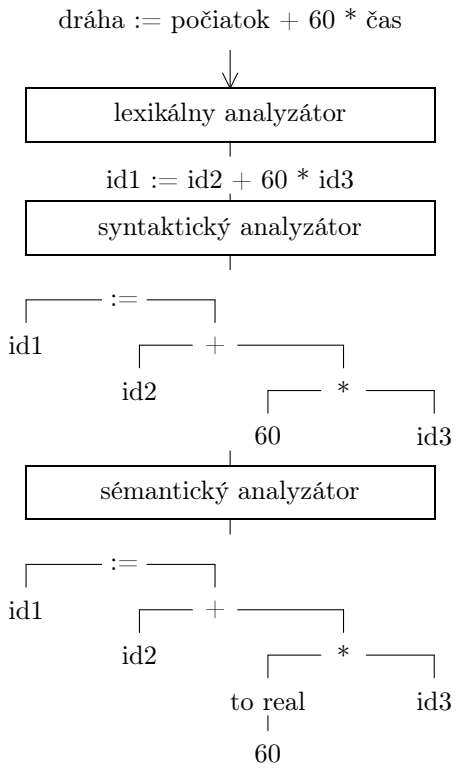
1.2 Podrobnejšia štruktúra kompilátora

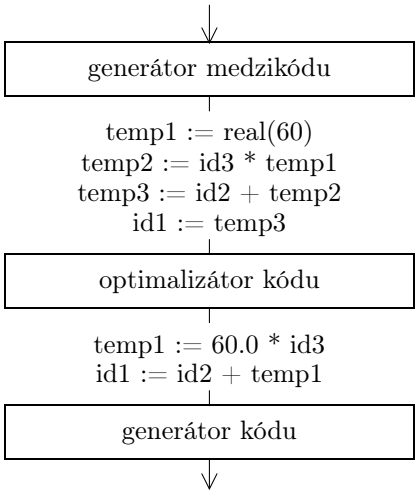




Tabuľka symbolov

dráha	...
počiatok	...
60	...
čas	...
⋮	





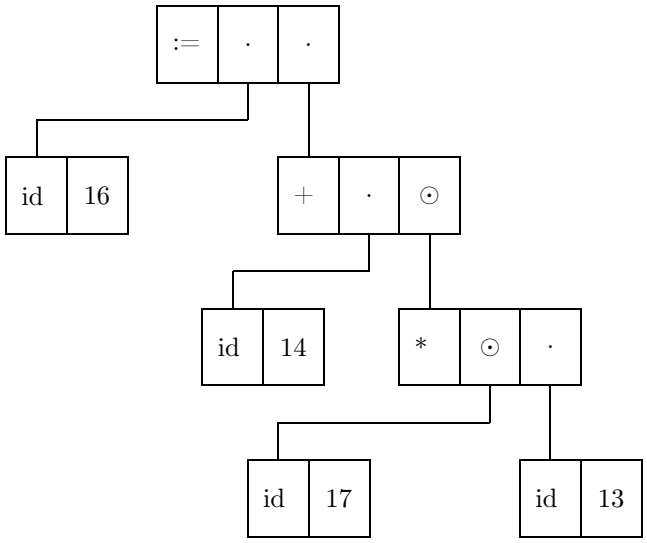

```
MOVF id3, R2
MULTF #60.0, R2
MOVF id2, R1
ADDF R1, R2
MOVF R1, id1
```

```
LOAD A id3
MULT A #60.0
ADD A id2
STORE A id2
```

Dátové štruktúry.

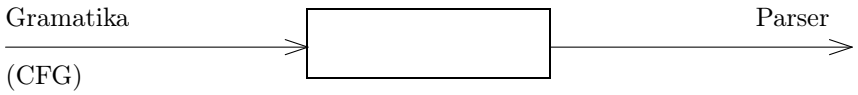
Tabuľka symbolov: hašovanie
Syntaktické stromy: smerníky.

	⋮
12	dráha
13	čas
14	počiatok
	⋮
17	60
	⋮

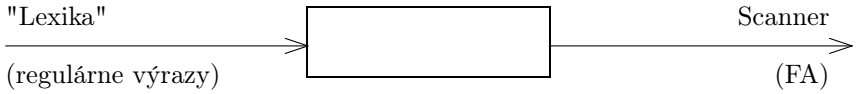


1.3 Nástroje pre tvorbu kompilátorov

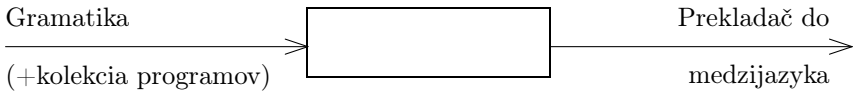
1. Generátory syntaktických analyzátorov (TWS, Compiler-compiler)



2. Generátory lexikálnych analyzátorov



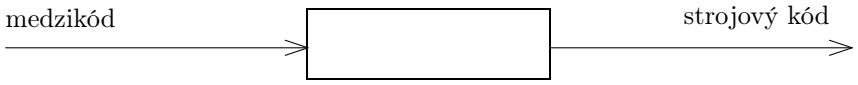
3. Syntaxou riadený prekladač



(TWS, Compiler–compiler)

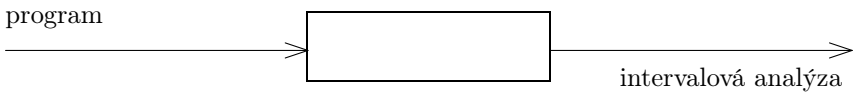
Front-end: strojovo nezávislá časť 1, 2, 3.

4. Generátory kódu



Technika: šablóny (templates), makrá pre inštrukcie medzikódu.

5. Analyzátory toku dát

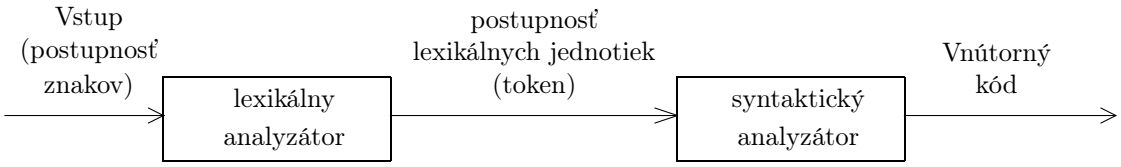


intervalová analýza
používanie premenných
informácie pre optimalizáciu

Back-end: strojovo závislé časti 4, 5

Prechody: Veľa prechodov (čas na zapisovanie a čítanie)
Málo prechodov (náročné na pamäť, veľké časti programu musia byť v pamäti)
Jedno-prechodové kompilátory – prakticky celý kompilátor je v pamäti.

2 Jednoduchý prekladač.



Bezkontextová gramatika $G = (T, N, R, \sigma)$

T - terminálne symboly (tokens)
N - neterminálne symboly (variables)
R - množina pravidiel (rules)
σ - počiatočný symbol $\sigma \in N$

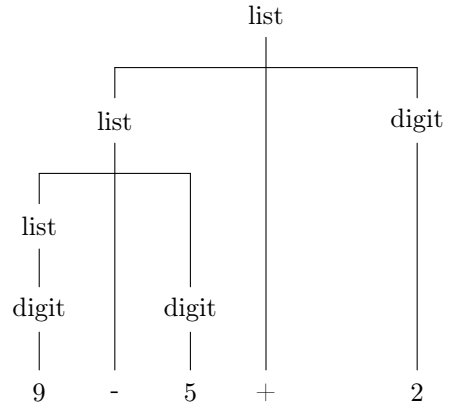
Príklad:

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

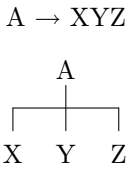
Príklad:

$9 - 5 + 2$

Syntaktický strom:



1. koreň
2. list - token
3. vnútorný uzol - premenná
4. pravidlo:



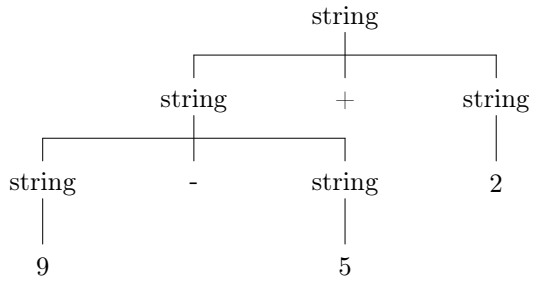
Nejednoznačnosť (ambiguity).

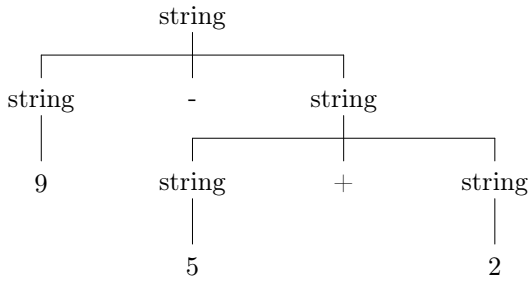
Definícia 2.1 *Gramatiku nazývame nejednoznačnou, ak nejakému slovu jazyka existujú dva neizomorfné stromy odvodenia.*

Ak používame nejodnoznačnú gramatiku pre popis syntaxe, musíme nejednoznačnosť vyriešiť na úrovni sémantiky – pomocou atribútov alebo sémantických programov.

Príklad:

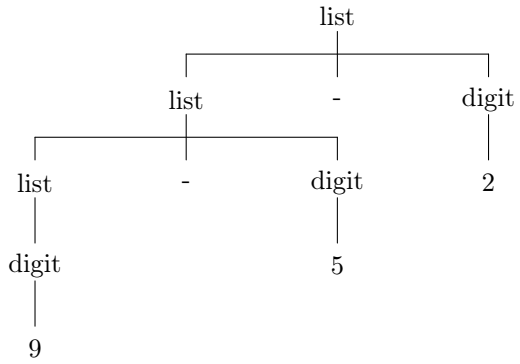
$$\begin{array}{rcl} \mathit{string} & \rightarrow & \mathit{string} + \mathit{string} \mid \mathit{string} - \mathit{string} \mid \\ & & 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \end{array}$$

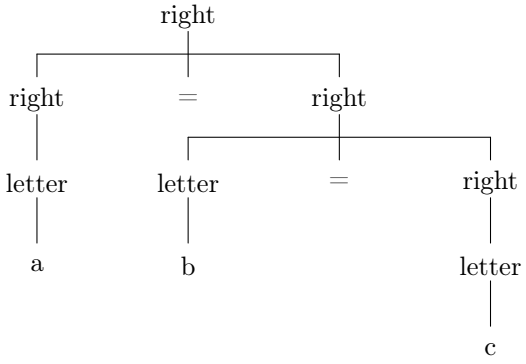




Asociatívnosť operátorov.

$$\begin{array}{rcl} right & \rightarrow & letter = right \mid letter \\ letter & \rightarrow & a \mid b \mid c \mid \ldots \mid z \end{array}$$





Priorita operátorov.

1. ľavo asociatívne + -
2. ľavo asociatívne * /

$$\begin{array}{rcl} expr & \rightarrow & expr + term \mid expr - term \mid term \\ term & \rightarrow & term * factor \mid term/factor \mid factor \\ factor & \rightarrow & digit \mid (expr) \end{array}$$

Príklad:

$$\begin{array}{rcl} stm & \rightarrow & id := expr \mid \\ & & \textbf{if } expr \textbf{ then } stm \mid \\ & & \textbf{if } expr \textbf{ then } stm \textbf{ else } stm \mid \\ & & \textbf{while } expr \textbf{ do } stm \mid \\ & & \textbf{begin } stm\_list \textbf{ end} \mid \\ stm\_list & \rightarrow & stm\_list;\ stm \mid \epsilon \end{array}$$

Postfixový zápis operátorov.

1. Ak E je premenná alebo konštanta potom $E' = E$.
2. Ak $E = E_1 \mathbf{op} E_2$ potom $E' = E_1' E_2' \mathbf{op}$.
3. Ak $E = (E_1)$ potom $E' = E$.

Príklad:

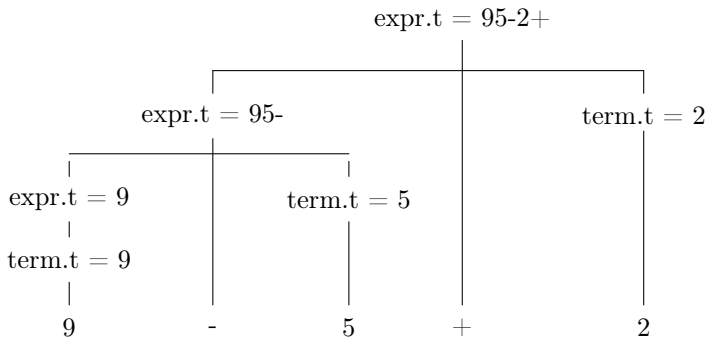
$$\begin{array}{lcl} (9-5)+2 & \Rightarrow & 95-2+ \\ 9-(5+2) & \Rightarrow & 952+- \end{array}$$

2.1 Syntaxou riadený preklad.

Pravidlo	Sémantické pravidlá
expr → expr + term	expr.t := expr1.t ‖ term.t ‖ '+'
expr → expr - term	expr.t := expr1.t ‖ term.t ‖ '-'
expr → term	expr.t := term.t
term → 0	term.t := 0
term → 1	term.t := 1
⋮	
term → 9	term.t := 9

Všetky premenné majú rovnaký atribút t.

Syntaktický strom ohodnotený atribútami:



Definícia 2.2 *Syntetizované atribúty sú také atribúty, že hodnota atribútu v každom uzle sa dá určiť z hodnôt atribútov jeho následníkov t.j. môžu byť vyhodnotené na jeden prechod zdola nahor.*

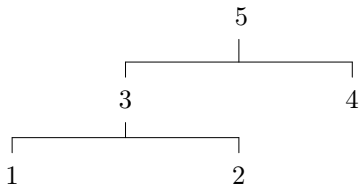
Príklad: Sledovanie pozícii robota.

instr → *east* | *north* | *west* | *south*
seg → *seginstr* | **begin**

Pravidlo	sémantické pravidlá
seg → **begin**	x := 0; y := 0;
seg → seg instr	x := x + dx; y := y + dy;
instr → east	x := 1; y := 0;
instr → north	x := 0; y := 1;
instr → west	x := -1; y := 0;
instr → south	x := 0; y := -1;

Prehľadávanie do hĺbky. (depth-first search)

```
procedure visit(n : node);
begin
    for each child m of n from left to right do
        visit(m);
    evaluate semantic rules at node n
end
```



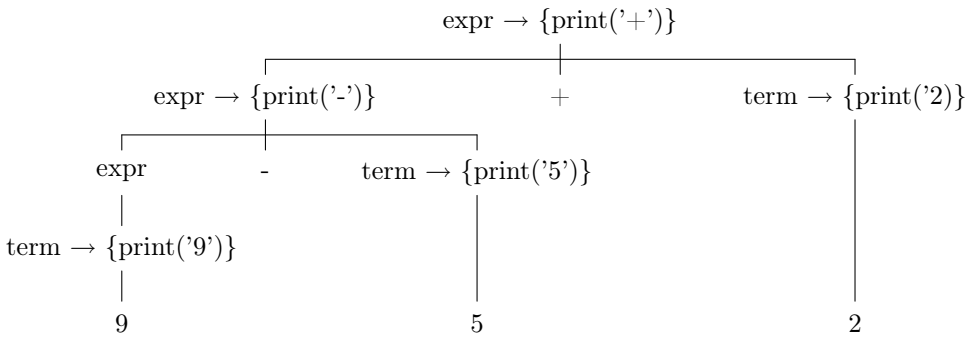
Čísla udávajú poradie, v akom sú uzly navštívené pri prehľadávaní do hĺbky.

Očakáva sa, že miesto sémantickej akcie je udané explicitne.

Nevýhoda: Výstup sa vypočíta počas analýzy a musí byť celý v pamäti. Chceli by sme ho produkovať inkrementálnde.

Pravidlá:

expr	→	*expr* + *term*	{print ('+')}
expr	→	*term*	{print ('-')}
expr	→	*term*	
term	→	0	{print ('0')}
term	→	1	{print ('1')}
⋮			
term	→	9	{print ('9')}



Synktatická analýza: CFG: $O(n^3)$.
Programovacie jazyky špeciálne prípady $O(n)$

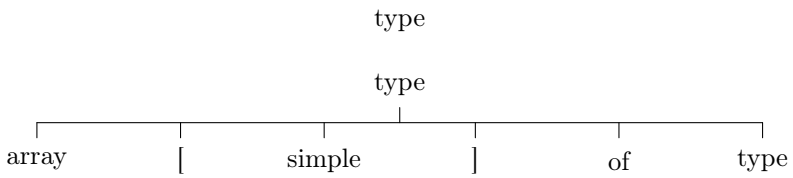
Rozdelenie metód syntaktickej analýzy:

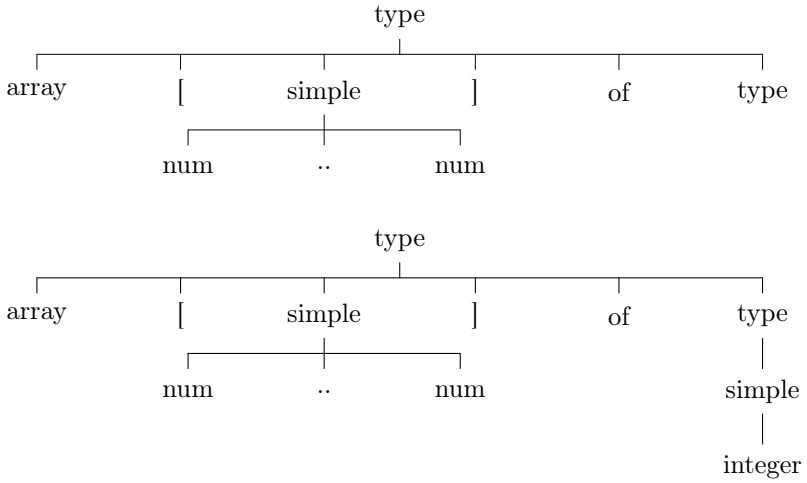
zhora-dolu (top-down)	zdola-nahor (bottom-up)
ľahšie sa ručne robí	širšia trieda gramatík
lepšie sa zaradia	a prekladových schém.
sémantické programy.	

Príklad:

type	→	*simple*	|	↑ *id*	|	**array***[simple] of type*
simple	→	*integer*	|	*char*	|	*num..num*

INPUT: array [num..num] of integer






```
procedure match(t:token);
begin
    if lookahead = t then
        lookahead:=nexttoken
    else error fi [1]
end ;

procedure type;
begin
    if lookahead is in {integer,char,num} then
        simple
    else if lookahead = '^' then
        match('↑'); match(id)
    else if lookahead = array then
        match(array); match('[');
        simple; match(']'); match('of');
        type;
    else error
    fi fi fi
end ;

procedure simple;
begin
    if lookahead = integer then
        match(integer)
    else if lookahead = char then
        match(char)
    else if lookahead=num then
        match(num); match('..'); match(num)
    else error
    fi fi fi
end
```

Definícia 2.3 $First(\alpha) = \{x : (x \in T \cup \{\varepsilon\}) \cap \alpha \xrightarrow{*} x.w\}$
Ak $A \to BC$ *a* $B \xrightarrow{*} \varepsilon$, *potom* $First(A) = First(B) \cup First(C)$.

Prázdny symbol ε treba ošetriť špeciálne. Pravidlo $A \to \varepsilon$ sa použije v rekurzívnom zostupe iba vtedy, ak sa žiadne iné pravidlo nedá použiť. Použitá technika sa nazýva prediktívna kompilácia resp. „recursive descent“.

[1] Používame nasledujúcu syntax podmieneného príkazu:
cond → **if** < *podmienka* > **then** < $S1$ > **else** < $S2$ > **fi**

Pri konštrukcii najprv ignorujeme sémantické akcie a zostrojíme syntaktický analyzátor. Potom vpíšeme sémantické akcie na miesto, kde patria.

Zostrojíme množiny First pre všetky neterminály. Ak tieto množiny nie sú disjunktné tak sa metóda rekurzívneho zostupu nedá použiť.

Odstránenie ľavej rekurzie.

Ľavá rekurzia spôsobuje zacyklenie pri rekurzívnom zostupe, preto ju nahrádzame pravou podľa nasledujúcej schémy:

$$\begin{array}{lll} A \to A\alpha & \Rightarrow & A \to \beta R \\ & & R \to \alpha R \ \mid \ \varepsilon \end{array}$$

Ak nejde o bezprostrednú ľavú rekurziu. Musíme očíslovať neterminálne symboly v cyklických pravidlách a postupnou substitúciou za ne dostaneme nakoniec bezprostrednú ľavú rekurziu.

Príklad:

$$\begin{array}{rcl} \mathrm{A} & \to & \mathrm{B}\alpha \\ \mathrm{B} & \to & \mathrm{C}\beta \\ \mathrm{C} & \to & \mathrm{A}\gamma \end{array}$$

Dosadíme:

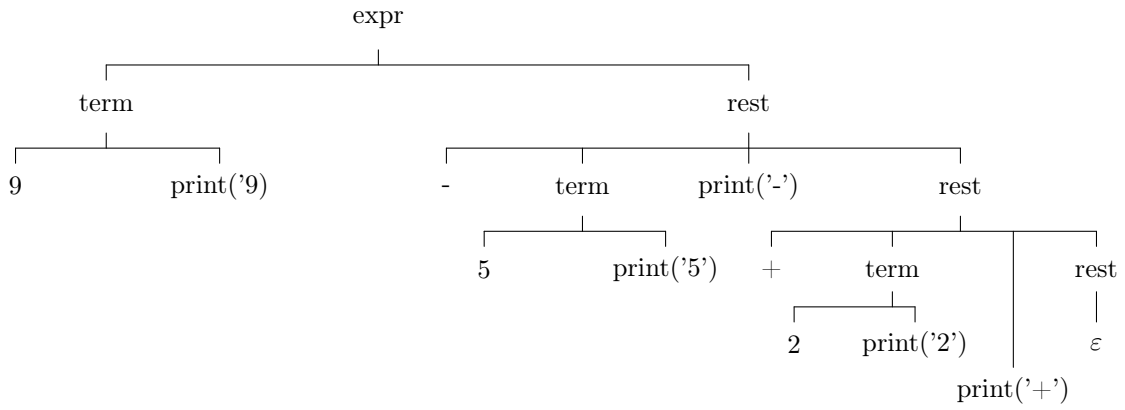
$$\begin{array}{rcl} \mathrm{C} & = & \mathrm{A}\gamma \\ \mathrm{B} & = & \mathrm{A}\gamma\beta \end{array}$$

Dostaneme pravidlo:

$$\mathrm{A} \to \mathrm{A}\gamma\beta\alpha$$

$$\begin{array}{rcl} \mathit{expr} & \to & \mathit{term}\ \mathit{rest} \\ \mathit{rest} & \to & +\ \mathit{term}\ \{\mathit{print}('+')\}\ \mathit{rest} \ \mid \\ & & -\ \mathit{term}\{\mathit{print}('-')\}\mathit{rest} \ \mid \ \varepsilon \\ \mathit{term} & \to & 0\ \{\mathit{print}('0')\} \ \mid \ \ldots \ \mid \ 9\ \{\mathit{print}('9')\} \end{array}$$

Príklad: 9-5+2



Program:

```
procedure expr;
begin
    term;
    rest
end

procedure rest;
begin
    case lookahead of
        '+': begin
                match('+'); term; print('+'); rest
             end
        '-': begin
                match('-'); term; print('-'); rest
             end
        otherwise: ;
    end
end

procedure term;
begin
    if is_digit(looahead) then
        begin
            print(lookahead);
            match(lookahead);
        end
        else ERROR(2);
end

function is_digit(x:char) : boolean;
    < is_digit=true ak x je číslica. Inak is_digit=false. Funkcia závisí na kódovaní znakov. >

procedure ERROR(n)
begin
    writeln(Error_message[n]);
    < správa o chybe >
    case n of
        1: while (lookahead ≠ '+') or (lookahead ≠ '-') do read(lookahead);
        2: while not is_digit(lookahead) do read(lookahead)
    end zotavenie z chyby;
end
```

Optimalizácia.
odstránenie "chvostovej"rekurzie.

```
procedure rest;
begin
    L: if lookahead='+' then
            begin
                match('+'); term; print('+'); goto L
            end
        else if lookahead='-' then
            begin
                match('-'); term; print('-'); goto L
            end
end
```

```
procedure expr;
begin
    term;
    loop
    if lookahead={'+' or '-'} then
        begin
            match(lookahead);
            term;
            print(lookahead);
        end
        else EXIT
    pool
end
```

2.2 Lexikálny analyzátor.

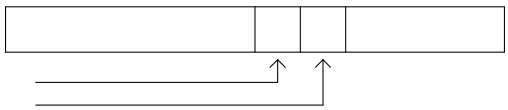
Prečo?

=		Dve funkcie
=	→	končiť lex. jednotku
oddelovače		iný význam

Príklad: id1+id2

Praktická realizácia
Vyrovnávacia pamäť

Pozerá ešte jeden
symbol v predstihu



Pravidlá tvaru $A \rightarrow aA \mid b \mid \varepsilon$ Nepotrebuje rekurziu.

Kľúčové slová.
Priamo – robí kompilátor príliš dlhý.

Príklad: Rozlíšenie medzi

else
elsa Vyžaduje dlhý predstih.

Nepriamo - cez tabuľku symbolov.

Medzery a poznámky

while lookahead = '␣' **do** read(lookahead) { alebo LF+CR či TAB (ASCII) }

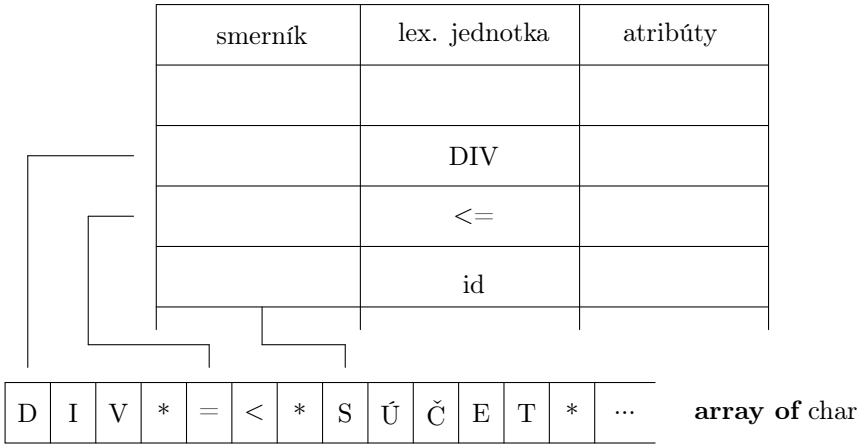
Môžu byť ťažkosti: napr. ALGOL 60

FOR I:=T1 TO T2 STEP T3

Tabuľka symbolov:

ADT: member(x):boolean
insert(x)
ask(x,attributes)

Realizácia:



Lexikálna jednotka môže byť kódovaná priamo relatívnou adresou v tabuľke. Na začiatku sa tabuľka inicializuje rezervovanými slovami.

Hašovanie, Rozšíriteľné hašovanie, sekvenčné stromy

Hašovanie:

Rozmer tabuľky: $n = \{ \begin{array}{l} 2^k \\ p - \text{prvočíslo} \end{array}$

Vstup je postupnosť znakov
Operátor #(xxxxxxxxxx) urobí z argumentu binárne číslo s rovnakým kódovaním
h(K)=#(K) mod n (sú aj iné možnosti)
h(K) relatívna adresa **K** v tabuľke symbolov
h(K) voľná O.K. uložíme
už obsadené pointer ukazuje na postupnosť identickú s **K**
O.K. vráti atribúty už obsadené inou postupnosťou

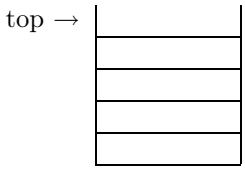
Kolízia.

Riešenie kolízie + a gcd(n,a)=1.
ďalšou hašovaciou funkciou.
Postupnosť hašovacích funkcií h1,h2,...

Cena vkladania alebo neúspešného hľadania: $\frac{1}{1-\alpha}$
Cena úspešného hľadania: $-\frac{\log(1-\alpha)}{\alpha}$,
kde α je faktor naplnenia.

2.3 Abstraktný zásobnikový počitač – generovanie kódu

zásobník (stack)



ADT top
push
pop
empty?

L-hodnoty a R-hodnoty

i:=5; ← hodnota
i:=i+1; ← referencia (adresa)

operácie a+b r value a
r value b
+

INAK push <a>
push <b>
AC:=a
AC:=a+b
pop
pop
push <AC>

operácie

push r - daj r do zásobníka
rvalue a - daj obsah adresy a do zásobníka
lvalue a - daj adresu a do zásobníka
pop - odstráň položku z vrcholu zásobníka
:= - lvalue je rvalue; pop; pop
copy - push top

Príklad:

$day := (1461 * y) div 4 + (153 * m + 2) div 5 + d$

Preklad:	1 l value day
	2 push 1461
	3 r value y
	4 *
	5 push 4
	6 div
	7 push 153
	8 r value m
	9 *
	10 push 2
	11 +
	12 push 5
	13 div
	14 +
	15 r value d
	16 +
	17 :=

Postupnosť inštrukcií ASM (abstract stack machine)
Inštrukcie sa vykonávajú v poradí, ak nie je špecifikované inak.

label l	môže byť cieľom skoku
	nemá iný účinok.
goto l	nasledujúci sa vykoná príkaz label l
gofalse l	pop top and if top= 0 then goto l
gotrue l	pop top and if topne0 then goto l
halt	zastavenie

Preklad príkazov:

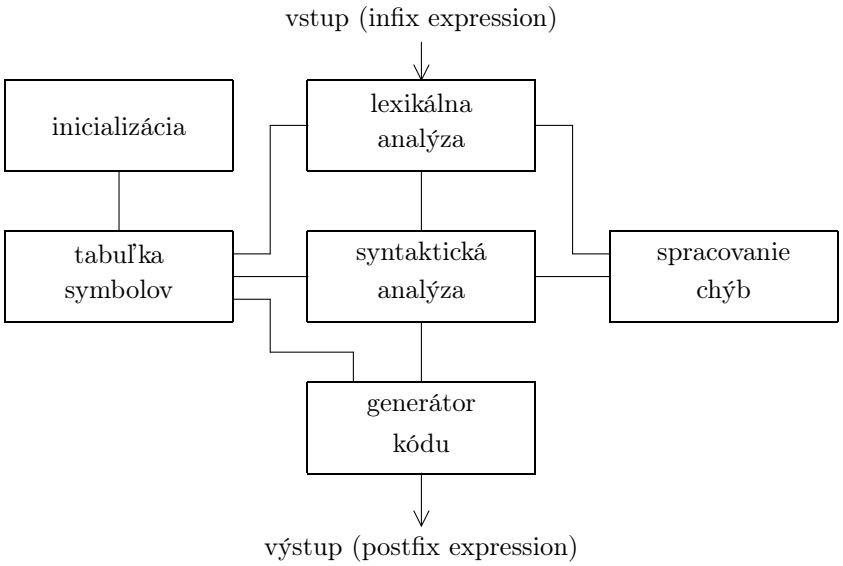
stmt → **if** expr **then** stmt1 IF stm → **if**

code for expr
go false out
code for stmt1
label out

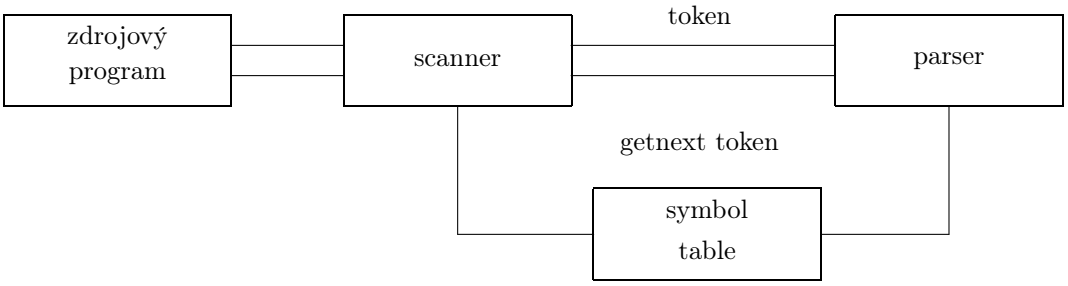
while expr **do** stmt1 **end** ;

label test
code for expr
go false out
code for stmt1
goto test
label out

Celková štruktúra kompilátora



3 Lexikálny analyzátor.



1. Prečo oddeľujeme lexikálnu a syntaktickú analýzu?
   - jednoduchší návrh (medzery,komentáre)
   - efektívnosť
   - portabilita a nezávislosť na kódovaní.

2. Čo rozpoznáva lexikálna analýza?

   -konštanty: numerické (numeral)
   reťazcové (literál,string)
   znakové (character)

   -operátory
   -oddeľovače (zátvorky, ";", ...)
   -rezervované slová
   -identifikátory.

Je to ťažké? *Obvykle nie, ale...*

Príklad:

FORTRAN: DO 5 I = 1.25
DO 5 I = 1,25

PL/1: IF THEN THEN THEN = ELSE;
ELSE ELSE = THEN;
DECLARE(A1,A2,A3,...,An)

Komunikácia medzi lex.analyzátorom a syntaktickým analyzátorom:

1-prech.	- prirodzená:	lexikálna jednotka + atribúty
	- skrátená:	lex.jednotka + smerník do tabuľky symbolov
- multiprech.	- minimálna:	smerník do tabuľky symbolov

Príklad:

E := M * C ** 2
< id, ^SYMB E >
< operátor_priradenia >
< id,^SYMB M >
< operátor_násobenia >
< id, ^SYMB C >
< operátor_umocnenia >
< konšt, integer 2 >

Lexikálne chyby a zotavenie z chýb.

ťažkosť: scanner vidí text príliš lokálne

Príklad:

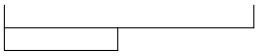
fi(a=f(x))

- identifikátor
- if?

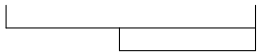
Zotavenie:
- vynechanie pochybnej lex. jednotky
- vynechanie zbytočného znaku
- vloženie chýbajúceho znaku
- transpozícia dvoch vedľajších znakov
- výmena nesprávneho znaku

Pojmy:

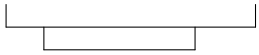
prefix



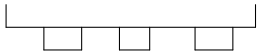
suffix



podreťazec



podpostupnosť



Príklad:

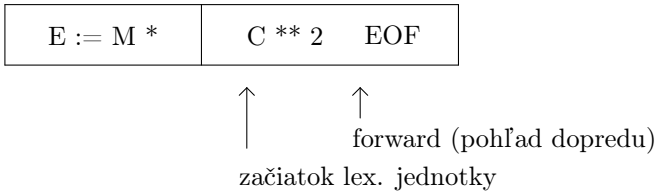
banana prefix (ban)
suffix (nana)

cele slovo < prefix / suffix < podreťazec (anan) < podpostupnosť (baaa)

Chybový stav: prezeraný reťazec nie je prefix žiadnej lexikálnej jednotky.
– teória: predpoklad minimálnej chyby
(prakticky sa nerealizuje – príliš náročné)

Ako sa implementuje scanner?	Obtiažnosť	Efektívnosť
-generátor lex. analyzátorov	1	3
-v konvenčnom program. jazyku (C, modula, pascal) I/O modul jazyka	2	2
-assembler s explicitným riadením I/O	3	1

Cyklické používanie 2 vyrovnávacích pamätí:



if *forward na konci 1. polovičky* **then**
begin
    načítaj 2. polovičku
    forward := forward + 1
end
else
if *forward na konci 2. polovičky* **then**
begin
    načítaj 1. polovičku
    forward := 0
end
else *forward := forward + 1*

Použitie zarážok:

E := M *	EOF	C ** 2 EOF	EOF

forward := forward + 1
if *forward^ = EOF* **then**
begin
    if *forward na konci 1. časti* **then**
        čítaj 2. časť
        forward := forward + 1
    end
    else
    if *forward na konci 2. časti* **then**
    begin
        čítaj 1. časť
        forward := 0
    end
    else < *koniec súboru* >
        skonči lexikálnu analýzu

end

3.1 Špecifikácia lexikálnych jednotiek

Regulárne výrazy (Rational language)

1.	ϵ	$\{ \epsilon \}$	
2.	a $\epsilon\Sigma$	{ a }	
3.	r|s	L(r) $\cup$ L(s)	
	rs	L(r)L(s)	
	r^*	$(L(r))^*$	$r^* = \epsilon \cup$ r $\cup$ rr $\cup$ rrr ...
4.	(r)	L(r)	uzátvorkovanie

Priorita: * . |

Skratky:

$r^+ = rr^*$	($r^* = \epsilon \cup r^+$)
	($r^* = r^+ \mid \epsilon$
r? = r $\mid\epsilon$	
[a-z] = a | b | ...z	

Algebraické vlastnosti:

r|s = s|r
(r|s)|t = r|(s|t)
(rs)t = r(st)
r(s|t) = rs|rt
(s|t)r = sr|tr
ϵr = rϵ = r
$r^* = (r|\epsilon)^*$
$r^** = r^*$

Príklad:

letter → *A* | *B* | *ldots* | *Z* | *a* | *b* | *ldots* | *z*
digit → 0 | 1 | ... | 9
id → *letter*(*letter* | *digit*)
digits → $digit^+$
optional_Fraction → *.digits* | ϵ
optional_Exponent → (*E*(+ | − | ϵ))*digits* | ϵ
num → *digitsoptional_Fractionoptional_Exponent*

Existujú jazyky, ktoré nie sú regulárne.
$\{a^n b^n ; n \leq 0\}$ S → aSb | ϵ

Existujú jazyky, ktoré nie sú bezkontextové
{wcw; w ϵ a,b* }

Príklad:

$$
\begin{array}{lcl}
if & \rightarrow & if \\
then & \rightarrow & then \\
else & \rightarrow & else \\
relop & \rightarrow & < \mid <= \mid = \mid <> \mid > \mid >= \\
id & \rightarrow & letter(letter \mid digit)^* \\
num & \rightarrow & digit^+(.digit^+)?(E(+ \mid - \mid \epsilon)?digit^+)? \\
delim & \rightarrow & blank \mid tab \mid newline \\
ws & \rightarrow & delim^+
\end{array}
$$

3.2 Rozpoznávanie lexikálnych jednotiek

Prechodové diagramy, konečné automaty <stav,symbol> → nový stav
∗ . . . označuje, že posledný vstupný symbol treba vrátiť.

Implementácia prechodových diagramov:
Začíname: start:=0; state:=0;
Ak nájdeme hranu, zodpovedajúcu prezeranému symbolu,prechádzame po nej.
Ak takáto hodnota neexistuje, prejdeme k nasledujúcemu diagramu (start:=9) atď.
Ak žiadna hodnota neuspela => chyba.

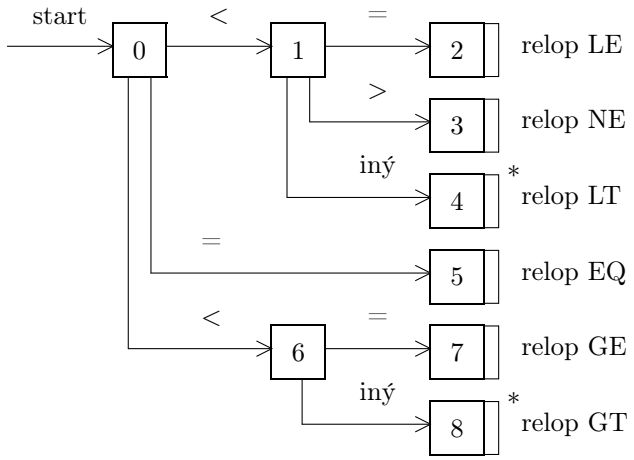
Prechodové diagramy a konečné automaty.
DFA,NFA
Prechodové tabuľky.

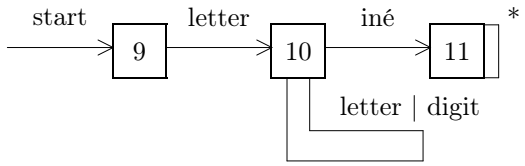
Konečný automat . . . Finite Automaton

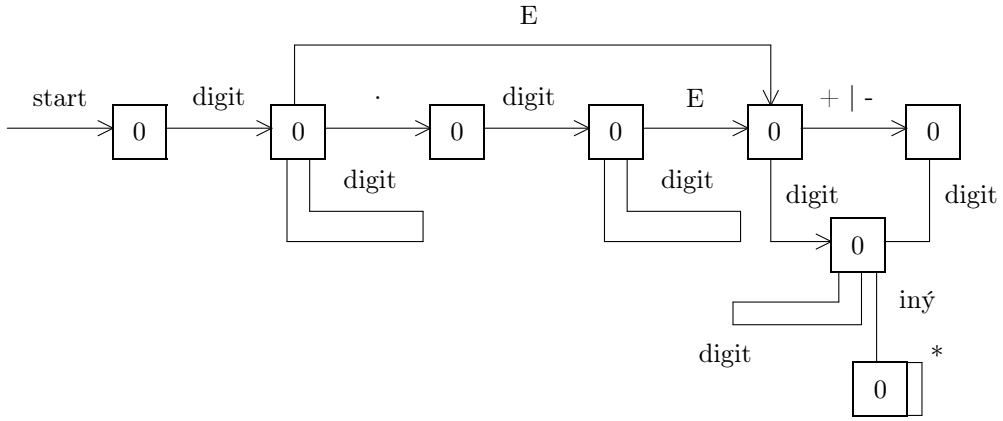
1. S - množina stavov
2. Σ - abeceda vstupných symbolov
3. Prechodová funkcia t: S x Σ → 2S
4. s_0 ϵ S počiatočný stav
5. F ⊂ S (konečná) akceptujúce stavy

Ak prechodová funkcia obsahuje na pravej strane najviac 1-prvkové množiny, tak automat je deterministický inak je nedeterministický.

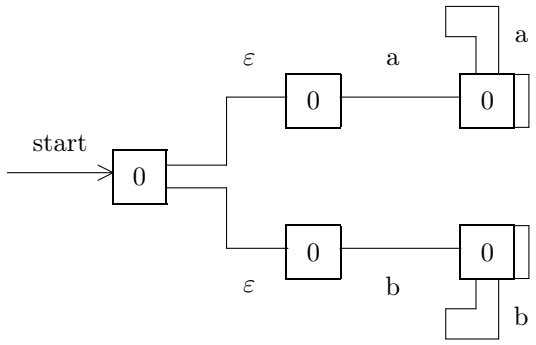
Príklad:







Príklad:



LA = { aa* | bb* }

Veta 3.1 *Každý regulárny jazyk sa dá rozpoznať nejakým DFA. Každý NFA rozpoznáva nejaký regulárny jazyk.*

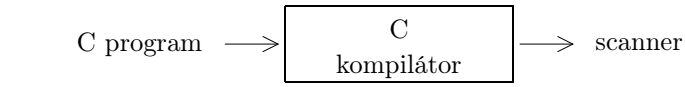
3.3 Automatický návrh lexikálneho analyzátora

Lex c Yacc c UNIX/c .

Lex program

deklarácie %% pravidlá %% procedúry.

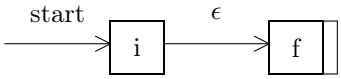
{ p_i { $action_i$ } }*
pi → regulárny výraz
actioni *rightarrow* činnosť pri jeho rozpoznaní



Stratégia realizácie

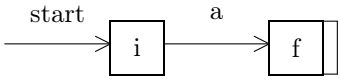
Regulárny výraz → NFA → DFA → minimal. DFA.
R → NFA: Thompsonova konštrukcia

1. ϵ



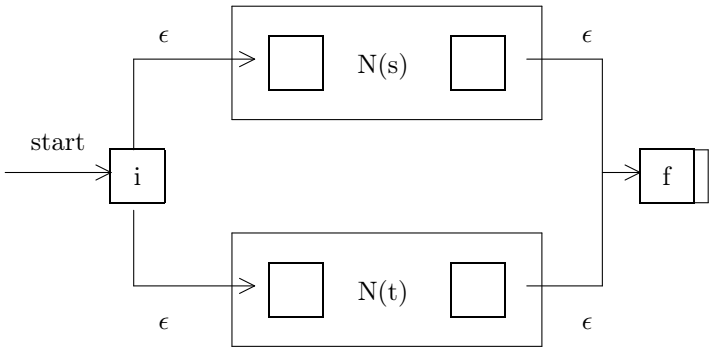
{ ϵ }

2. a$\epsilon\Sigma$



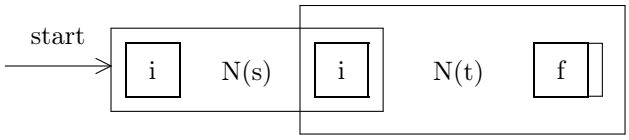
{ a }

3. N(s) – rozpoznáva s
   N(t) – rozpoznáva t



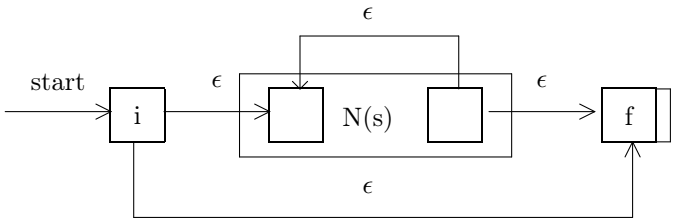
S | t

(a)



st

(b)



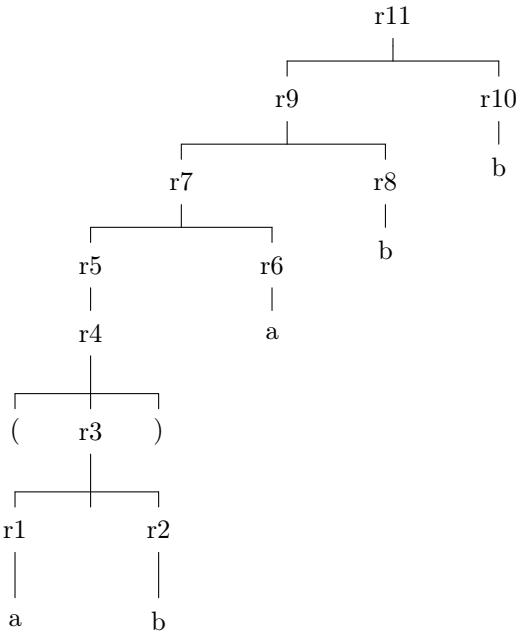
s^*

(c)

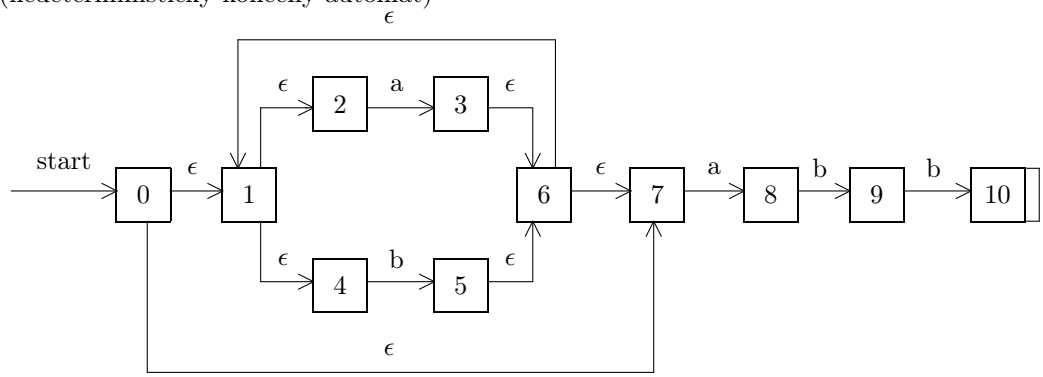
(d) NFA pre (s) je priamo N(s).

Príklad:

(a | b)*abb



NFA (nedeterministický konečný automat)



NFA → DFA subset construction

ϵ-closure(s) = { q: NFA s $\xrightarrow{\epsilon}$ q }

ϵ-closure(T) = $\bigcup_{s \in T}$ ϵ-closure(s)

move (s,a) = { q: s $\xrightarrow{a}$ q } move (T,a) = $\bigcup_{s \in T}$ move (s,a).

stavy DFA D-states sú množiny stavov NFA. VÝPOČET ϵ-closure (T);

```
procedure ε-closure(T);
begin
    push all states in T onto stack;
    ε-closure(T) := T;
    while stack is not empty do
    begin
        pop t, the top of stack;
        for each u such that there is an edge t --ε--> u do
            if u is not in ε-closure(T) then
                ε-closure(T) := ε-closure ∪ {u};
                push u onto stack fi
    end
end
```

KONŠTRUKCIA DFA:

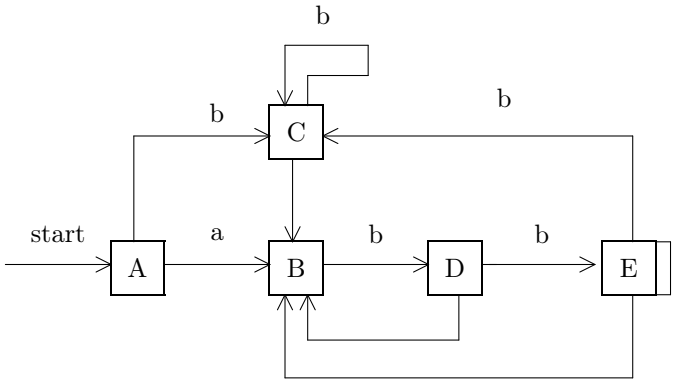
```
procedure DFA-construction;
begin
    Dstates = { ε-closure(s0); unmarked };
    while there is an unmarked state T in Dstates do
    begin
        mark T;
        for each input symbol a do
        begin
            u := ε-closure(move(T,a));
            if u is not in D states then
                Dstates:=Dstates u { u; unmarked };
            fi ;
            Dtran[T,a]:=u
        end
    end
end ;
```

A = ϵ-closure(0) = {0,1,2,4,7}
B = ϵ-closure(move(A,a)) = {1,2,3,4,6,7,8}
C = ϵ-closure(move(A,b)) = {1,2,4,5,6,7}
D = ϵ-closure(move(B,b)) = {1,2,4,5,6,7,9}
E = ... {1,2,4,5,6,7,10}
nie sú iné nové stavy

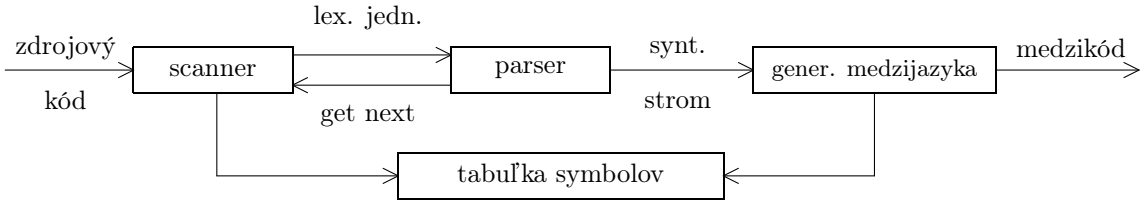
Dtran (prechodová tabuľka deterministického automatu)

	a	b
A	B	C
B	B	D
C	B	C
D	B	E
E	B	C

DFA (deterministický konečný automat



4 Syntaktická analýza



Typy synt. analýzy
podľa metódy:
- zhora-dole | **LL**(k)
- zdola-hore | **LR**(k)

podľa triedy gramatík:
- všeobecné CFG | Cocke-Younger-Kasami; Earley
- podmnožiny CFG | **LL**(k) **LR**(k) operátorovo precedenčné gr.

Výhody a nevýhody:
LL(k) vždy vieme pravidlo i symbol
LR(k) vieme symbol pravidla až po rozpoznaní pravej strany.

Chyby a zotavenie z chýb:
- lexikálne chyby (misspeling)
- syntaktické chyby:
  interpunkcia (;)
  zátvorky (**begin** , **end**) zle spárované
  operátory := versus =
  zložené príkazy neuzátvorkované

(Značná časť chýb.)
- sémantické chyby (nesprávne typy operandov)
- logické chyby (prg. nekončí, nerobí čo má a pod.)

Požiadavky:
- Oznámiť chyby: presne,jasne a zrozumiteľne.
- Rýchle sa zotaviť z chýb, aby mohli byť diagnostikované ďalšie chyby.
- Spracovanie chýb nesmie výrazne spomaliť beh správnych programov.

Range and type checking.

Stratégie:
- panický stav (panic mode)
- oprava konštrukcií jazyka (phrase level)
- chybové pravidlá (error productions)
- globálna oprava (global correction)

Bezkontextové jazyky:

– správne spárované zátvorky
$S \rightarrow (S)S \quad | \quad \epsilon$
CFG G=< N,T,S,R >
konvencie: N < >, modro, italický
S symbol na ľavej strane 1.pravidla
T tučne, červene, ...

Odvodenie
$\rightarrow$ jeden krok
$\overset{*}{\rightarrow}$ reflexívny a tranzitívny uzáver

lefmost derivation (najľavejšie odvodenie)
syntaktický strom
Nejednoznačnosť.

Príklad:

E $\rightarrow$ E+E | E-E | E*E | (E) | -E | id

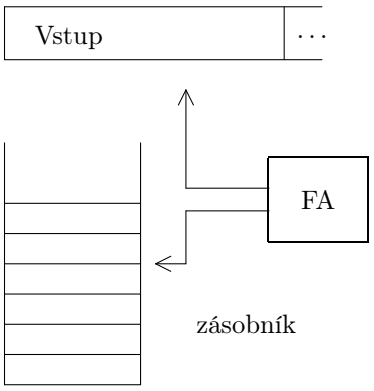
1	E	$\rightarrow$	E+E		2	E	$\rightarrow$	E*E
		$\rightarrow$	id+E				$\rightarrow$	E+E*E
		$\rightarrow$	id+E*E				$\rightarrow$	id+E*E
		$\rightarrow$	id+id*E				$\rightarrow$	id+id*E
		$\rightarrow$	id+id*id				$\rightarrow$	id+id*id

Zjednoznačňujúce pravidlá alebo ekvivalentná jednoznačná gramatika.

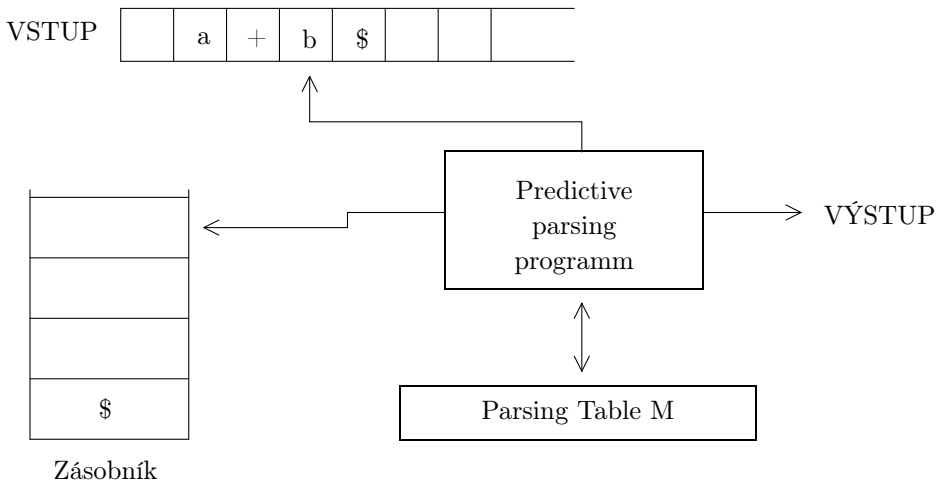
Dôkaz, že CFG generuje daný jazyk: S $\rightarrow$ (S)S|ϵ generuje práve všetky dobre spárované zátvorky.

1. Generuje len dobre spárované zátvorky.
   indukcia vzhľadom na dĺžku odvodenia
   (i) S $\rightarrow \epsilon$ triviálne
   (ii) S $\rightarrow$ (S)S $\overset{*}{\rightarrow}$ (x)S $\overset{*}{\rightarrow}$ (x)y
   predpokladáme, že pre menej než n krokov funguje ukážeme, že to funguje aj pre n krokov.
2. Všetky dobre spárované zátvorky sa dajú generovať. indukcia vzhľadom na dĺžku generovaného reťazca.
   $|w| = 2n$ w = (x) y
   S $\rightarrow$ (S)S $\overset{*}{\rightarrow}$ (x)S $\overset{*}{\rightarrow}$ (x)y
   Pretože $|x| < 2n$ a $|y| <= 2n$, výrazy x, y sa dajú generovať podľa indukčného predpokladu.

PDA:



Realizácia:



First(x)

1. **if** $x \in$ T **then** $First(x) = \{x\}$;
2. **if** $x \in N$ **then** $First(x) = \emptyset$;
   (i) and $x \to \epsilon \in R$ then $First(x) := First(x) \cup \{\epsilon\}$
   (ii) if $x \to Y_1Y_2.....Y_n \in R$ **then**
   begin $First(x) := First(x) \cup First(Y_1)$
   while $Y_i \to \epsilon$ **do**
   $First(x) := First(x) \cup First(Y_{i+1})$
   end ;

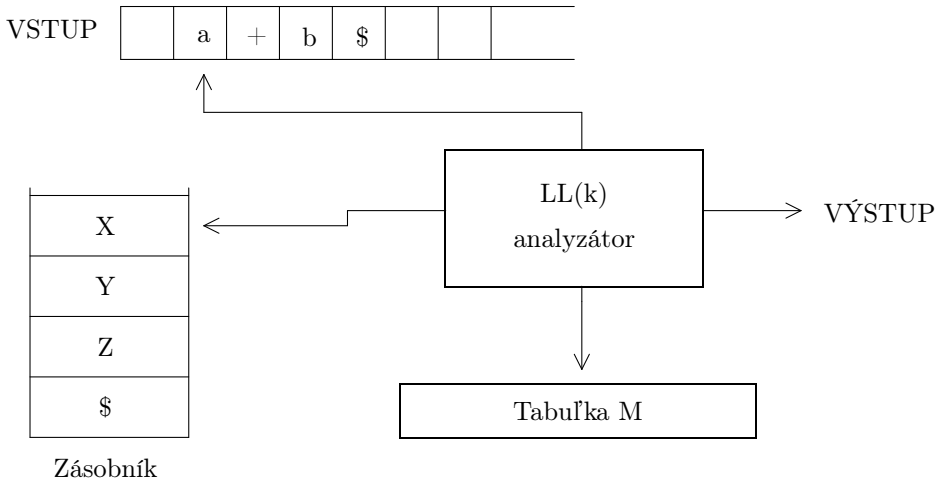
Zovšeobecnenie pre reťazce.

$First(X_1.....X_n)$

Follow(x)

1. $Folow(x) = \emptyset$
2. $Follow(S) = \{\$\}$
3. **if** $A \to \alpha B\beta$ **then**
   $Follow(B) := Follow(B) \cup (First(\omega) - \{\epsilon\})$;
4. $A \to \alpha B$ or $A \to \alpha B\omega$ and $\epsilon \in First(\omega)$ **then**
   $Follow(B) := Follow(B) \cup Follow(A)$;

4.1 LL(k) metóda



VSTUP: slovo w $\in \sigma*$ w\$
VYSTUP: najľavejšie odvodenie ak w $\in$ L(G) inak správa o chybách.

INITIALIZE:
STACK := \$S
ˆIN := 1. symbol vstupu

repeat
x := top; a := INPUT(ˆIN);
if $x \in T \cup \{\$\}$ **then**
if $x = a$ **then** *pop; next(ˆIN)* **else** *error* **fi**
else $< x \in N >$
if $M[x,a] = X \rightarrow Y_1\ Y_2 \ldots Y_k$ **then**
begin
pop;
for $i := k$ **to** *1* **do** *push*(Y_i);
output($"X \rightarrow Y_1\ Y_2 \ldots Y_k"$)
end
else *error*
until *X=\$ < stack je prázdny >;*

G: E → TE'
E' → +TE' | ϵ
T → FT'
T' → *FT' | ϵ
F → (E) | id

first(E) = first(T) = first(F) ={(,id}
first(E') = {+,ϵ}
first(T') = {*,ϵ}

follow(E) = follow(E')={),\$}
follow(T) = follow(T')={+,),\$}
follow(F) = {+,*,),\$}

Tabuľka M:

N	id	+	*	(	)	$
E	E → TE'			E → TE'		
E'		E' → +TE'		E' → ϵ	E' → ϵ	
T	T → FT'		T → FT'			
T'		T' → ϵ	T' → FT'		T' → ϵ	T' → ϵ
F	F → id			F → (E)		

Komprimovaná tabuľka:

N	id	+	*	(	)	$
E	TE'			TE'		
E'		TE'		ϵ	ϵ	
T	FT'		FT'			
T'		ϵ	FT'		ϵ	ϵ
F	id			(E)		

Konštrukcia tabuľky M:
Vstup: gramatika G
Výstup: tabuľka

Príklad:

for *all* $a \in T$ **do** *inicializácia*
  for *all* $x \in N$ **do**
    $M[a,x] := 0;$
for *all* $\{ A \to \alpha \} \in R$ **do**
  for *all* $a \in First(\alpha)$ **do**
  begin
    $M[A,a] := M[A,a] \cup \{ A \to \alpha \};$
    if $\epsilon \in First(\alpha)$ **then**
    begin
      for *all* $b \in Follow(A)$ **do**
        $M[A,b] := M[A,b] \cup \{ A \to \alpha \};$
      end
  end

Príklad:

S → iEtSS' | a
S' → eS | ϵ E → b

	a	b	e	i	$
S	a			**i**E**t**SS'	
S'			ϵ		ϵ
			eS		
b		b			

gramatika je nejednoznačná – preferujeme **e**S

Ak gramatika má v každom políčku tabuľky najviac 1 položku, nazýva sa LL(1)
L - left-to-right
L - left most derivation
1 - 1 symbol look ahead

LL(k) ... k-tice neterminálov záhlavia tabuľky

Vlastnosti LL(1) gramatík:
(i) nie sú nejednoznačné (ambiguous)
(ii) neobsahujú ľavorekurzívne pravidlá

Veta 4.1 *Gramatika G je LL(1) práve vtedy, ak pre ľubovolné dve pravidlá* $A \to \alpha$ *a* $A \to \beta$ *platí:*

1. *First(*α*)* $\cap$ *First(*β*) = 0*
2. *Ak* $\overset{*}{\to} \epsilon$*, potom* β *sa nedá odvodiť žiaden reťazec začínajúci terminálom vyskytujúcim sa vo FOLLOW(A)*

Eliminácia ľavej rekurzie.

E	→	E+T	| T	E	→	TE'		
T	→	T*F	| F	E'	→	+TE'	|	ϵ
F	→	(E)	| id	T	→	FT'		
				T'	→	*FT'	|	ϵ
				F	→	(E)	| id	

Technika:
1. Eliminácia bezprostrednej ľavej rekurzie (často stačí)

A → Aα_1 | Aα_2 | ... | Aα_m | β_1 | β_2 | ... | β_n
A → β_1A' | β_2A' | ... | β_nA'
A'→ α_1A' | α_2A' | ... | α_mA' | ϵ

Eliminácia ľavej rekurzie:

Vstup: gramatika G bez cyklov (A $\overset{*}{\to}$ A) a ϵ- pravidiel
Výstup: gramatika bez ľavej rekurzie.
Metóda:

1. Definuj usporiadanie A ,A ,...A medzi neterminálmi.
2. **for** *i:=1* **to** *n* **do**
   for *j:=1* **to** *i-1* **do**
   begin
   nahraď pravidlo $A_i \to A\gamma$ *pravidlami*
   $A_i \to \delta_1\gamma$ | $\delta_2\gamma$ | ... | $\delta_n\gamma$, *kde*
   $A_j \to \delta_1$ | δ_2 | ... δ_n *sú všetky pravidlá s pravou stranou* A_i*;*
   odstráň bezprostrednú rekurziu
   end ;

Ľavá faktorizácia:
schéma:
A → $\alpha\beta_1$ | $\alpha\beta_2$ (α-dlhšie lookahead)
A → αA'
A' → β_1 | β_2

Algoritmus:

Vstup: Gramatika G
Výstup: Ekvivalentná ľavo faktorizovaná gramatika
Metóda:

Pre každý neterminál nájdi najdlhší prefix (<> ϵ) pre aspoň dve alternatívy.
Všetky pravidlá tvaru A $\rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n \mid$ y, y reprezentuje alternatívy nezačínajúce α,
nahraď pravidlami A $\rightarrow \alpha$A' | y
A' $\rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$
Aplikuj túto transformáciu pokiaľ žiadne dve alternatívy pre ten istý neterminál nemajú spoločný prefix.

Príklad:

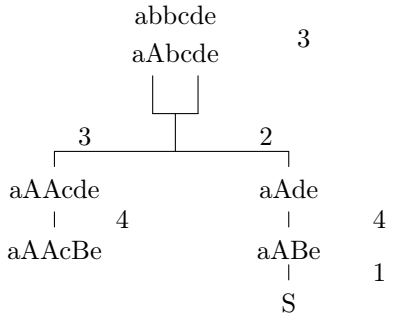
S	→	**i**E**t**S	|	**i**E**t**S**e**S	|	a	S	→	**i**E**t**SS'	| a
E	→	b					S'	→	**e**S	| ϵ
							E	→	b	

4.2 Metódy zdola - nahor

Príklad:

1 S → aABe
2 A → Abc
3 → b
4 B → d

Vstup:

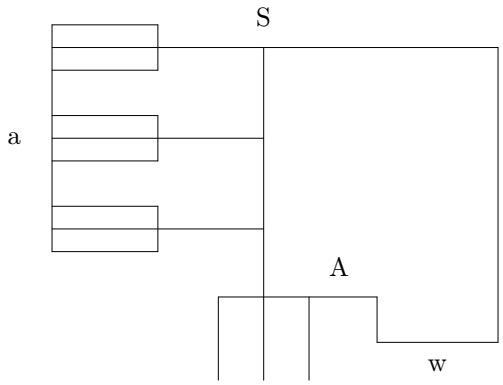


Najpravejšie odvodenie (Rightmost derivation)

S $\xrightarrow{1}$ aABe $\xrightarrow{4}$ aAde $\xrightarrow{2}$ aAbcde $\xrightarrow{3}$ abbcde

'a handle' $\Rightarrow$ pravá strana pravidla, vyskytujúca sa v nejakom najpravejšom odvodení.

"handle pruning- redukcia rukovätí



Príklad:

(1) E → E + E
(2) E → E * E
(3) E → (E)
(4) E → id

Reťazec: id1 + id2 * id3 Odvodením (rm)
124444
24144

Pravá vetná forma	handle	Redukčné pravidlo
id1 + id2 * id3	id1	E → id
E + id2 * id3	id2	E → id
E + E * id3	id3	E → id
E + E * E	E * E	E → E * E
E + E	E + E	E → E + E
E		

Posunovo - redukčná schéma:

Posunovo - redukčný stroj:

Shift - Reduce Parsing

	Stack	Vstup
Začiatok:	$	w$
činnosť (action)		
Koniec:	$ S	$

Actions:
(1) shift: push input symbol onto stack
advance input
(2) reduce: pop the handle
push lefthand side of the rule
(3) accept
(4) error

Príklad:

Stack	Vstup	Činnosť
$	id1 + id2 * id3 $	shift
$ id1	+ id2 * id3 $	reduce E → id
$ E	+ id2 * id3 $	shift
$ E+	id2 * id3 $	shift
$ E+id2	* id3 $	reduce E → id
$ E+E	*id3 $	shift
$ E+E*	id3 $	shift
$ E+E*id3	$	reduce E → id
$ E+E*E	$	reduce E → E*E
$ E+E	$	reduce E → E+E
$ E	$	accept

Handle sa vyskytuje vždy na vrchole zásobníka, nikdy vo vnútri.

Konflikty počas analýzy:
shift / reduce : Nevie rozhodnúť, či r alebo s
reduce/ reduce : Nevie rozhodnúť, ktoré pravidlo použiť.

Vie celý zásobník a nasledujúci vstupný symbol:
LR(1) LR(k)

Príklad:

S → **iEtS** | **iEtSeS** | a

Stack	Vstup
$.... **iEtS**	**e**....$

Na základe stacku nevieme rozlíšiť, či shift / reduce.
Ak dáme prednosť shift, parser sa chová prirodzene.

4.3 Operátorovo precedenčné gramatiky

Okrem iného:
1. neobsahujú ϵ- ové pravidlá
2. žiadna pravá strana neobsahuje dva po sebe idúce neterminály.

Príklad:

E → E + E | E - E | E*E | E/E | E↑E | (E) | -E | id

Precedencie medzi terminálnymi symbolmi (operátormi).
$< \cdot$ ľavé ohraničenie handle
$\doteq$ vo vnútri toho istého handle
$\cdot >$ pravé ohraničenie handle

	+	-	*	/	↑	id	(	0	$
+	$\cdot >$	$\cdot >$	$< \cdot$	$< \cdot$	$< \cdot$	$< \cdot$	$< \cdot$	$\cdot >$	$\cdot >$
-	$\cdot >$	$\cdot >$	$< \cdot$	$< \cdot$	$< \cdot$	$< \cdot$	$< \cdot$	$\cdot >$	$\cdot >$
	$\cdot >$	$\cdot >$	$\cdot >$	$\cdot >$	$< \cdot$	$< \cdot$	$< \cdot$	$\cdot >$	$\cdot >$
/	$\cdot >$	$\cdot >$	$\cdot >$	$\cdot >$	$< \cdot$	$< \cdot$	$< \cdot$	$\cdot >$	$\cdot >$
↑	$\cdot >$	$\cdot >$	$\cdot >$	$\cdot >$	$< \cdot$	$< \cdot$	$< \cdot$	$\cdot >$	$\cdot >$
id	$\cdot >$	$\cdot >$	$\cdot >$	$\cdot >$	$\cdot >$			$\cdot >$	$\cdot >$
(	$< \cdot$	$< \cdot$	$< \cdot$	$< \cdot$	$< \cdot$	$< \cdot$	$< \cdot$	$\doteq$	
)	$\cdot >$	$\cdot >$	$\cdot >$	$\cdot >$	$\cdot >$			$\cdot >$	$\cdot >$
$	$< \cdot$	$< \cdot$	$< \cdot$	$< \cdot$	$< \cdot$	$< \cdot$	$< \cdot$		

Príklad:

$$\$ < \cdot \text{ id } \cdot > + < \cdot \text{ id } \cdot > * < \cdot \text{ id } \cdot > \$$$

Scan:

1. LR $\rightarrow \cdot >$
2. RL $\rightarrow < \cdot$
3. $< \cdot$ HANDLE $\cdot >$ zahrňuje aj priľahlé neterminály
   $\$ < \cdot + < \cdot * \cdot > \$$

Algoritmus precedenčnej analýzy

Input : Precedenčná tabuľka + w$
Output: syntaktický strom (parse) alebo error
Metóda:

set ip na začiatok w$;
loop
  if *top=$ and ip↑ = $* **then** *EXIT*
  else
  begin
    a := top; b := ip^;
    if *a* $< \cdot b$ *or a* $\doteq$ *b* **then**
      push; advance(ip)
    else
      if *a* $\cdot >$*b* **then** <*reduce*>
      repeat *pop*
      until *top* $< \cdot$*symbol last popped*
      else *error*
      fi
    fi
pool

Konštrukcia precedenčnej tabuľky na základe priority a asociativity operátorov.

1. O_1 má väčšiu prioritu ako O_2
   $O_1 \cdot > O_2$
   $O_2 < \cdot O_1$
   Príklad: $+ < \cdot *, * \cdot > +$
2. O_1 a O_2 majú rovnakú prioritu (O_1 je O_2)
   $O_1 \cdot > O_2$, $O_2 \cdot > O_1$ ak operátory sú ľavo asociatívne
   $O_1 < \cdot O_2$, $O_2 < \cdot O_1$ ak operátory sú pravo asociatívne

Príklad: $+ \cdot > +$, $+ \cdot > -$, $- \cdot > +$, $\uparrow < \cdot \uparrow$

3. $O < \cdot$ id, id $\cdot > O$ $\quad O \cdot >$, $< \cdot O$
   $O < \cdot$ (, ($< \cdot O$ $\quad$ pre každé O
   $O \cdot >$),) $\cdot > O$
   ($\doteq$) $\quad$ \$ $< \cdot$ ($\quad$ \$ $< \cdot$ id
   ($< \cdot$ ($\quad$ id $\cdot >$ \$ $\quad$) $\cdot >$ \$
   ($< \cdot$ id $\quad$ id $\cdot >$) $\quad$) $\cdot >$)

Kompresia precedenčnej tabuľky – precedenčné funkcie

dvojica funkcií f,g.
f(a) < f(b) ak a $< \cdot$b
f(a) = f(b) ak a $\doteq$ b
f(a) > f(b) ak a $\cdot >$b

Príklad:

	+	-	*	/	↑	id	(	0	$
f	2	2	4	4	4	0	6	6	0
g	1	1	3	3	5	5	0	5	0

Metóda:

1. Vytvor symboly fa a ga pre všetky a $\in$ T $\cup$ { \$ }
2. Zlučuj skupiny: Na počiatku je každý symbol jedna skupina.
   Ak a = b zlúč skupinu fa a gb.
3. Zo skupín urob uzly orientovaného grafu. Ak a $< \cdot$b $\Rightarrow$ hrana (gb) $\rightarrow$ (fa)
   Ak a $\cdot >$b $\Rightarrow$ hrana (fa) $\rightarrow$ (gb).
4. Ak graf je cyklický precedenčné funkcie neexistujú.
   $\left.\begin{matrix} \text{fa} \\ \text{ga} \end{matrix}\right\}$ = najdlhšia cesta začínajúca v danej skupine.

Spracovanie chýb.

	id	(	)	$
id	e3	e3	$\cdot >$	$\cdot >$
(	$< \cdot$	$< \cdot$	$\doteq$	e4
)	e3	e3	$\cdot >$	$\cdot >$
$	$< \cdot$	$< \cdot$	e2	e1

e1: D: Chýba celý operand (výraz).
T: insert id into the input
M: "missing operand"

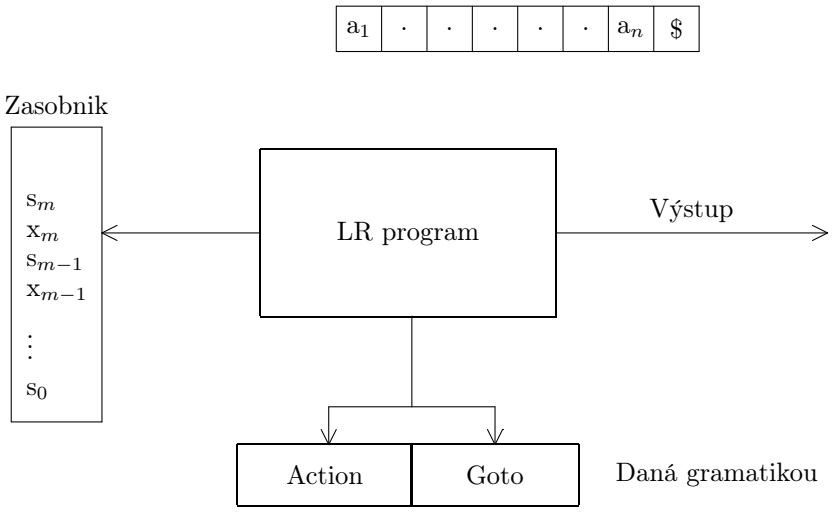
e2: D: Výraz začína pravou zátvorkou.
T: delete) from the input
M: "missing left parenthesis"

e3: D: Za výrazom nasleduje 'id' alebo '('
T: insert + into the input
M: "missing operator"

e4: D: Výraz končí ľavou zátvorkou.
T: pop (from the stack
M: "missing right parenthesis"

4.4 LR(k) – metódy synaktickej analýzy

1. LR-analyzátory rozpoznávajú všetky programovacie jazyky, pre ktoré vieme nájsť CF-gramatiku.
2. LR-metóda je najvšeobecnejšia metóda bez pokusov a omylov (backtracking).
3. Trieda gramatík analyzovateľná LR-analyzátormi je vlastnou nadmnožinou predikatívne (zhora-dolu) analyzovateľných gramatík.
4. LR-analyzátor môže zistiť chyby najskôr ako je to možné pri prezeraní zľava doprava.



actions: 1. shift s := push(in); push(s);
2. reduce := pop(right side A → β);
3. accept
4. error

Konfigurácia – okamžité opísanie
(s_0 X_1s_1 X_2s_2 … X_ms_m, a_ia_{i+1} … a_n \$)
zásobník zvyšok vstupu

Konfigurácii zodpovedá pravá vetná forma, ktorú dostaneme vynechaním stavových symbolov.

1. action [s_m, a_i] = shift s
   (s_0 X_1s_1 X_2s_2 … X_ms_m, a_ia_{i+1} … a_n \$)
   (s_0 X_1s_1 X_2s_2 … X_ms_m a_is, a_{i+1} … a_n \$)

2. action [s_m, a_i] = reduce A → β, where $|\beta|$ = r and goto [s,A]=s.
   (s_0 X_1s_1 X_2s_2 … X_ms_m, a_ia_{i+1} … a_n \$)
   (s_0 X_1s_1 X_2s_2 … $X_{m-r}s_{m-r}$As, a_ia_{i+1} … a_n \$)

3. action [s_1, \$] = accept

4. action [s_m, a_i] = error vo všetkých ostatných prípadoch.

Príklad:

(1) E → E + T
(2) E → T
(3) T → T * F
(4) T → F
(5) F → (E)
(6) F → id

LR-analýza.

```
set ip to the first input symbol;
loop
    s:= top;
    a:= input(ip);
    case action[s,a].a of
        shift:
        begin
            push a
            push action[s,a].s
            next(ip)
        end ;
        reduce:
        begin
            <action[s,a].s =reduce( A → β)>
            for i := 2*|β|-1 downto 0 do pop;
                s:=top;
                push A;
                push goto[s,A];
                output(A → β)
        end
        accept: EXIT;
        error: CALL ERROR (s,a)
    esac
pool ;
```

Príklad:

Zásobník	Vstup	Činnosť
0	id*id+id $	shift 5
0 id 5	*id+id $	reduce F → id
0 F3	*id+id $	reduce T → F
0 T2	*id+id $	shift 7
0 T2*7	id+id $	shift 5
0 T2*7 id5	+id $	reduce F → id
0 T2*7 F10	+id $	reduce T → T * F
0 T2	+id $	reduce E → T
0 E1	+id $	shift 6
0 E1+6	id $	shift 5
0 E1+6 id5	$	reduce F → id
0 E1+6 F3	$	reduce T → F
0 E1+6 T9	$	reduce E → E + T
0 E1	$	accept

Rôzne druhy LR-analýzy :

- SLR
- LR
- LALR

Sa odlišujú len v rozsahu a spôsobe konštrukcie riadiacej tabuľky.

LR(0) – item

A → XYZ	A → .XYZ
	A → X.YZ
	A → XY.Z
	A → XYZ.
A → ϵ	A → .

item = < # rule, position >

I = set of items, označíme $\bar{I}$ = closure (I)

1. $\bar{I} \supseteq I$ t.j každý item z I patrí do closure(I).
2. **if** [A → α.Bβ] ∈ I and B → δ ∈ G **then** $\bar{I} := \bar{I}$ + {[B → .δ]};
   Bod 2 sa opakuje kým sa niečo pridáva.

function *closure(I)*
begin
  repeat
    for *each item* [$A \to \alpha.B\beta$] ∈ J **do**
    for *each production* $B \to \delta \in G$ **do**
      $J := J$ {[$B \to .\delta$]};
  until *no more items can be added to J;*
end *;*

Príklad:

E' → E
E → E + T / T
T → T * F / F
F → (E) / id

I = { [E' → .E]}
J = { E' → .E, E → .E + T, E → .T, T → .T * F, T → .T, F → .(E), F → .id }

Kernel items: S' → .S, a tie, čo nemajú bodku na ľavom kraji.
Nonkernel items: Majú bodku na ľavom okraji.
Zaujímavé množiny: Uzáver kernel items.
Non kernel sa pridávajú operáciou uzáveru.

goto(I,X)

- I - set of items
- X - symbol gramatiky
- **if** [A → α.Xβ] ∈ I **then** [A → αX.β] ∈ goto(I, X).

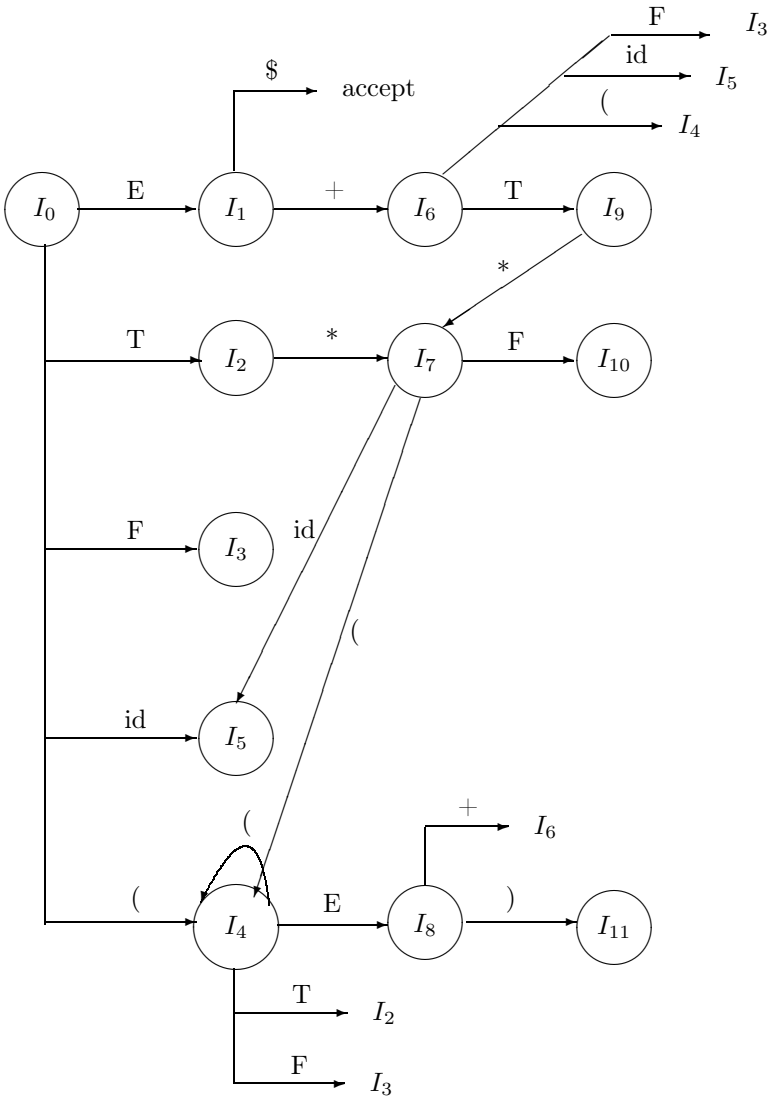
Konštrukcia stavov

1. closure (S' → .S) je stav
2. Pre každý stav I a pre každý symbol X ak goto(X, I) je neprázdne, tak closure(goto(X,I) je stav.

Príklad:

I_0:	E'	→ .E$	I_1:	E'	→ E .$
	E	→ .E + T		E	→ E .+ T
	E	→ .T			
	T	→ .T * F	I_2:	E	→ T.
	T	→ .F		T	→ T .* F
	F	→ .(E)			
	F	→ .id	I_3:	T	→ F.
I_4:	F	→ (.E)	I_5:	F	→ id.
	E	→ .E + T			
	E	→ .T	I_6:	E	→ E + .T
	T	→ .T * F		T	→ .T * F
	T	→ .F		T	→ .F
	F	→ .(E)		F	→ .(E)
	F	→ .id		F	→ .id
I_7:	T	→ T *. F	I_8:	F	→ (E.)
	F	→ .(E)		E	→ E.+ T
	F	→ id			
I_9:	E	→ E + T.	I_{10}:	T	→ T * F.
	T	→ T. * F	I_{11}:	T	→ (E).

Prechodový diagram LR(0) - automatu



Konštrukcia SLR - tabuľky

1. Skonštruuj množinu stavov
   C= {I_1 ,I_2 ,..., I_n } LR(0) analyzátora
   stav i I_i
2. Akcie
   (a) Ak A $\rightarrow \alpha.a\beta \in I_i$ a goto(I_i, a) = I_j , potom action [i,a] = shift j;
   (b) Ak [A $\rightarrow \alpha.$] $\in I_i$, potom action [i,a] = reduce A $\rightarrow \alpha$. pre všetky a $\in$ FOLLOW(A).
   (c) Ak [S' $\rightarrow$ S.] je I_i , potom action [i,$] = accept. Ak sú akcie v konflikte, gramatika nie je SLR.
3. goto tabuľka
   Pre každé i a každý neterminál A ak goto(I_i, A) = I_j, potom goto(i,A) = j.
4. Všetky ostatné položky sú chybové.
5. Stav obsahujúci S' $\rightarrow$ .S je počiatočný stav.

Konštrukcia kánonického LR(1) analyzátora.

LR(1) . item <#rule, position, terminal>

```
function closure(I);
begin
    J:= I;
    repeat
        for each item [A → α.Bβ, a] ∈ I do
        for each production B → γ do
        for each terminal b ∈ FIRST(βa) do
            J:= J ∪ {[B → .γ, b]}
    until no more item can be added;
    end ;
end ;


function goto(I, X)
begin
    K:=∅;
    for each [A → α.Xβ, a] ∈ I do
        K:= K ∪ {[A → αX.β, a]};
        K:= closure(K)
end ;


procedure states(G, C)
begin
    C:= {closure({[S' → .S, $]})}
    repeat
        for each I ∈ C do
        for each X ∈ N do
            if goto(I, X) ≠ 0 then C:=C ∪ {goto(I, X)}
    until no more sets can be added;
end ;
```

LALR(1) – metóda syntaktickej analýzy.

V predošlých vždy sme pracovali s LR(0) stavmi. V **SLR** analýze sme sa o lookahead veľmi nestarali. Iba v prípade, keď sme uvažovali o redukcii sme sa pre ňu rozhodli vždy keď lookahed patril do Follow(A), pre uvažované pravidlo A $\rightarrow \gamma$. To je príliš optimistické a nevyužíva všetku možnú informáciu. Kánonická LR(1) metóda pristupuje k problému opatrne a pre istotu všetky LR(0) stavy rozdelí na viac stavov podľa dovolených lookaheadov. To je bezpečné, ale vedie na veľké množstvo stavov. Navyše informácia o lookahed symboloch sa využíva iba pri rozhodovaní o redukcii.

Jadro stavov (kernel) je item [S' $\rightarrow$ .S$] a všetky itemy, v ktorých sa bodka nie je prvý symbol pravej strany.

Ľahká ale neefektívna (hlavne priestorovo) metóda konštrukcie LALR(1) syntaktického analyzátoru je vytvoriť LR(1) analyzátor a v tomto stotožniť stavy s rovnakým jadrom.

5 Kontrola typov

1. Operátor aplikovaný na nekompatibilné typy

   f: function (a: integer, b: real): real
   a: array [0..n] of char
   ⋮
   $\ldots a + f$

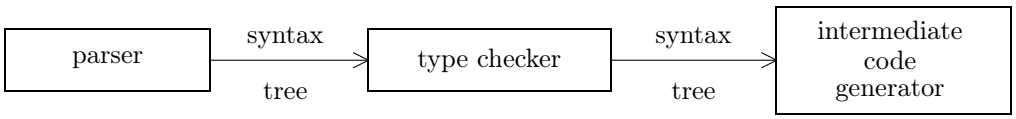
2. Kontrola toku riadenia
   EXIT,GOTO,BREAK
   Chyba ak tieto príkazy nemajú definované ciele.

3. Jednoznačnosť deklarácií a definícií.

4. Kontrola opakovaných mien.

   Príklad:

   LaTeX **begin** {enumerate}
   end {enumerate}
   procedure ADA loop (a)
   end (a);



Systém typov:

1. Základné typy: obvykle: char,integer,real,boolean. potrebujeme: void (žiadna hodnota) error (chybný typ).

2. Zložené typy (type constructors)

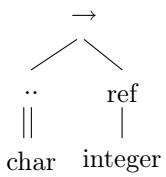
   (a) Arrays

   (b) (i) Products T1 z T2
   (ii) Records položky kartézskeho súčinu majú mená.

   (c) Smerníky (,ref)

   (d) Funkcie
   Pr. char x char → ref(integer)
   Pr. (integer → integer) → (integer → integer)

Znázornenie



Type system (systém typov)

Definícia 5.1 *Množina pravidiel priraďujúca typ rozličným častiam jazyka (programu).*

Type checker je implementácia systému typov.
Pre ten istý jazyk môžeme mať rôzne systémy typov.

Príklad:

```
Rôzne kompilátory.
C-kompilátor
lint (UNIX)
```

Statická kontrola.
Dynamická kontrola.
Recovery (zotavenie).
Každá kontrola sa dá urobiť dynamická.
Niektoré kontroly sa nedajú robiť staticky.

Príklad:

table: **array** [0..255] **of** char
i: integer (i: 0..255)
i:=i+1;
⋮ table[i]

Jednoduchý jazyk:

P → D;E
D → D;D | id:T
T → char | integer | array[num] of T | T
E → literal | num | id | E mod E | E[E] | E

Syntaxou riadený preklad pre uchovanie typov.

P → D;E	
D → D;D	
D → id:T	{addtype (id.entry,T.type)}
T → char	{T.type := char}
T → integer	{T.type := integer}
T → T_1	{T.type := pointer_to (T_1 .type)}
T → array[num] of T_1	{T.type := array (0..num.val,T_1 .type)}

Kontrola výrazov

E → literal	{E.type := char}
E → num	{E.type := integer}
E → id	{E.type := lookup (id.entry)}
E → E_1 mod E_2	{E.type := **if** E_1.type = integer} and E_2.type = integer **then** integer **else** error}
E → E_1[E_2]	{E.type:= **if** E_2.type=integer and E_1.type=array(s,t) **then** t **else** error;}
E → E_1	{E.type := **if** E_1.type=pointer_to(t) **then** t **else** error;}

Cvičenie: Zaveďte do jazyka typ boolean porovnanie a and operátor!

Kontrola typu výrazov.

$S \rightarrow id := E$	{S.type := **if** id.type = E.type **then** void **else** error}
$S \rightarrow$ if E then S_1	{S.type := **if** E.type = boolean **then** S_1.type **else** error}
$S \rightarrow$ while E do S_1	{S.type := **if** E.type = boolean **then** S_1.type **else** error}
$S \rightarrow S_1;S_2$	{S.type := **if** (S_1.type = void) and (S_2.type = void) **then** void **else** error}

Kontrola typov funkcií.

Definícia typu:
$T =\rightarrow T_1 \rightarrow T_2$ {T.type := T_1.type $\rightarrow$ T_2.type}
function f(id:T_1):T_2;

Aplikácia funkcie.
$E \rightarrow E_1(E_2)$ {E.type := **if** E_2.type = s and E_1.type = s $\rightarrow$ t **then** t else error.}

Viac argumentov (product type)
$T \rightarrow T_1$ z $T_2 \rightarrow T_3$
$T_1 \rightarrow T_2$ predošlý prípad.

Zovšeobecnenie (funkcia argumentom).
root: (real $\rightarrow$ real) z real $\rightarrow$ real
function root (function f(real): real, x: real) : real;

Ekvivalencia typov.

štrukturálna ekvivalencia:
Výrazy sú toho istého základného typu, alebo sú vytvorené tými istými konštruktormi z ekvivalentných typov.

```
function sequiv (s,t) : boolean;
begin
    if s and t are the same basic types then
        sequiv:= true
    else if (s = array(s1, s2) and t = array (t1, t2)) or
            (s = s1 .. s2 and t = t1 .. t2) or
            (s = s1 → s2 and t = t1 → t2) then
        sequiv(s1, t1);
        sequiv(s2, t2);
    else if s = pointer_to (s1) and t = pointer_to (t1) then
        sequiv (s1, t1)
    else sequiv = false
end
```

Urýchlenie: Kódovanie typov pomocou bitových postupností.

5.1 Polymorfické funkcie a operácie

Príklad:

Pascal:

```
type
    link = cell;
    cell = record
        into: integer;
        next: link
    end ;
function lenght(lptr:link): integer
var len: integer;
begin
    len:=0;
    while lptr <> nil do
    begin
        len:= len+1;
        lptr:= lptr↓next
    end ;
    lenght:= len
end ;
```

ML:
fun lenght(lptr)= If mull(lptr) Then 0 Else lenght(tl(lptr))+1;

Pascal:
Type = pointer_to type
pointer_to type = type
C:
type = pointer_to type

Dátové typy
- stack (zásobník)
- queue (fronta)
- priority queue (prioritná fronta)
- set (množina)

Typové premenné: α,β,...

Príklad:

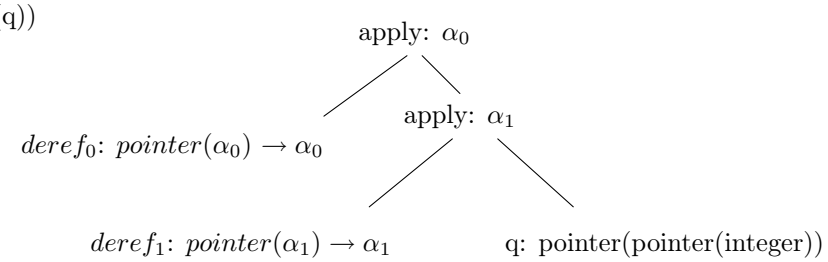
```
type link = cell;
procedure mlist(lptr:link; procedure p);
    begin
        while lptr <> nil do
            begin
                p(lptr);
                lptr:= lptr.next
            end
    end ;
type of p = link → void


function ref(xα): pointer(α);
begin
    ref:= x;
end ;


function deref(p: pointer(β)): β;
begin
    deref:= p;
end ;
```

ref: α. $\alpha \rightarrow$ pointer(α)
deref: β. pointer(β) $\rightarrow \beta$

Gramatika pre kontrolu polymorfických typov.

P → D;E
D → D;D | **id**:Q
Q → ∀*type_variable*.Q | T.
T → T'→ 'T | T..T | unary constructor(T) | basic-type | type-variable | (T)
E → E(E) | E,E | | **id**

Príklad: deref: pointer(α) → α;
q: pointer(pointer(integer));
deref(deref(q))



1. Rôzne výskyty tej istej polymorfickej funkcie vo výraze nemusia mať argumenty rovnakého typu
2. Premenné menia pojem ekvivalencie (kompatibility) typov. Namiesto zistenia ekvivalencie musíme dva typy ünifikovať".
3. Pretože tá istá premenná sa môže vyskytnúť viac krát, treba výsledky unifikácie zaznamenávať.

Term-algebry, čiastočné usporiadanie na term-algebrách, substitúcie, inštancie a unifikácia.

Term-algebra: premenné
konštanty
funkčné symboly arita
výrazy: premenná/konštanta
Ak $t_0, ..., t_{n-1}$ sú výrazy a f je funkčný symbol arity n,
potom $f(t_0, t_1, ..., t_{n-1})$ je výraz.
Neexistujú iné výrazy.

Čiastočné usporiadanie na množine termov je definované:

1. Pre každý term t a každú premennú $x \sqsupseteq t$
2. Ak $t = ft_1...t_n$ a $s = gs_1...s_n$, potom $t \sqsupseteq s$ práve vtedy, ak
   (a) $f = g$ ($\Longrightarrow$ $m = n$)
   (b) $\forall i \leq n$ $t_i \sqsupseteq s_i$

Príklad:

$pointer(\alpha)$ $pointer(integer)$
$pointer(real)$
$\alpha pointer(\beta)$
$\alpha \times \beta$ $\alpha \rightarrow \alpha(integer \rightarrow integer)$

Neporovnateľné dvojice

$integerreal$
$integer \rightarrow real\alpha \rightarrow \alpha$
$integer \rightarrow \alpha$ $\alpha \rightarrow integer$
substitúcia – $\{\alpha_i \rightarrow t_i\}_{i<n}$

6 Podpora počas behu – runtime support

6.1 Odovzdávanie parametrov

call by value
reference
name
copy - restore (copy in - copy out)

Príklad:

```
program main;
var a, b: integer;
procedure swap( var x, y: integer );
    var temp: integer;
    begin
        temp:= x;
        x:= y;
        y:= temp;
    end ;
    begin
        a:= 1;
        b:= 2;
        swap(a, b);
        writeln('a= ', a ,'b= ', b);
    end ;
    swap(i, a[ i ])
end .
```

Príklad:

```
real function f( x )
real x;
begin
    :
    :
    :
    f :=
    x := x + h
end ;

real function SUM( a, n )
value n;
real a;
integer n;
begin
    integer i;
    real a;
    s := 0;
    for i := 0 step 1 to n do s := s + a
end ;
```

```
void function swap ( x, y )
    int *x, *y;
    { int temp;
    temp := *x ;
    *x := *y ;
    *y := temp }

program main()
{ int a=1 ; b=2 ;
swap ( &a , &b )
printf( ä is now % d, b is now % d\ n", a, b); }

program copy-out;
var a: integer;
procedure unsafe ( var x: integer )
begin
    x := 2;
    a := 0;
end ;
begin
    a := 1;
    unsafe( a );
    writeln( a );
end .
```

Realizácia

by - value

1. Vypočíta sa hodnota.
2. Hodnota sa pošle volanej procedúre.
3. Volaná procedúra vztvorí novú, lokálnu premennú s touto hodnotou.

by - reference
Posiela sa smerník. Ak skutočný parameter nie je smerník, $(a + b, 2)$ sa vyhodnotí, uloží sa a pošle sa smerník na toto miesto.

by - name
Aktuálny parameter sa skompiluje ako funkcia bez parametrov, ktorá ho počíta a odovzdá sa smerník na jej volanie.

6.2 Tabuľka symbolov

Hashing. Hash functions:
$s = c_1 . . . . . c_h$ $Ord(c)$ $h = \#s mod p$

Iteratívne:
$h_0 = 0$ $\alpha = 65599$ (32 bit) $h_i = \alpha h_{i-1} + c_i$ $\alpha = 16$

```
function hashpjw (s: string): int; // Weinberger's C
var h, g, k: int; p:= prime;
begin
    h:= 0;
    k:= Lenght(s);
    for i:= 1 to k do
    begin
        h:= 16*h + s[i];
        if h > 2↑28 then
            begin
                g:= h div 2↑24; // v C a assembleri „shift"
                h:= h⊕0xf0000000⊕ g ; {( ⊕ = EOR )}
            end ;
    end ;
    hash := h mod p
end .
```

Zdá sa to zbytočne komplikované. Jednoduchá aritmetická funkcia poslúži rovnako.

```
function hashjs (s: string): int;
var h, k:int; p:= prime;
begin
    h:= 0;
    k:= Lenght(s);
    for i:= 1 to k do
        begin
            h:= 17*h + s[i];
            if h > p then h:= h mod p;
        end ;
end .
```

Riešenie kolízií :

```
1. separate chaining
```

```
2. open addressing      $\alpha < 1$
                        $h_i  := h_{i-1} + h'$
```

Pre kompilátory je vhodnejšie otvorené adresovanie. S ohľadom na jednoduchosť stačí na rehašovanie použiť konštantu h' nesúdeliteľnú s veľkosťou tabuľky m.

```
program dynamic_memory;

    type link = cell;
    cell = record key, info : integer
    next : link

    end ;
    var head : link;

    procedure insert( k, i : integer );
    var p: link;
    begin
        new(p);
        p.key := k;
        p.info:= i;
        p.next:= head;
        head := p;
    end ;
    begin
        head := nil;
        insert ( 7 , 1);
        insert ( 4 , 2);
        insert ( 76, 3);
        dispose( head^.next );
    end .
```

7 Optimalizačný kompilátor

Zdá sa,že efekt transformácií na najvyššej úrovni je najvýraznejší

Príklad: pre $N = 1024$ Insert sort $2.02N^2 = 2.02M$
Quick sort $12NlgN = 120K$

Príklad:

```
void function quicksort (m,n) < int m,n;
{ int i,j;
    int v,x;
    if (n<=m) return
    /* začiatok fragmentu */
    i:=m-1; j:=n; v:=a[n];
    while (1) {
        do i:=i+1 while (a[i]<v);
        do j:=j-1 while (a[j]>v);
        if (iňj) break ;
        x:=a[i];a[i]:=a[j];a[j]:=x;}
    x:=a[i];a[i]:=a[n];a[n]:=x;
    /* koniec fragmentu */
    quicksort(m,j);quicksort(i+1,n)
}
```

Fragment programu v 3-adresovom kóde.

Vyňatie spoločných operácií pred cyklus a náhrada drahších operácií lacnejšími

Pr.	j:= j-1	t:= 4*j
	t:= 4*j	t:= t-4

Eliminácia zbytočných výpočtov a spoločných podvýrazov bloku B5.

```
B5:  t6:= 4*i      - t4 v B2
      x:= a[t6]    - vypočítané ako t v bloku B2
     t7:= 4*i
     t8:= 4*j      - t5 v B3
     t9:= a[t8]
     a[t7]:= t9    - t5 v B3
     t10 := 4*j
     a[t10]:= x
     goto B2
```

Optimalizovaný blok B5

```
B5:  x:= t3
     a[t7]:= t5
     a[t4]:= x
     goto B2
```

Analogicky sa optimalizuje blok B6.

Blokový diagram (graf toku riadenia)

B4: if $i < j$ goto B6

Alternatívy:

1. Absolutný kód.
2. BRC- binárny relokatívny kód.
3. Assembler.

```
Riadenie pamäti:

 štvorica + count (ofset kde začína kód)
```

Preklad inštrukcii:

```
Priamočiary spôsob (jednoadresný)

   a:= b+c          MOV b,R0
   d:= a+e          MOV c,R0
                    MOV R0,a ?
                    MOV a,R0 -
                    ADD e,R0
                    MOV R0,d

 a:=a+1             MOV a,R0       INC a
                    ADD \#1,R0
                    MOV R0,a        pamäťové cykly
```

Použitie registrov:

1. Výber premenných, ktoré sa budú vyskytovať v danom bode vykonávania programu v registroch.
2. Postupné vymieňanie hodnôt v registroch počas vykonávania programu.

Registre dvojnej dĺžky (dva po sebe idúce registre)

```
Cena inštrukcie:
                    1. načítanie a dekódovanie
                    2. možno adresa v ďaľšom slove
                    3. operand
dodatočná cena      4. zložité inštrukce ( MULT,DIV)
                    5. úroveň nepriamej adresy

  overlay inštukčný (pamäťových cyklov).

Volanie rekurzívneho podprogramu.

r:      MOV   r+a, called return address
        JUMP  called code begin
          .
          .
  n-1     .    activation record
          .
```

Dynamické podprogramy – stack

```
inicializácia zásobníka:



volanie:
      ADD  \# caller.recordsize ,SP
      MOV  r+a,*SP
      JUMP called code begin

called:
       .
       .
       .
       JUMP *0(SP)    {return, double indirection}

JE TO ASI ZBYTOČNÁ OZDOBA
```

Intervaly – basic blocks

```
   t1=a*a
   t2=a*b
   t3=2*t2
   t4=t1+t3
   t5=b*b
   t6=t4+t5
```

Algoritmus rozdelenia na základné bloky:

```
  Vstup:  Program (postupnosť štvoríc)
  Výstup: Základné bloky a graf toku riadenia.
  Metóda:
      Určenie hlavičiek intervalov.
         1.  Prvý príkaz je hlavička
         2.  Každý cieľ skoku je hlavička
         3.  Každý príkaz bezsprostredne nasledujúci
             za skokovým príkazom je hlavička.
Interval =  hlavička + postupnosť príkazov bezsprostredne
            nasledujúcich po následujúcu hlavičku.
Graf toku riadenia:
        Uzly - Intervaly
        Hrany- skoky
                      bezsprostredná následnosť za
                      podmieneným skokom.
```

7.1 Transformácie zachovávajúce štruktúru

1. Eliminácia spoločných podvýrazov
2. Eliminácia "mŕtveho bodu"
3. Premenovanie dočasných premenných
4. Výmena dvoch susedných príkazov
5. algebraické transformácie

```
  Použitie premenných
    i: x:= y op z            j používa x
    .                        ak existuje cesta z i do j,
    .                        po ktorej sa hodnota
    .                        x nemení
    j:      := x

Použitie premenných - "mŕtve" a "aktívne" premenné sa dajú
zistiť spätným prezeraním grafu toku riadenia.

     1. register describtor : čo je v danom  momente
                              v každom registri
     2. address describtor:   kde sa nachádzajú hodnoty
                              premenných v danom okamihu.
     3. getreg:               funkcia, ktorá pri prekalde
                              príkazu x:= y op z vráti register pre x.
```

Stratégia pre getreg

1. Ak register pre y (z) nenesie inú hodnotu a y (z) je mŕtva premenná vráti tento register.
2. Ak 1. neuspeje vráti voľný (prázdny) register.
3. Ak nie je žiaden register voľný a x má nasledujúce použite v tom istom bloku alebo x je použitý ako index vráti obsadený register R generuj inštrukciu: STORE R,M Dobrá stratégia je uvoľniť ten register, čo sa bude používať najneskoršie.
4. Ak sa x nepoužíva v danom bloku, alebo sa nedá nájsť vhodný register nájde sa miesto v pamäti.

```
Pr.
     Príkaz      code          Register descr.   Address descr.
     t:=a-b    MOV a,R0        R0 obsahuje t        t in R0
               SUB b,R0

     u:=a-c    MOV a,R0        R0 obsahuje t        t in R0
               SUB c,R1        R1 obsahuje u        u in R1

     v:=t+u    ADD R1,R0       R0 obsahuje v        v in R0
                               R1 obsahuje u        u in R1

     d:=v+u    ADD R1,R0       R0 obsahuje d        d in R0
               MOV R0,d
```

Priradenie registrov farbením grafu.

```
     1.   Predpokladáme neobmedzený počet registrov
     2.   register inference graph
               uzly - registre
               hrany $a \rigtharrow b$,  keď premenná v $a$ je živá, kde premenná v $b$ je
               definovaná.
     3.   k-registrov: Či graf  RIG sa dá zafarbiť k-farbami
               ( NP- úplný problém )

   Heuristika: Odstraňovanie vrcholov stupňa menšieho než k.
     1. Úspech graf sa postupne redukuje na prázdny
     2. Nedá sa použiť, zapamätáme nejaký register.


  Dag intervalu:

Pr.
     1: t1:=4*i
     2: t2:=a[t1]
     3: t3:=4*i
     4: t4:=b[t3]
```

Def: Uzol d dominuje uzol n, ak každá cesta z počiatku o do uzla n vedie cez d.

Algoritmus pre výpočet relácie dominancie D.

Vstup: graf $G =< N, E >$, počiatok o
Výstup: relácia D.

Metóda:

for *each* $n \in N$ **do** $D(n) := \{n\}$; *// inicializácia*
while *changes to any* $D(n)$ *occur* **do**
  for *each* $n \in N - \{o\}$ **do**
    $D(n) := D(n) \bigcup \{D(p) : \ hrana\ p \rightarrow n \in E\}$

Prirodzené cykly:

```
1.  cyklus má jediný vstupný bod --  hlavičku,  ktorá  dominuje  všetky
    uzly cyklu
2.  z každého uzlu cyklu existuje aspoň jedna cesta späť k hlavičke.

DP:  Prirodzený cyklus prislúchajúci spätnej hrane $n  \rightarrow  d$. Je  cyklus
     s hlavičkou $d$ obsahujúci všetky uzly z ktorých existuje cesta  do uzla $n$
     neprechádzajúca cez hlavičku $d$.

 Algoritmus konštrukcie prirodzeného cyklu k spätnej hrane $n \rightarrow d$
     Vstup:  graf $G$ a spätná hrana $n \rightarrow d$
     Výstup: všetky uzly prirodzeného cyklu k $n \rightarrow d$.
```

Metóda:

PROGRAM
  PROCEDURE insert(m);
    IF $m \neq d$ *and* $m \notin loop$ *THEN*
      BEGIN $loop := loop \cup \{m\}$;
        push m onto stack
      END;
  BEGIN
    $stack := empty$;
    $loop := \{d\}$;
    insert(n);
    WHILE stack is not empty DO
      BEGIN m:=top(stack);
        pop(stack);
        FOR each predecessor p of m DO
          IF d dominates p THEN insert(p)
      END
  END.

Transformácia dvoch cyklov so spoločnou hlavičkou na jeden cyklus:

Def: Graf G sa nazýva redukovateľný, ak:

1. Dopredné hrany tvoria acyklický graf, v ktorom každý uzol je dosiahnuteľný z počiatku.
2. Spätné hrany sú také, že výstupný uzol dominuje vstupný uzol.
3. Iné hrany graf neobsahuje.

7.2 Intervalová analýza

Def: Interval I(n) s hlavičkou n je množina uzlov taká, že:

1. $n \in I(n)$.
2. Ak pre nejaký uzol m všetci predchodcovia patria do $I(n)$, potom aj m patrí do $I(n)$.
3. Žiadny iný uzol nepatrí do $I(n)$.

Algoritmus intervalovej analýzy - rozdelenie grafu na disjunktné intervaly.
Vstup: graf $G =< N, E >$, $n_0 \in N$ počiatočný uzol
Výstup: rozdelenie grafu na disjunktné intervaly.

Metóda:

PROGRAM;
FUNCTION $I(n)$*;*
BEGIN $S := \{n\}$*;*
WHILE (existuje $m \neq n_0$*)(pre všetky p) také, že* $(p \rightarrow m \in E)$ *and* $p \in S$
DO $S := S \cup \{m\}$*;*
$I(n) := S$
END;
BEGIN $I(n_0)$*;*
WHILE exists m not yet selected but with a selected predecessor
DO $I(m)$
END.

Def: Intervalový graf.

1. uzly sú intervaly rozdelenia
2. počiatočný uzol $I(n_0)$
3. hrana $I \rightarrow J$ práve vtedy, ak existuje hrana $m \rightarrow n \in E$ v pôvodnom grafe a $m \in I$ a $n \in J$.

iterovaná intervalová analýza, limitný graf.

T1 – T2 analýza

T1: Ak n je uzol so slučkou (t.j. hrana $n \rightarrow n$), vynechaj slučku.

T2: Ak uzol n má jediného predchodcu, uzol m. Vynechaj uzol n a urob všetkých následníkov uzlu n následníkmi uzlu m.

VETA 1: Limitný graf z T1 -T2 analýzy a z iterovanej intervalovej analýzy je ten istý graf.

VETA 2: Ak graf je redukovateľný, limitný graf je jeden uzol.

Iteratívny algoritmus:

$in[B] := \bigcup_{p \prec B} out[P]$
$out[B] = gen[B] \cup (in[B] - kill[B])$.
for *each block* B **do** $out[B] := gen[B]$; *{ inicializácia}*
$change := true$;
while *change* **do**
begin $change := false$;
for *each block* B **do**
begin $in[B] := \bigcup_{p \prec B} out[P]$;
$change := true$
end
end