KATEDRA
INFORMATIKY
UNIVERZITA PALACKÉHO V OLOMOUCI

KMI/PRKL - Překladače

Poznámky z výuky *(2025 – 2026)*
Verze z *16. 03. 2026*

Vojtěch Netrh
vojtanetrh@gmail.com

Obsah

1 Režijní informace **3**
1.1 Státnicové okruhy 3

2 Historie vývoje **3**

3 Překlad programu **4**
3.1 Front End 4
3.2 Optimalizace 4
3.3 Back End 5
3.4 Křížový překladač 5
3.5 Tabulka symbolů 5
3.6 Interpret 5

4 Regulární gramatiky **5**
4.1 Regulární výrazy 6

5 Lexikální analýza / analyzátor **6**
5.1 Interní forma posloupnost symbolů 7
5.2 Rozpoznání konce lexikálního symbolu 7
5.3 Klíčová slova v programu 7
5.4 Chyby rozpoznávané při lexikální analýze 8

6 Syntaktická analýza **8**
6.1 Analýza shora-dolů LL 8
6.1.1 Expanze 8
6.1.2 Srovnání 8
6.1.3 Konstrukce zásobníkového automatu pro gramatiku LL(1) 9
6.1.4 Činnost automatu 9
6.2 Konstrukce analyzátoru metodou rekurzivního sestupu 10
6.2.1 Problémy 11
6.3 Syntaktická analýza zdola-nahoru 11
6.3.1 Konstrukce automatu pro SLR(1) gramatiku 12
6.3.2 Konstrukce automatu pro LALR(1) gramatiku 13
6.3.3 Nedeterminismus sestaveného automatu 14
6.3.4 Řešení konfliktů 14

1 Režijní informace

Vyučující: Arnošt Večerka

Zkouška: ústní; otázky buď podle probraných témat nebo na konci semestru

Zápočet: interpret příslušného programovacího jazyka (konkrétně Pascal)

Materiály: dostupné z cloudu katedry informatiky

1.1 Státnicové okruhy

1. Základní struktura překladače.
2. Fáze analýzy a syntézy překladu.
3. Lexikální analýza,její úloha a konstrukce lexikálního analyzátoru.
4. Syntaktická analýza shora-dolů, gramatiky LL(1).
5. Konstrukce syntaktického analyzátoru metodou rekurzivního sestupu.
6. Syntaktická analýza zdola-nahoru.
7. Konstrukce syntaktického analyzátoru pro gramatiky SLR(1), LALR(1).
8. Sémantická analýza.
9. Atributové gramatiky a jejich specifické typy pro analýzu shora-dolů a analýzu zdola-nahoru.
10. Interní formy programu.
11. Překlad základních příkazů programovacích jazyků do interní formy.
12. Tabulky symbolů.
13. Metody lokální a globální optimalizace programu.
14. Generování kódu, úloha přidělování registrů.

2 Historie vývoje

- Operační systémy
- Pouze instrukce (operační kódy) a čísla (uložené v paměti)
- Soubory s instrukcemi nebyly dříve tak bohaté
- Vylepšení ve formě Assembleru
  - návěští
  - označení instrukcí slovy
  - nutnost překladu ⇒ musíme vymyslet a používat překladač
- Další fází byly programovací jazyky
  - přenositelnost programu mezi systémy (procesory)
  - musí být příslušný překladač
  - překlad je mnohem složitější ⇒ přeložený program je méně efektivní oproti přímému zápisu
- `Fortran` jako první programovací jazyk
- K němu vytvořený překladač byl kvalitní (měl i optimalizace)
- Sestavovací program
- Jazyky `C, C++`
- Celý postup překladu
  - preprocesor
  - linker
  - objektový kód

    - strojový kód
    - ⋮
- Specifické interní reprezentace
    - bytecode pro `Javu` nebo CIL pro `C#`
    - jsou nezávislé na cílové platformě
    - JIT (just in time) překladač
- 2 základní formy překladače
    1. interní forma – posloupnost symbolů
    2. interní forma – nejběžnější jsou AST (abstraktní syntaktický strom) a lineární forma (čtveřice)

3 Překlad programu

Překlad má obvykle 3 fáze (průchody) – front end, optimalizace, back end. V prvních průchodu se udělá analýza zdrojového programu a vygeneruje se 2. interní forma. V druhém probíhá optimalizace. Ve třetím se už generuje cílový program.

> **Poznámka 1**
>
> Assembler měl překlad dvoufázový.

[Možná vložit obrázek]

První 2 fáze jsou nezávislé na cílové platformě. Pro různé platformy je tedy možné mít první 2 části společné a měnit jen část 3. (Back End).

3.1 Front End

Tato první část je závislá na zdrojovém jazyku. Dělí do 3 částí:
- lexikální,
- syntaktická,
- sémantická analýza.

Lexikální analýza pro atomické části programu (končí převodem do 1. interní formy). Syntaktická analýza dělá rozbor deklarací a příkazů (např. `a = b + ;` zda je korektní). Sémantická analýza řeší smysl (např. zda v `a = b + c` jsou korektní datové typy proměnných).

3.2 Optimalizace

Optimalizace není povinná, neboť nemá vliv na výslednou funkčnost (zda dá správný výsledek). Pro zvýšení rychlosti a efektivity. Může a nemusí být závislá na cílové platformě.

Základní dělení:
- lokální
- cyklů
- globální
- mezi funkcemi
- okénková (okno má určitý počet instrukcí; *peephole*)

Příklad 2

Ukázka na výpočtu konstantního výrazu.

```cpp
const float epsilon = 0.0001
if (u < 2 * epsilon) { }
```

Do čeho se přeloží (ne)optimalizovaný kód nahoře?

Rozebrat co v každém (i jednoduchém programu) musí překladač vykonat. Až na určení adres, výpočet výrazů, nahrazení různá atd.

3.3 Back End

Je závislá na cílové platformě (jdeme do strojového kódu). Volba příslušných instrukcí a přidělování registrů (je jich omezený počet $\Rightarrow$ efektivní využití). Okénkové optimalizace se provádí v této části. Prochází se krátké sekvence instrukcí následujících ihned za sebou, kde se dělají transformace.

3.4 Křížový překladač

Překladač, který generuje cílový program pro jinou platformu, než tu na které běží. Přeložený program se už přenáší do cílového systému hotový.

3.5 Tabulka symbolů

Jsou do ní ukládány informace z deklarativních částí programu. Např. jména, datové typy, počet dimenzí pole, deklarace funkcí jejich hlavičky, …

3.6 Interpret

Na rozdíl od překladače negeneruje cílový kód, ale zdrojový program přeloží do interní formy a následně interní formu interpretuje (udělá výpočet). Následuje po části *Front End*. Oproti překladači je výrazně jednodušší. Ale oproti překladu je výrazně pomalejší. Dříve se používal i v Javě, nyní JavaScript nebo Python. Programy v interpretovaných mají lepší přenositelnost.

4 Regulární gramatiky

Označovány jako typ 3. Jedná se o nejjednodušší formu (kladeno nejvíce nároků na pravidla).

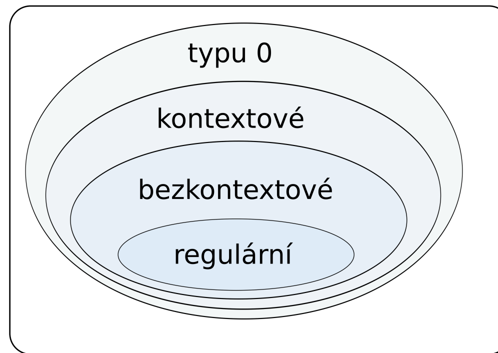
Pravidla: pro gramatiku $G = (N, T, P, S)$
$A \to aB$
$A \to a$

Věta **Theorem 3**

Ke každá regulární gramatice G existuje deterministický konečný automat A, který rozpozná jazyk generovaný gramatikou G. Platí tedy $L(A) = L(G)$.

Příklad regulární gramatiky **Příklad 4**

seznam pravidel, terminálů, neterminálů, atd.



Obrázek 1: Chomského hierarchie.

4.1 Regulární výrazy

Pro sestavení regulárních výrazů nad T, což je abeceda terminálů platí pravidla:

- každý symbol $x \in T$ je regulární výraz
- prázdný řetězec ε je regulární výraz
- jsou-li α a β regulární výrazy, pak jsou regulární výrazy i $\alpha\beta$, $\alpha \mid \beta$, α^+ a (α)
- pro a^* platí, že je to $\varepsilon \mid a^+$

[Opět možno uvést příklad terminálů a z něj regulárních výrazů, případně automat]

> **Konečný automat pro čísla v jazyce C** **Příklad 5**
>
> *[TBA]*

5 Lexikální analýza / analyzátor

Jedná se o první část překladače. Průběh je takový, že čte zdrojový program a vykonává činnost, konkrétně:

- rozpozná atomy (lexikální symboly) zdrojového programu
- program převede do 1. interní formy (posloupnost symbolů)
- vynechá všechny pro překlad nepotřebné části (formátování kódu, poznámky a komentáře).

Prostě to pak vypadá jak kdybychom klasický kód hodily na jediný řádek.

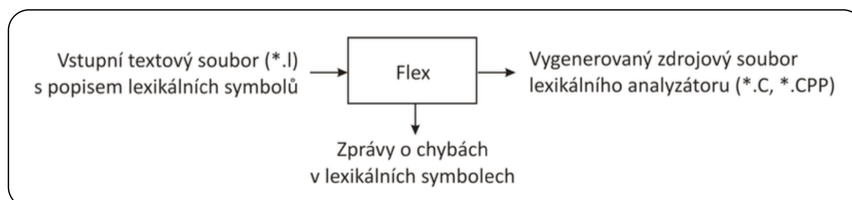
Atomy (lexikální symboly) jsou:

- klíčová slova – `int, typedef, enum, class, if, for, return`
- identifikátory – `i, suma, pocet`
- literály – `12 1.E5 "Evzen"`
- oddělovače – `,`
- omezovače –`[] {} () ;`
- operátory – `+ - * ++`
- další věci

Analýza syntaxe je rozdělena na dvě části (lexikální a syntaktickou analýzu) kvůli zjednodušení. Lexikální analyzátor funguje jako konečný automat. Syntaktický analyzátor už pracuje s 1. interní formou a tím je jednodušší generovaná bezkontextová gramatika.

V překladači se lexikální analyzátor nachází jako jedna funkce. Ta se volá vždy když navazující syntaktický analyzátor potřebuje další symbol pro analýzu.

Lexikální analyzátor je možné realizovat dvěma způsoby (1) ručně ho sestavit nebo (2) použít generátor lexikálních analyzátorů (program *Lex* či *Flex*).



Obrázek 2: Fungování analyzátoru Flex.

5.1 Interní forma posloupnost symbolů

Ke každému druhu lexikálního symbolu je v danám jazyce přiřazeno celé číslo. Obvykle se pro jednoznakové symboly používá číslo z ASCII, pro víceznakové se používají čísla nad ASCII (vyšší než 255). Víceznakové jsou třeba operátory `==` nebo `<<=`.

Pokud má symbol kromě typu i hodnotu (čili identifikátor nebo literál) lexikální analyzátor předává navíc i jeho hodnotu. U literálů převede hodnoty do příslušné interní formy (např. `12` je `int`, což bude běžně 32 bitové číslo nebo znak `"A"` je `char` a ten se převede do ASCII hodnoty a podle typu kódování systému se hodnota uloží na správný počet bitů).

5.2 Rozpoznání konce lexikálního symbolu

Ve zdrojovém programu jsou jednotlivé symboly za sebou ⇒ lexikální analyzátor je musí korektně oddělit (zjisti kde končí). Používá se způsob kdy lexikální analyzátor k právě rozpoznanému symbolu připojuje další následující znaky dokud vzniká korektní lexikální symbol (jakéhokoliv typu).

Např. výraz `b+++c` lze interpretovat dvěma způsoby (a) `b++ + c` nebo (b) `b + ++c`. Lexikální analyzátor zvolí variantu (a) neboť operátor `++` za `b` je delší než `+`.

```
program Matice ;
  var i , j of integer ;
  matice array [ 1 .. 10 , 1 .. 20 ] of integer ;
begin
  for i := 1 to 10 do
    for j := 1 to 20 do
      matice [ i , j ] := 10 * i + j
end .
```

Obrázek 3: Rozdělení kódu v Pascalu lexikálním analyzátorem.

5.3 Klíčová slova v programu

Obvykle se pro zápis klíčových slov používá stejná syntaxe jako pro identifikátory. Lze tedy zvolit dva přístupy k jejich rozpoznání:

1. klíčová slova rozpozná lexikální analyzátor
2. lexikální analyzátor je předa jako běžné identifikátory a jsou rozpoznána až v syntaktickém analyzátoru

U druhého případu (syntaktický analyzátor) se používá hashovací tabulka (minimální s perfektním hashováním), kde po rozpoznání identifikátoru je možná rychle zjistit zda se jedná o klíčová slovo.

V jazyce C se používá hashovací funkce $h(z_1, z_2, ..., z_k) = \text{kod}(z_1) + \text{kod}(z_k) + k$, kde *kod* označuje, že jednotlivá písmena nejsou kódována obvyklým způsobem, ale podle speciální tabulky.

5.4 Chyby rozpoznávané při lexikální analýze

Rozpoznány jsou chyby jako:

- identifikátor má větší počet znaků než jazyk připouští
- chybný zápis čísla
- číslo má nepřípustnou velikost hodnoty
- chybný zápis znaku
- chybný zápis řetězec
- řetězec má nepřípustnou délku
- na začátku dalšího symbolu je znak, který netvoří žádný lexikální symbol (v C je to `&` a `@`)

6 Syntaktická analýza

Syntaktický analyzátor rozpozná syntaktické celky programu (deklarace, výrazy, příkazy) a ty předá dál k sémantické analýze.

Syntaxi programovacího jazyka nelze popsat regulární gramatikou. Proto se pro tento popis používají bezkontextové gramatiky (typ 2). Syntaktický analyzátor je založen na deterministickém zásobníkovém automatu. Při této analýze se sestavuje derivační strom, který odpovídá překládanému programu. Máme 2 základní způsoby sestavení stromu – směrem dolů od kořene stromu a směrem nahoru od listů. Tyto konstrukce můžeme provést pouze pro určité podtřídy bezkontextových gramatik.

> **Poznámka 6**
>
> U analýzy shora-dolů *LL(k)* je kořenem stromu počáteční symbol gramatiky. U analýzy zdola-nahoru *LR(k)* jsou listy symboly interní formy, které vytvořil lexikální analyzátor.

LL jako **l**eft to right a **l**eft most.

LR jako **l**eft to right a **r**ight most.

6.1 Analýza shora-dolů LL

Používá se zásobníkový automat, který nevyužívá stavy, ale pouze zásobník. Pracuje s vrcholem (v každém kroku odebrán právě 1 symbol). Pro zásobníkové symboly se používá množina $N \cup T$ z výchozí gramatiky. Analýza se skládá ze 2 operací: expanze a srovnání.

6.1.1 Expanze

Operace probíhá v případě, kdy na vrcholu zásobníku je neterminální symbol. Expanze probíhá nahrazením tohoto symbolu pravou stranou příslušného pravidla.

6.1.2 Srovnání

Provádí se v případě kdy je na vrcholu zásobníku terminální symbol. Symbol je srovnán se symbolem na vstupu a pokud jsou shodné, tak se oba odstraní.

Automat rozpozná správný zdrojový program, pokud ho přijme celý, což se prokáže prázdným zásobníkem na konci.

Z třídy gramatik LL(k), které umožňují deterministickou analýzu se používají v překladačích výlučně LL(1). Což znamená, že se čte zleva doprava, používá se levá derivace a stačí znát jen jeden symbol vstupního řetězce (ten který je právě na vstupu).

LL(1) gramatika **Definice 7**

Bezkontextová gramatika je LL(1) gramatikou, jestliže každá 2 její pravidla se stejným symbolem na levé straně

$$A \to \alpha$$

$$A \to \beta$$

splňující podmínku disjunktnosti množin *First-Follow*
`First(a Follow(A)) ∩ First(b Follow(A)) = Ø`

Množina First **Definice 8**

Definována předpisem

$$\text{First}(\alpha) = \left\{ t \mid \alpha \overset{*}{\Rightarrow} t\mu, t \in T \right\} \cup \left\{ \varepsilon \mid \alpha \overset{*}{\Rightarrow} \varepsilon \right\}$$

Jedná se o množinu všech terminálů (α), které se mohou objevit na začátku řetězce použitím libovolných pravidel.

Množina Follow **Definice 9**

Definována předpisem

$$\text{Follow}(A) = \left\{ t \mid S \overset{*}{\Rightarrow} \mu A v, v \neq \varepsilon, t \in \text{First}(v) \right\} \cup \left\{ \varepsilon \mid S \overset{*}{\Rightarrow} \mu A \right\}$$

Množina všech terminálů, které se mohou kdykoliv vyskytnout bezprostředně za neterminálem A.

6.1.3 Konstrukce zásobníkového automatu pro gramatiku LL(1)

Postup je popsán v krocích:

1. pravidla gramatiky očíslujeme
2. pro každé pravidlo $A \to \alpha$ vypočítáme množinu `Firs(a Follow(A))`
3. sestavíme tabulku automatu (řádky jsou neterminály gramatiky, sloupce jsou terminály a ε)
4. jednotlivé pozice vyplníme tak, že do pozice, která je na řádku s neterminálem A a v sloupci se symbolem t napíšeme číslo pravidla (případně celé pravidlo), právě když t patří do množiny `First-Follow` tohoto pravidla (pokud neexistuje, tak pozice zůstane prázdná)

6.1.4 Činnost automatu

1. na počátku je vstupem řetězec terminálních symbolů gramatiky (interní forma zdrojáku, kterou vytvořil lexikální analyzátor) a na vrcholu zásobníku je počáteční symbol gramatiky
2. v každém kroku se provede *expanze* nebo *srovnání* podle toho co je na vrcholu zásobníku (viz 6.1.1 a 6.1.2)

3. když zásobník zůstane prázdný je zdrojový kód syntakticky správný

Příklad 10

[V PDF prekladace03 na straně 6.]

6.2 Konstrukce analyzátoru metodou rekurzivního sestupu

Pro každý neterminál A sestavíme jednu funkci, která udělá analýzu podle všech pravidel gramatiky s tímto symbolem na levé straně.

Značení **Poznámka 11**

u … proměnná, ve které je lexikální symbol naposledy předaný lexikálním analyzátorem
`lex()` … funkce lexikálního analyzátoru
`chyba()` … funkce pro ošetření (syntaktické) chyby v předkládaném programu

Pro pravidla se symbolem A na levé straně se spočítají jejich množiny *First-Follow* jako:
$A \rightarrow \alpha_1$ … $\texttt{First(}\alpha_1\texttt{Follow(A))} = \{a_{11}, a_{12}, ..., a_{1p}\}$
$A \rightarrow \alpha_2$ … $\texttt{First(}\alpha_2\texttt{Follow(A))} = \{a_{21}, a_{22}, ..., a_{2p}\}$
až po k.

Funkce pro analýzu pak vypadá takto:

```
1   ParseA()
2   { switch (u) {
3     case a11:
4     case a12:
5     ...
6     case a1p: // akce pro α1
7     case a21:
8     case a22:
9     ...
10    case a2q: // akce pro α2
11    ...
12    case ak1:
13    case ak2:
14    ...
15    case akr: // akce pro αk
16    default: chyba()
17    }
18  }
```

Poznámka 12

Symbol `#` označuje konec analyzovaného programu (nahrazuje ε).

Pro gramatiku s pravidly:

$$E \rightarrow TF$$

$$F \rightarrow +TF \mid \varepsilon$$

$$T \to i \mid (E)$$

máme takovýto program:

```
Vim
1  Srovnani(v)
2  {
3  if (u==v) u=lex(); else chyba();
4  }
5  ParseE()
6  {
7    switch (u)
8    { case '(':
9      case 'i': ParseT(); ParseF(); return; // E→TF
10     default: chyba(); }
11 }
12 ParseF()
13 {
14   switch (u) {
15     case '+': u=lex(); ParseT(); ParseF(); return; // F→+TF
16     case ')':
17     case '#': return; // F→ε
18     default: chyba(); }
19 }
20 ParseT()
21 {
22   switch (u) {
23     case 'i': u=lex(); return; // T→i
24     case '(': u=lex(); ParseE(); Srovnani(')'); // T→(E)
25       return;
26     default: chyba(); }
27 }
```

6.2.1 Problémy

Může nastat, že některá pravidla popisující daný programovací jazyk nevyhovují pravidlům LL(1) gramatiky. Jsou možná řešení:

1. konflikty odstranit transformacemi nebo
2. konflikty v gramatických pravidlech ošetřit ručně při sestavování funkcí pro rekurzivní sestup.

Postup transformace

Mělo by být v transformace LL(1).pdf

6.3 Syntaktická analýza zdola-nahoru

Opět se používá zásobníkový automat, který využívá pouze zásobník. Specifická varianta která pracuje s celým zásobníkem = při přechodu může být odebrán **libovolný počet** symbolů. Konstrukce automatů pro tuto analýzu je značně složitější než pro analýzu shora-dolů.

Na začátku je zásobník prázdný. Pokud je vstupní věta rozpoznána, obsahuje počáteční symbol gramatiky.

Analýza se skládá ze 2 operací:

- **přesun** … symbol se přesune ze vstupu na zásobník
- **redukce** … na zásobníku nahradíme pravou stranu pravidla za neterminální symbol, který je na straně levé

Ukázka operací a stavu vstupu a zásobníku **Příklad 13**

vstup	zásobník	operace
$(i+i)*i-i$		přesun
$i+i)*i-i$	$($	přesun i
$+i)*i-i$	$(\ i$	redukce podle $E \to i$
$+i)*i-i$	$(\ E$	přesun $+$
$\vdots$	$\vdots$	$\vdots$

Deterministickou analýzu zdola-nahoru umožňuje třída LR (**L** … vstupní řetězec se čte zleva doprava a **R** … třída je založena na pravé derivaci). Tato třída má 4 podtřídy – $LR(0)$, $SLR(1)$ [1], $LALR(1)$[2] a $LR(1)$. *Čím více je třída na začátku tím je menší a má jednodušší automaty, čím více je na konci tím je větší a má složitější automaty.* Číslo v závorce určuje kolik symbolů ze vstupu je nutné znát pro deterministickou činnost.

6.3.1 Konstrukce automatu pro SLR(1) gramatiku

Pro takovýto deterministický automat musíme zásobníkové symboly sestavit jako množiny podle pravidel gramatiky (nestačí pouze její terminály a neterminály). Výchozí gramatiku G si rozšíříme o $N' = N \cup \{S'\}$ a $P' = P \cup \{S \to S'\}$. Tím si usnadníme zjištění kdy je činnost ukončena (z pohledu stromu se přidá 1 uzel nad kořen – bude to nový kořen).

Zásobníkové symboly nazveme jako **množiny položek pravidel gramatiky**. Takováto položka je zápis pravidla s vyznačením místa, kde je právě fáze přesunu. Tvar vypadá takto $[A \to \mu.v]$. Tečka na pravé straně pravidla (tedy vpravo od šipky) odděluje již přesunuté symboly (tady μ) od nepřesunutých (tady v). Pro pravidlo $A \to aBC$ máme tedy 4 možné kombinace $[A \to .aBc]$ (začátek), $[A \to a.Bc]$, $[A \to aB.c]$, $[A \to aBc.]$. Specifická jsou pravidla typu $A \to \varepsilon$, ty jsou ihned redukční.

Při sestavování nového zásobníkového symbolu se vytváří **uzávěr $U(M)$** výchozí množiny položek gramatiky M. Do uzávěru se pro každou položku v M, která má právě na přesunu neterminální symbol, přidají výchozí položky všech pravidel s tímto neterminálním symbolem na levé straně. Jinak řečeno jde o pravidla jejichž redukcí lze realizovat přesun daného neterminální symbolu. Výpočet může mít tedy rekurzivní charakter, neboť nově přidaná položka může mít na přesunu (tedy za tečkou) opět neterminální symbol.

Uzávěr $U(M)$ **Definice 14**

$$U(M) = M \cup \{[B \to .\beta] \mid [A \to \mu.Bv] \in U(M), B \to \beta \in P\}$$

[V krocích lze velmi přesně popsat postup sestavení automatu].

[Taktéž lze v krocích popsat činnost automatu.]

[1] Simple
[2] Look-Ahead

Větná forma ... řetězec, který lze odvodit z počátečního symbolu gramatiky (průběžný krok věty)

Věta ... řetězec terminálů, který lze odvodit z počátečního symbolu

Fáze ... levá část větné formy

Prediktivní množina **Definice 15**

Prediktivní množina $\omega \subseteq T \cup \{\#\}$, kde # je konec překládaného programu. Jde o množinu, která se používá pro rozhodnutí kdy lze provést operaci redukce.

6.3.2 Konstrukce automatu pro LALR(1) gramatiku

Výchozí gramatiku G si opět rozšíříme o $N' = N \cup \{S'\}$ a $P' = P \cup \{S \to S'\}$ (tedy stejným způsobem jako pro *SLR(1)* gramatiku).

Zásobníkové symboly nazveme opět jako **množiny položek pravidel gramatiky**. Jejich tvar je ale lehce odlišný, konkrétně navíc obsahuje prediktivní množinu symbolů (ω) používanou pro rozhodnutí kdy lze provést operaci redukce. Zápis má tvar

$$[A \to \mu.v; \omega]$$

kde $\omega \subseteq T \cup \{\#\}$ (# ... značí konec překládaného programu).

Pravidlo se může opět nacházet ve 4 různých fázích.

Pokud je řetězec $v = a\eta$, kde $a \in T$ (terminál) je situace zřejmá. Pokud však $v = B\eta$, kde $B \in N$ (neterminál) musí být proveden přesun pravé strany některého z pravidel s tímto symbolem B na levé straně a následně pak redukce pravé stran β tohoto pravidla. Položka $[B \to .\beta; ...]$ pak bude taky obsažena v zásobníkovém symbolu.

Opět se provádí výpočet **uzávěru $U(M)$** pro nějž platí:

$$U(M) = M \cup \{[B \to .\beta; \tau] \mid [A \to \mu.Bv; \omega] \in U(M),\ B \to \beta \in P,\ \tau = \text{First}(v\omega)\}$$

Při konstrukci automatu pro LALR(1) jsou slučovány zásobníkové symboly se stejným jádrem. **Jádro zásobníkového symbolu** je tvořeno jeho položkami, ve kterých jsou vynechány prediktivní množiny. Neboli

$$z = \{[A_1 \to \mu_1.v_1; \omega_1]...[A_1 \to \mu_k.v_k.\omega_k]\}$$

$$\text{kernel}(z) = \{[A_1 \to \mu_1.v_1]...[A_1 \to \mu_k.v_k]\}$$

V tomto slučování je oproti konstrukci pro LR(1) rozdíl, neboť tam slučování neprobíhá. Automat LR(1) má tedy více zásobníkových symbolů ⇒ zvyšuje jeho rozpoznávací schopnosti. Třídy jazyků LR(1) je tedy obsáhlejší než třída LALR(1).

Postup sestavení automatu (kroky)

1. Z pravidla s počátečním symbolem na levé straně zkonstruujeme počáteční zásobníkový symbol $z_0 = U(\{[S' \to .S; \#]\})$. Zařadíme ho do množiny Z ($Z = \{z_0\}$).
2. *lorem ipsum*
3. *lorem ipsum*
4. *lorem ipsum*
5. *lorem ipsum*
6. *lorem ipsum*

Činnost automatu (kroky)

1. Počáteční podmínky: na vstupu je vstupní řetězec a na zásobníku je symbol z_0
2. *průběžné kroky*
3. $\vdots$

> **Příklad činnosti automatu LALR(1)** **Příklad 16**
>
> *TBA*

6.3.3 Nedeterminismus sestaveného automatu

Výše sestavený automat je nedeterministický, neboli umožňuje více akcí, čímž vzniknou konflikty. **Konflikty** máme 2 druhů:

- **přesun-redukce** – na dané pozici automatu je akce přesunu i redukce, vznikne pokud je v zásobníkovém symbolu položka pro přesun symbolu t a také položka pro redukci symbolu t

$$z = \{...[A \to \mu.tv]...[B \to \beta.]...\} \quad t \in \text{Follow}(B)$$

- **redukce-redukce** – na pozici automatu jsou 2 různé akce redukce, v zásobníkovém symbolu musí být 2 redukční položky pro stejný symbol t

$$z = \{...[A \to \alpha.]...[B \to \beta.]...\} \quad t \in \text{Follow}(A) \wedge t \in \text{Follow}(B)$$

> **Komplikovanější konflikty** **Poznámka 17**
>
> Výjimečně se mohou vyskytnou i komplikovanější konflikty jako *přesun + 2 redukce*.

6.3.4 Řešení konfliktů

Prakticky se používá pouze možnost **přepsání pravidel gramatiky**, které konflikt způsobují. Gramatiky obsahuje konflikty pokud **není jednoznačná**. Tento přepis obvykle udělá gramatiku rozsáhlejší.

> **Příklad 18**
>
> *TBA*

Konflikty se případně mohou řešit i ručně, kdy se pro každé políčko automatu, kde je konflikt, rozhodne co se tam má nechat a zbytek se odstraní.

Při těchto konfliktech (čili i analýzách) je vhodné si nakreslit derivační strom, který umožní na první pohled vidět, která z operací se má provést dříve a je tedy korektní. Např. víme, že obvykle je v programovacích jazycích součet levě asociativní, proto tuto variantu v tabulce ponecháme.