Fakulta matematiky, fyziky a informatiky
Univerzita Komenského v Bratislave

UNITY Interpreter

Diplomová práca

Martin Šebík

BRATISLAVA, 2008

UNITY Interpreter

DIPLOMOVÁ PRÁCA

Martin Šebík

UNIVERZITA KOMENSKÉHO V BRATISLAVE
FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY
KATEDRA INFORMATIKY

Študijný odbor: Informatika

Vedúci diplomovej práce
RNDr. Damas Gruska, PhD.

BRATISLAVA 2008

Čestne prehlasujem, že som túto diplomovú prácu vypracoval samostatne s použitím uvedenej literatúry a elektronických dokumentov.

.....

podpis

Poďakovanie

Ďakujem diplomovému vedúcemu RNDr. Damasovi Gruskovi, PhD. za cenné rady a pripomienky pri písaní tejto práce.

Abstrakt

Výpočtový model a dokazovací systém *UNITY* bol vyvinutý za účelom návrhu a dokazovania vlastností konkurentných paralelných programov. Skratka UNITY vznikla z prvých písmen anglických slov *Unbounded Nondeterministic Iterative Transformations*. Tento názov zahŕňa jeho dve hlavné črty - neohraničenosť a nedeterministickosť. Medzi ďalšie významné vlastnosti patrí paralelizmus a nezávislosť od architektúry.

Diplomová práca popisuje programovú notáciu UNITY, jeho syntax a sémantiku. Súčasťou práce je implementácia UNITY interpretera, jeho popis a prehľad teórie kompilátorov.

Kľúčové slová: UNITY, interpreter, Delphi Lex & Yacc, úvod do teórie kompilátorov

Obsah

Zoznam obrázkov

Zoznam tabuliek

Kapitola 1

Úvod

Systém *UNITY* (*Unbounded Nondeterministic Iterative Transformations*) predstavuje výpočtový model a dokazovací systém, navrhnutý v knihe *Parallel Program Design: A Foundation* od autorov K. Mani Chandy a Jayadev Misra [CM88].

Jeho hlavnými črtami sú:

- neohraničenosť – výpočet programu prebieha donekonečna, každý príkaz sa vykoná nekonečne veľa krát
- paralelnosť – možnosť vykonať viac príkazov súčasne
- nedeterministickosť / absencia toku riadenia – príkazy sa vykonávajú nedeterministicky
- nezávislosť na architektúre – detaily sa doriešia pri mapovaní na danú architektúru

Vďaka týmto vlastnostiam sú navrhované programy dostatočne abstraktné na to, aby sa pozornosť programátora sústredila na samotné jadro problému. Nezávislosť od architektúr, ktoré postupom času zastarajú a budú nahradené novými, nám zaručuje dlhú životnosť programov.

Výpočtovým modelom UNITY sú neohraničené nedeterministické iteratívne zmeny programového stavu, ktoré sú reprezentované zloženými priraďovacími príkazmi.
Dokazovací systém je oddelený od samotného programu a nie je v tejto práci popísaný.

Návrh programu v UNITY sa skladá z písania programov a mapovania na architektúry. Správnosť programu závisí len od programu, zatiaľ čo zložitosti (časová, priestorová) závisia od implementácie na danej architektúre.

Prehľad architektúr:

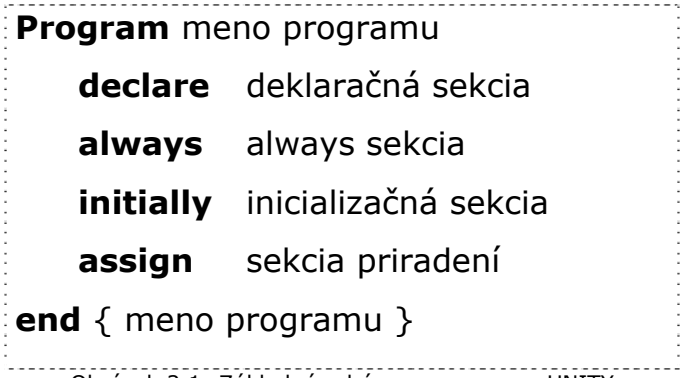
- *asynchrónne modely so zdieľanou pamäťou* – pevne stanovená množina procesorov a pamäti, spolu s informáciou, ktoré procesory môžu pristupovať do ktorej časti zdieľanej pamäte

- *distribuované systémy* - pevne určená množina procesov a kanálov, pričom každý kanál má buffer a priradené dva procesory – jeden ktorý do neho správy posiela (ak buffer nie je plný) a druhý, ktorý správy prijíma (ak buffer nie je prázdny); poradie správ v kanáli ostáva zachované

- *synchrónne modely* – na rozdiel od asynchrónneho modelu majú procesory spoločné hodiny, a pri každom tiku hodín vykonajú procesory inštrukciu

Kapitola 2

Popis UNITY programov



Obrázok 2.1: Základná schéma programu v UNITY

Na začiatku sa nachádza kľúčové slovo **Program**, za ktorým sa uvedie meno programu. Nasledujú jednotlivé sekcie v tomto poradí: *deklaračná sekcia*, tzv. *always sekcia*, *inicializačná sekcia* a ako posledná, *sekcia priradení*. Každá sekcia sa skladá z *kľúčového slova* (**declare**, **always**, **initially**, **assign**), za ktorým nasleduje jej *telo*. Kľúčové slovo sa vynecháva v prípade, keď je telo príslušnej sekcie prázdne.

Deklaračná sekcia obsahuje deklarácie premenných, always sekcia dáva premenné do vzájomného vzťahu, inicializačná sekcia inicializuje hodnoty premenných a sekcia priradení obsahuje množinu priraďovacích príkazov. Budeme predpokladať, že aspoň jedna zo sekcií *always* a *assign* je neprázdna[1] a každá premenná použitá v programe je uvedená v deklaračnej sekcii. V literatúre sa zvykne vynechávať i deklaračná sekcia, avšak v tom prípade musí byť z kontextu zrejmé, akého typu sú jednotlivé premenné programu.

Na záver sa uvedie kľúčové slovo **end**, za ktorým zvykne nasledovať meno programu ako komentár, ktorý sa píše v zložených zátvorkách.

[1] Ak je assign sekcia prázdna, tak sa vynechá aj inicializačná sekcia.

2.1 Deklaračná sekcia

Deklaračná sekcia obsahuje deklarácie všetkých premenných, ktoré sa v programe používajú, pričom každá premenná tu musí byť uvedená nie viac ako jeden krát. Jednotlivé deklarácie sú v tvare:

názov premennej : *typový výraz*

Typový výraz je buď základný, alebo zložený typ.
Medzi *základné typy* patria:

- **Integer** – celé číslo
- **Boolean** – pravdivostná hodnota (**true** / **false**)

Vo väčšine prípadov sú tieto základné typy v UNITY postačujúce, avšak v prípade potreby možno použiť aj ďalšie[2], ktoré poznáme z klasických programovacích jazykov (Pascal, C), napríklad typ **Float** – reálne číslo, typ **String** – reťazec, a iné...

Medzi *zložené typy* patrí *n*-rozmerné pole, ktoré sa deklaruje v tvare:

názov poľa **: Array**[$r_{1,1}$**..**$r_{1,n}$, ..., $r_{m,1}$**..**$r_{m,n}$] **of** *základný typ*

pričom m, $n > 0$ sú konečné prirodzené čísla, určujúce rozmer poľa a $r_{i,j}$ je konštantný celočíselný výraz, ktorý určuje jeho hranice.

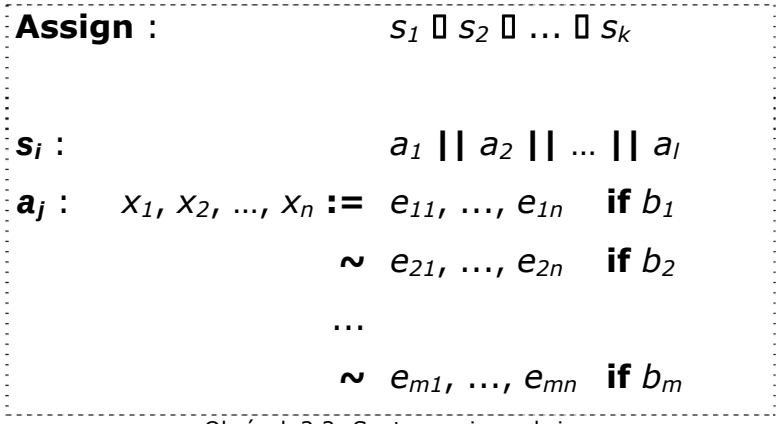
Ak sú typové výrazy premenných x_1, ..., x_n identické (x_1:t, x_2:t, ..., x_n:t kde t je typový výraz), tak deklaráciu týchto premenných môžeme zapísať v skrátenom tvare:

x_1, x_2, ..., x_n : t

2 záleží od konkrétneho UNITY kompilátora

2.2 Sekcia priradení

Syntax tejto sekcie sa dá zjednodušene zapísať nasledovne:



Obrázok 2.2: Syntax assign sekcie

Telo sa skladá z *príkazov* s_1, ..., s_k ($k \geq 1$ je konečné), ktoré sú oddelené znakom obdĺžnik, predstavujúcim *nedeterminizmus*. V každom kroku výpočtu programu sa nedeterministicky vyberie práve jeden príkaz s_i a ten sa vykoná.

Každý príkaz sa skladá z *priradení* a_1, ..., a_l [3] ($l \geq 1$ je konečné a pre každé s_i môže byť rôzne), ktoré sú oddelené znakom „dve čiary", predstavujúcim *paralelizmus*. Ak sa má vykonať daný príkaz s_i, tak sa vykonajú všetky jeho priradenia paralelne a navzájom nezávisle. Priradenie môže byť buď podmienené, alebo nepodmienené.

2.2.1 Podmienené priradenie

Podmienené priradenie sa skladá z jednej *ľavej strany*, ktorá obsahuje premenné x_1, ..., x_n a z *m pravých strán*, pričom každá z nich sa skladá z *výrazov* e_{i1}, ..., e_{in} a *podmienky* b_i ($n, m \geq 1$ sú konečné a pre každé a_i môžu byť rôzne[4]).

---

3 presný formálny zápis: $a_{si,1}$, ..., $a_{si,l}$

4 presný formálny zápis: $x_{ai,1}$, ..., $x_{ai,n}$:= $e_{ai,j1}$, ..., $e_{ai,jn}$ if b_j ; $1 \leq j \leq m$

Sémantika podmieneného priradenia: ak je splnená podmienka b_i, potom sa vykoná priradenie $x_1, ..., x_n := e_{i1}, ..., e_{in}$ tak, že sa naraz vyhodnotia indexy polí na ľavej strane, všetky výrazy na pravej strane a následne sa ich hodnoty paralelne priradia premenným na ľavej strane ($x_j \leftarrow e_{ij}$ pre všetky $1 \leq j \leq n$).

Ak sa premenná x na ľavej strane vyskytuje viac ako jeden krát, tak musí byť do nej priradená vždy rovnaká hodnota, formálne:

$\forall\ i, j : 1 \leq i, j \leq n : x_i = x_j \Rightarrow h(e_i) = h(e_j)$; h je hodnota daného výrazu

V prípade, že je viac podmienok b_i splnených, tak zodpovedajúce hodnoty výrazov k nim priradených musia byť rovnaké, formálne:

$\forall\ i, j : 1 \leq i, j \leq n : g(b_i) \wedge g(b_j) \Rightarrow h(e_{i1},..., e_{in}) = h(e_{j1},..., e_{jn})$; kde

g je pravdivostná hodnota booleovského výrazu - podmienky a

h je usporiadaná n-tica hodnôt daných výrazov.

2.2.2 Nepodmienené priradenie

Nepodmienené priradenie je špeciálny typ podmieneného priradenia, v ktorom platí $m = 1$ a $b_1 = \text{true}$. Ide teda o jednoduché priradenie $x_1, ..., x_n := e_1, ..., e_n$, ktorého sémantický význam je rovnaký ako v prípade podmieneného priradenia.

Príklady

$x, y := 1, 2$

- priradí v jednom kroku hodnotu 1 do premennej x a súčasne hodnotu 2 do premennej y; toto priradenie sa dá zapísať i pomocou príkazu $x := 1 \ || \ y := 2$

$x, y := y, x$ if $x > y$

- vymení obsah premenných, ak je x väčšie ako y

2.2.3 Kvantifikované priradenie

Doteraz uvažované priradenia boli *vymenované* (*enumerated*). V UNITY existujú i *kvantifikované priradenia* (*quantified assignments*), pomocou ktorých dokážeme v skrátenej forme zapísať konečnú množinu priradení.

Zjednodušený tvar *kvantifikovaného priradenia* je:

<□ i_1, ..., i_n **:** *bool*(i_1, ..., i_n) **::** *príkazy*(i_1, ..., i_n)**>** ; $n \geq 1$ konečné

resp.

<|| i_1, ..., i_n **:** *bool*(i_1, ..., i_n) **::** *príkazy*(i_1, ..., i_n)**>** ; $n \geq 1$ konečné

Premenné i_1, ..., i_n sa nazývajú *viazané* / *lokálne*. Uvažujú sa všetky *prípady* (kombinácie hodnôt viazaných premenných), v ktorých je po dosadení týchto hodnôt booleovský výraz *bool* splnený. Príkazy v týchto vyhovujúcich prípadoch sa podľa znaku □ resp. || vykonávajú buď nedeterministicky (□), alebo paralelne (||) tak, že viazané premenné, ktoré obsahujú sa nahradia príslušnými hodnotami i_1, ..., i_n. Tieto príkazy môžu byť opäť kvantifikované, avšak požadujeme, aby takéto „vnorenie" bolo konečné. Taktiež požadujeme, aby počet vyhovujúcich prípadov bol konečný. Tieto predpoklady nám zaručujú, že množina príkazov bude konečná.

Booleovský výraz *bool*(i_1, ..., i_n) môže okrem viazaných premenných obsahovať i *neviazané* / *globálne* premenné, ktorých hodnoty sa počas vykonávania programu nesmú meniť. To nám zaručuje, že množina príkazov bude pevná.

Príklad. Kvant. priradenie <|| i, j : $1 \leq i, j \leq 10$ :: A[i, j] := 0> v jednom kroku vynuluje dvojrozmerné pole A. Ide o skrátený zápis 100 priradení A[1, 1] :=0 || ... || A[1, 10] :=0 || A[2, 1] :=0 || ... || A[10, 10] :=0

2.2.4 Kvantifikované výrazy

Kvantifikáciu môžeme využiť i pri písaní výrazov. Všeobecný tvar *kvantifikovaného výrazu* je:

<*Op* $i_1, ..., i_n$ **:** *bool*($i_1, ..., i_n$) **::** *výraz*($i_1, ..., i_n$)**>** ; $n \geq 1$ konečné

Op predstavuje asociatívnu a komutatívnu binárnu operáciu. Hodnota kvantifikovaného výrazu je definovaná ako výsledok aplikovania tejto operácie na množine výrazov, získaných po substitúcii viazaných premenných vo vnútornom výraze.
Ak je množina výrazov po substitúcii prázdna, potom kvantifikovaný výraz nadobúda hodnotu neutrálneho prvku operácie.

Operácia	**Neutrálny prvok**
min	∞
max	-∞
+	0
*	1
∧	true
∨	false
≡	true

Tabuľka 2.1: Prehľad operácií a ich neutrálnych prvkov

Príklady

<+ n : $0 \leq n \leq$ N :: z^n >

- výraz predstavuje sumu $\sum_{n=0}^{N} z^n$

<max i, j, k : $1 \leq i, j, k \leq$ N :: $A[i, j, k]$ >

- vráti maximálny prvok z trojrozmerného poľa *A*

2.3 Inicializačná sekcia

Špecifikuje počiatočné hodnoty niektorých premenných. Syntax je rovnaká ako v sekcii priradení s tým rozdielom, že namiesto znaku priradenia **:=** sa používa znak rovnosti **=**.

Množina rovností je *správna (proper)*, práve vtedy ak:

1. sa premenná nachádza na ľavej strane rovnosti najviac jeden krát
2. existuje usporiadanie rovností také, že každá premenná v kvantifikácii je buď viazaná alebo sa nachádza na ľavej strane nejakej predchádzajúcej rovnosti
3. existuje usporiadanie všetkých rovností po kvantifikácii také, že každá premenná, uvedená na pravej strane alebo v indexe, sa nachádza na ľavej strane niektorej z predchádzajúcich rovností

Inicializačná sekcia predstavuje správnu množinu rovností. Z nich sa dá zostaviť najsilnejší predikát, platiaci na začiatku vykonávania programu – tzv. *initial condition*: znaky a || zameníme za znak $\wedge$ a podmienené priradenia tvaru $x = e_0$ if $b_0 \sim \ldots \sim e_n$ if b_n za výrazy
$(b_0 \Rightarrow (x = e_0)) \wedge \ldots \wedge (b_n \Rightarrow (x = e_n))$

Príklady

initially $N = 10$ <|| $i : 1 \leq i \leq N$ :: A[i] := i >

- množina rovností je správna, existujú požadované usporiadania
- nemožno zameniť za ||, pretože by došlo k porušeniu 2. podmienky pre správnu množinu rovností

initially $x = 0$ if $y > 0$

- ekvivalentná s rovnosťou $x = 0$ if $y > 0 \sim x$ if $y \leq 0$
- táto množina obsahujúca jednu rovnosť nie je správna v zmysle definície – porušená je 3. podmienka

2.4 Always sekcia

Definuje niektoré premenné ako funkcie iných premenných. Syntax je zhodná so syntaxou inicializačnej sekcie. Always sekcia teda predstavuje konečnú množinu rovností, ktoré budú platiť vždy, počas celého priebehu výpočtu. Z nich sa dá zostrojiť množina *invariantov* podobným spôsobom, ako initial condition z inicializačnej sekcie.

Premenná na ľavej strane rovnosti sa nazýva *transparentná*. Transparentná premenná je funkciou netransparentných premenných a nenachádza sa na žiadnej ľavej strane

1. rovnosti z inicializačnej sekcie
2. priradenia z assign sekcie

Môže sa však vyskytovať na pravých stranách. Aby sme zaručili, že každá transparentná premenná je dobre definovanou funkciou netransparentných premenných, tak požadujeme rovnaké podmienky ako v inicializačnej sekcii.

Always sekcia nie je nevyhnutná pre písanie programu, pretože každý výskyt transparentnej premennej v nasledovných sekciách môžeme nahradiť jej definíciou (netransparentnými premennými) bez zmeny sémantiky programu. Avšak použitie tejto sekcie prináša so sebou niekoľko výhod – prehľadnosť, efektívnosť, invarianty, ...

Príklad

always *celkovy_pocet = pocet_muzov + pocet_zien*

- ak sa zmení hodnota premennej *pocet_muzov* alebo *pocet_zien*, potom sa „automaticky zmení" i hodnota transparentnej premennej *celkovy_pocet*
- do premennej *celkovy_pocet* nemôžeme priamo priraďovať hodnoty, pretože „by sme stratili informáciu o počte mužov a žien"

2.5 Výpočet programu

Vo všeobecnosti prebieha výpočet UNITY nasledovne:

1. zadeklarujú sa premenné uvedené v *declare* sekcii

2. prebehne inicializácia premenných, ktoré sú uvedené na ľavých stranách rovností *initially* sekcie; premenné, ktoré ostanú neinicializované, nadobúdajú ľubovolnú (náhodnú) počiatočnú hodnotu

3. začne prebiehať nekonečný cyklus: v každej iterácii sa nedeterministicky vyberie jeden priraďovací príkaz z *assign* sekcie a ten sa vykoná - každý príkaz sa vykoná nekonečne veľa krát

Stav programu, ktorý sa ďalším vykonaním nezmení, sa nazýva *pevný bod* programu (*Fixed Point*). Dosiahnutie pevného bodu je ekvivalentné terminácii v terminológii štandardného sekvenčného programovania. Tento fakt môžeme využiť pri implementácii – ak nastane pevný bod, výpočet ukončíme (ďalšie vykonávanie by bolo zbytočné).

Príklady

$i := i + 1$ if $i < N$; kde N je konštanta

- pevný bod nastane, keď i nadobudne rovnakú hodnotu ako N

$x := y + 1$ ▯ $y := x + 1$

- pevný bod neexistuje, pretože vždy sa zmení hodnota premennej x alebo y

Kapitola 3

Kompilátory

Cieľom tejto kapitoly je poskytnúť prehľad základných pojmov a postupov z teórie kompilátorov, ktoré sa využívajú pri návrhu a implementácii interpretera UNITY. Kapitola predpokladá základnú znalosť teórie formálnych jazkov a automatov, ktorú možno nájsť v [HU78].

3.1 Úvodné pojmy

Kompilátor je aplikácia, ktorá číta program v *zdrojovom jazyku* a prekladá ho do ekvivalentného programu *cieľového jazyka*. Ak počas tohto procesu nájde chybu vo vstupnom programe, poskytne túto informáciu užívateľovi.

Fázy kompilátora:[5]

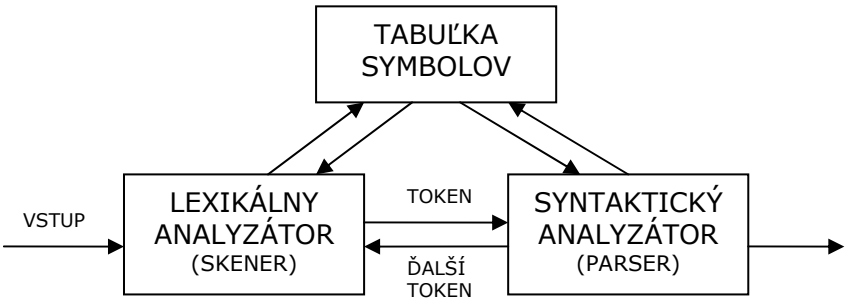
1. Lexikálna analýza
2. Syntaktická analýza
3. Sémantická analýza
4. Generovanie *medzikódu*[6]
5. Optimalizácia medzikódu
6. Generovanie kódu

Interpreter je aplikácia, ktorá číta program v zdrojovom jazyku a jeho príkazy priamo vykonáva - interpretuje. Od kompilátora sa teda líši v poslednom kroku. Príklady známych interpreterov: Perl, PHP, Python, MATLAB, ...

5 medzi týmito fázami existuje zdieľaná správa tabuľky symbolov a správa chýb

6 strojovo nezávislý kód

3.2 Lexikálna analýza



Obrázok 3.1: Interakcia medzi skenerom a parserom

Lexikálny analyzátor / *skener* číta na vstupe sekvenčný súbor a rozpoznáva vzory popisujúce tzv. tokeny, ktoré posiela do ďalšej fázy.

Token je gramatická trieda reprezentujúca skupinu reťazcov s rovnakým syntaktickým významom, zodpovedajúcim jednému patternu. Je výstupom lexikálneho analyzátora a vstupom syntaktického analyzátora, v ktorom predstavuje terminál. Medzi tokeny patria identifikátory, konštanty, operátory, rezervované slová a iné..

Pattern / *vzor* je regulárny výraz popisujúci nejaký token. *Kleeneho regulárne výrazy* sú definované nasledovne:

1. ε, $a \in \Sigma$ sú regulárne výrazy popisujúce jazyky L(ε) resp. L(*a*)
2. ak *a*, *b*, *c* sú regulárne výrazy popisujúce jazyky L(*a*), L(*b*), L(*c*), tak *ab*, *a* | *b*, *c** sú regulárne výrazy popisujúce jazyky L(*a*)L(*b*), $L(a) \cup L(b)$, L(*c*)*. Platia tieto vlastnosti a označenia: εε = ε, (*a*) = *a*, $a^+ = aa^*$, a? = a | ε, [A - Z] = A | ... | Z

Regulárna definícia je pomenovaný regulárny výraz v tvare *názov* $\rightarrow$ *a* (napríklad: digit $\rightarrow$ [0 - 9]). Na rozpoznávanie regulárnych definícii môžeme použiť Thomsonovou metódu konštrukcie NKA z regulárneho výrazu, priamu konštrukciou DKA, prípadne tzv. *pattern matching* (algoritmus *Aho-Corasick*).

Lexémou nazývame reťazec znakov zo vstupu, ktorý zodpovedá patternu pre nejaký token. Lexéma predstavuje inštanciu tokenu.

Úlohy lexikálnej analýzy:

- čítanie vstupu a transformácia na postupnosť tokenov
- zapisovanie lexém do tabuľky symbolov
- kompresia / odstránenie *bieleho priestoru* (medzery, tabulátory, ...)
- rátanie riadkov pre parser
- kopírovanie vstupu s vyznačenými chybami

Lexikálna analýza je z hľadiska teórie zbytočná, avšak v praxi prináša niekoľko výhod:

- jednoduchosť - prehľadnejšia gramatika v parseri
- efektívnosť – pomocou tzv. dvojbuffrovej schémy
- portabilita – príznačnosť abecedy sa rieši na úrovni parsera

Implementácia skenera:

- pomocou generátora lexikálnych analyzátorov
- vo vyššom programovacom jazyku
- v assembleri (efektívne, zložité)

Na rozpoznanie jednotlivých lexém je zväčša potrebné vidieť viac znakov dopredu, preto sa používa vyrovnávacia pamäť – buffer, ktorý sa napĺňa a spracúva až pokým nenastane koniec súboru. Technika *dvojbuffrovej schémy* spočíva v tom, že jeden buffer čítame a druhý spracovávame paralelne.

Ak postupnosť znakov na vstupe nezodpovedá žiadnemu patternu, nastáva chybový stav. Možnosti zotavenia sa z tohto stavu sú nasledovné:

1. ignorovanie nevhodných znakov
2. vloženie chýbajúceho znaku
3. substitúcia nesprávneho znaku
4. transpozícia susedných znakov

3.3 Syntaktická analýza

Na popísanie syntaxe daného programovacieho jazyka sa používajú gramatiky Chomského hierarchie, najčastejšie bezkontextové gramatiky. Cieľom *syntaktického analyzátora* / *parsera* je zistiť, či slovo (program) patrí do jazyka generovaného takouto gramatikou a ak áno, tak určiť príslušný strom odvodenia - *syntaktický* / *parsovací strom*.

Metódy parsovania:

1. algoritmy *Cocke-Younger-Kasami*, *Earley* – dokážu spracovať ľubovolnú gramatiku, no sú neefektívne a v praxi sa nepoužívajú
2. metóda *zhora-nadol*, *rekurzívny zostup*
3. metóda *zdola-nahor*, *shift-reduce parsing*

Posledné dve metódy sú síce efektívne, no dokážu rozpoznať iba jazyky generované špeciálnymi gramatikami (LL resp. LR). Tieto nám však v praxi postačujú, preto sa tieto metódy často používané. Pri oboch sa vstupný program spracováva zľava doprava.

V syntaktickej analýze sa vyskytuje väčšia časť detekcie a zotavovania sa z chýb vstupného programu (*error handling*). Poznáme tieto stratégie:

- *panický mód* – v prípade že sa vyskytne chyba, parser ignoruje tokeny pokým nenarazí na tzv. *synchronizačný token*, po ktorom sa pokúsi zotaviť (napr. znak bodkočiarka v jazyku C / C++)
- *oprava na úrovni fráz* – parser sa pokúsi vykonať lokálnu korekciu aby mohol pokračovať (napríklad zámena čiarky za bodkočiarku)
- *chybové pravidlá* – ak v danom kroku odvodenia očakávame častý výskyt chýb, pridáme do gramatiky pravidlá, ktoré ich „zachytia"
- *globálna korekcia* – v prípade chyby sa snažíme nájsť taký postup opravy chybného bloku, ktorý vykoná čo najmenej zmien - v praxi nepoužiteľné kvôli výpočtovej zložitosti

3.3.1 Metóda zhora-nadol (top-down parsing)

Cieľom tejto metódy je nájsť ľavé krajné odvodenie vstupu konštrukciou stromu odvodenia začínajúc v koreni a vytvárajúc uzly v preorderi. Požadujeme, aby bola príslušná gramatika bez ľavej rekurzie, inak by mohlo dôjsť k zacykleniu.
Algoritmus: rekurzívny zostup

3.3.2 Metóda zdola-nahor (bottom-up parsing)

Úlohou je skonštruovať syntaktický strom začínajúc v listoch a postupujúc smerom ku koreňu. Týmto spôsobom získame pravé krajné odvodenie vstupu zapísane odzadu.

Shift-Reduce (SR) parsing je všeobecná metóda parsovania zdola-nahor. V podstate sa jedná o proces, pri ktorom redukujeme vstupný reťazec na počiatočný neterminál gramatiky spätným nahradzovaním pravidiel. Parser, ktorý používa túto metódu, označujeme *SR parser*.

Redukcia je akcia, v ktorej sa podslovo, ktoré sa nachádza na pravej strane nejakého pravidla nahradí ľavou časťou toho istého pravidla.

Handle (*rukoväť*) je podreťazec, ktorý zodpovedá pravej strane nejakého pravidla a ktorého redukcia predstavuje jeden krok pravého krajného odvodenia, formálne, handle pravovetnej formy γ je pravidlo $A \rightarrow \beta$ a pozícia β v γ, kde sa β môže nahradiť neterminálnom A.

Pri implementácii parsera sa využíva zásobník a vstupný buffer. Handle sa eventuálne objaví na vrchole zásobníka, nikdy nie vo vnútri.

Viable (životaschopný; uskutočniteľný) prefix je prefix pravej vetnej formy, ktorý nepresahuje pravý koniec najpravejšej handle vetnej formy. Je to taký prefix, ktorý sa môže objaviť v zásobníku *SR parsera*.

Operácie SR parsera:

- *SHIFT* (posun) – nasledujúci symbol vstupu sa posunie na vrchol zásobníka

- *REDUCE* (redukcia) – ak sa na vrchole zásobníka vyskytuje nejaká handle, redukuje sa podľa príslušného pravidla gramatiky

- *ACCEPT* – úspešné dokončenie parsovania
- *ERROR* – volá sa rutina pre ošetrenie chyby

Ak je gramatika popisujúca syntax programovacieho jazyka *nejednoznačná*[7], tak sa počas parsovania vyskytne jeden z nasledovných konfliktov:

- Shift / Reduce – parser sa nevie deterministicky rozhodnúť, či má v danom kroku vykonať *shift* alebo *reduce*

- Reduce / Reduce – parser nedokáže jednoznačne určiť, ktoré pravidlo sa má použiť pre redukciu

Deterministický SR parser sa musí vedieť na základe obsahu zásobníka a symbolov na vstupe jednoznačne rozhodnúť akú akciu má vykonať.

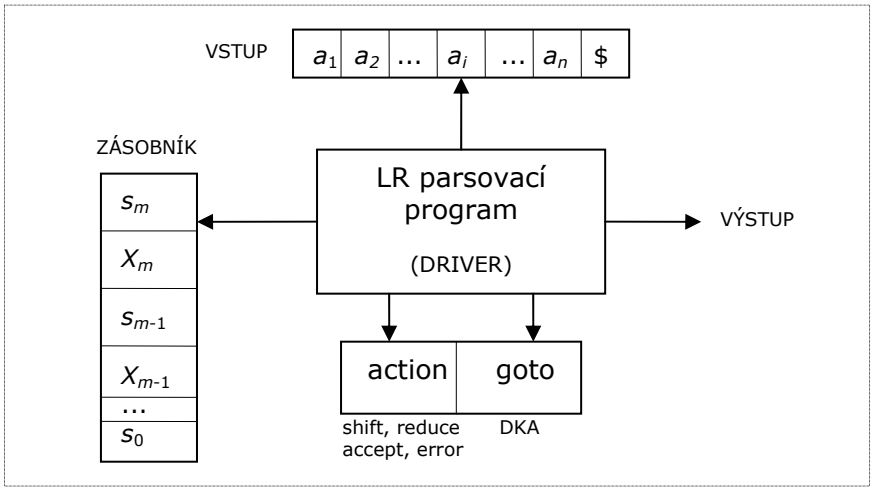
7 ak existuje program, pre ktorý sa dajú zostrojiť aspoň dva rôzne syntaktické stromy

3.3.3 LR parsing

LR parsing je efektívna technika založená na shift-reduce parsingu. Môže byť použitá pre veľkú podmnožinu bezkontextových gramatík – tzv. *LR(k) gramatík*:

- *Left-to-right* – čítanie vstupu zľava doprava
- *Rightmost derivation* – pravé krajné odvodenie
- *k: look ahead – počet videných* symbolov

LR parsery dokážu rozpoznať všetky programové konštrukcie, pre ktoré existuje bezkontextová gramatika. LR parsing patrí medzi najsilnejšie metódy SR parsovania, ktoré nepoužívajú *backtracking*. Medzi ďalšie výhody patrí fakt, že chyby sa detekujú najskôr ako je to možné. Naproti tomu sú LR parsery náročnejšie na implementáciu.



Obrázok 3.2: Model LR parsera

a_i – terminály, s_i – stavy, X_i – vetné formy

Nech X_1, ... X_m sú vetné formy; a_1, ..., a_n terminály (tokeny) a s_0, ..., s_m stavy LR parsera. Potom jeho *konfiguráciu* predstavuje dvojica ($s_0X_1s_1X_2s_2...X_ms_m$, $a_i...a_n$\$), ktorá reprezentuje pravovetnú formu $X_1...X_ma_1...a_n$

Operácie LR parsera:

- action [s_m, a_i] = shift s ($s_0X_1s_1...X_ms_ma_is$, $a_{i+1}...a_n$\$)
- action [s_m, a_i] = reduce $A \rightarrow \beta$ ($s_0X_1s_1...X_{m-1}s_{m-1}A_s$, $a_i...a_n$\$)
  kde s = goto[s_m, a_i], r = $|\beta|$
- action [s_m, a_i] = accept
- action [s_m, a_i] = error

Algoritmus *LR Parsing (w, M)*:

```
push (s0); ip^ := w$;
loop
    x := top; a := ip^;
    if (action[x, a] = shift x' ) then begin
        push (a); push (x' ); ip++; end
    else if (action[x, a] = reduce A → β) then begin
        for i:=1 to 2*| β | do pop;
        x' := top; push (a);
        push (goto[x', A]); output (A → β); end
    else if (action[x, a] = accept) then break
    else error;
pool;
```

Vlastnosti LR gramatík:

- vieme pre ne zostrojiť LR parsovaciu tabuľku
- ak je gramatika nejednoznačná, tak nie je LR
- existujú bezkontextové gramatiky, ktoré nie sú LR (vieme sa im vyhnúť pri písaní syntaxe)

Poznáme viacero typov LR parserov:

1. *SLR*: jednoduchý LR parser – výpočtovo najslabší (nevyužíva všetky možné informácie), no najjednoduchší na implementáciu
2. *LR*: kanonický LR parser – výpočtovo najsilnejší, avšak najzložitejší na realizáciu a priestorovo neefektívny
3. *LALR*: LR parser s výhľadom – kompromis medzi predchádzajúcimi

3.4 Syntaxou riadený preklad

Táto sekcia popisuje spôsob prekladu jazykov využívajúc bezkontexové gramatiky, navrhnuté v predchádzajúcich fázach.

Symbolom gramatiky priradíme *atribúty* (premenné ľubovolného typu), ktoré obsahujú informácie o konštrukciách rozpoznávaných syntaxou. Hodnoty atribútov sú určované *sémantickými rutinami* pridruženými k pravidlám gramatiky. Tieto rutiny môžu generovať kód, vytvárať správy o chybách, kontrolovať typy, pracovať s tabuľkou symbolov a vykonávať ďalšie akcie...

Existujú dve notácie pre asociáciu sémantických rutín s pravidlami gramatiky:

1. Syntaxou riadené definície
2. Prekladové schémy

Syntaxou riadené definície špecifikujú preklad na vyššej úrovni. Abstrahujú od implementačných detailov a nespomínajú v akom poradí má výpočet prebiehať.

Prekladové / translačné schémy určujú poradie volania sémantických rutín, ktoré nemusí závisieť od metódy syntaktickej analýzy. Pre danú metódu sa dá zostrojiť viacero prekladových schém.

3.4.1 Syntaxou riadené definície

Predstavujú zovšeobecnenie bezkontextovej gramatiky, v ktorom má každý symbol asociovanú množinu atribútov.

Atribúty môžu byť buď *syntetizované* alebo *dedičné*. Hodnota syntetizovaných atribútov v uzle parsovacieho stromu sa vypočíta z hodnôt atribútov v potomkoch tohto uzla. Hodnota dedičných atribútov sa vypočíta z hodnôt atribútov v rodičovi a súrodencoch.

Anotovaný parsovací strom má definované hodnoty atribútov v každom uzle. *Anotácia / zdobenie* parsovacieho stromu je proces vyhodnocovania hodnôt atribútov.

Graf závislostí D_p znázorňuje vzájomné závislosti (hrany grafu) medzi dedičnými a syntetizovanými atribútmi (uzly grafu). Syntaxou riadené definície sú *cyklické*, ak graf závislostí pre nejaký parsovací strom generovaný gramatikou má cyklus. *Topologické utriedenie* grafu závislostí: Ak je acyklický, tak existuje vrchol bez vstupnej hrany. Tento označíme ako prvý, odstránime ho a postup opakujeme.

3.4.2 Prekladové schémy

Prekladová schéma je množina pravidiel, ktoré určujú graf závislostí medzi atribútmi a poradie ich vyhodnotenia. Typy prekladových schém:

- lokálne – v rámci pravidla
- globálne – spoločné pre všetky pravidlá

Graf závislostí určuje poradie volania sémantických rutín, ktoré okrem iného počítajú hodnoty atribútov.

Prekladové schémy môžu mať syntetické i dedičné atribúty. Musíme zabezpečiť, aby bola hodnota atribútu k dispozícii vždy, keď ju sémantická rutina používa. Požiadavky na dobre definovanú prekladovú schému:

1. dedičný atribút symbolu na pravej strane pravidla musí byť vypočítaný rutinou pred ním
2. sémantická rutina nesmie používať syntetizovaný atribút symbolu napravo od rutiny

- syntetizovaný atribút neterminálu naľavo môže byť vypočítaný až po vypočítaní všetkých atribútov, na ktoré sa odkazuje

3.5 Nástroje pre tvorbu kompilátorov

Cieľom nástrojov pre tvorbu kompilátorov je urýchliť a zjednodušiť proces ich prípravy a implementácie, využívajúc postupy a techniky popísané v teórii.
Tieto nástroje môžu byť rozdelené do niekoľkých skupín:

1. nástroje pre tvorbu lexikálnych analyzátorov / skenerov
2. nástroje pre tvorbu syntaktických analyzátorov / parserov
3. nástroje pre tvorbu generátorov kódu

V ďalšom sa zameriame sa prvé dve skupiny, do ktorých patria i nástroje použité pri implementácii UNITY Interpretera.

3.5.1 Nástroje pre tvorbu skenerov

Ich vstupom býva zvyčajne regulárna gramatika a výstupom lexikálny analyzátor. Najznámejším z nich je program *Lex* od autorov Eric Schmidt a Mike Lesk, ktorý sa štandardne vyskytuje na Unixových systémoch. Na vstupe číta špecifikáciu lexikálneho analyzátora (najčastejšie v textovom súbore s koncovkou .l alebo .lex) a na výstupe generuje zdrojový kód, ktorý reprezentuje vstupný analyzátor. Existujú viaceré varianty tohto programu, ktoré sa líšia väčšinou v podpore cieľového jazyka.

Program	**Cieľový jazyk**
Lex	C
Flex	C
JLex	Java
PLY	Python
TP / Delphi Lex	Pascal

Tabuľka 3.1: Prehľad variánt programu Lex

3.5.2 Nástroje pre tvorbu parserov

Vstupom je obvykle bezkontextová gramatika, na základe ktorej vygenerujú syntaktický analyzátor vo forme zdrojového kódu cieľového jazyka.
Medzi najznámejší generátor parserov patrí program *Yacc* (*Yet Another Compiler-Compiler*), ktorého autorom je Stephen C. Johnson z firmy AT&T. Yacc je, podobne ako Lex, štandardnou súčasťou systému Unix. Na základe neho boli vyvíjané ďalšie generátory parserov, adaptované na rôzne cieľové jazyky. Yacc používa LALR parsovaciu techniku a prekladové schémy definované v kapitole 3.4.2.

Program	**Cieľový jazyk**
Yacc	C
Bison	C
Jacc	Java
PLY	Python
TP / Delphi Yacc	Pascal

Tabuľka 3.2: Prehľad variánt programu Yacc

Generátory parserov bývajú často dostupné spolu s generátormi skenerov, čím vytvárajú sadu nástrojov, postačujúcu pre tvorbu kompilátorov, interpreterov alebo prekladačov. Medzi typické dvojice patria Lex-Yacc a Flex-Bison.

Kapitola 4

UNITY Interpreter

Aplikáciu, ktorá bude interpretovať UNITY programy som sa rozhodol implementovať v objektovom Pascale. Výhodou tohto riešenia je, že Pascal je známy a rozšírený jazyk, ktorý je jednoduchý na pochopenie a z ktorého syntax UNITY z časti vychádza (deklarácie premenných). Ďalšou motiváciou bol fakt, že zatiaľ neexistuje, alebo nie je verejne dostupný kompilátor / interpreter UNITY v tomto jazyku.
Na tvorbu aplikácie som využil generátory skenerov a parserov. Z variánt uvedených v kapitolách 3.5.1 a 3.5.2 som podľa cieľového jazyka zvolil sadu nástrojov *Delphi Yacc & Lex*, ktoré sú voľne dostupné na domovskej stránke[8] pod licenciou GNU. Napriek tomu, že prostredie Delphi je známe najmä svojím grafickým rozhraním, UNITY Interpreter je konzolová aplikácia, konfigurovateľná cez parametre. Hlavným parametrom je názov vstupného súboru, obsahujúceho zápis programu v UNITY. Tento by mal byť textový súbor (s koncovkou .unity) v kódovaní ASCII alebo ANSI.

Paralelizmus je v UNITY Interpreteri simulovaný sekvenčným priradením (najprv prebehne vyhodnotenie všetkých pravých strán a indexov) a nedeterminizmus je nahradený daným poradím vykonávania príkazov (*pravidlo nestrannosti*[9] ostáva zachované). Počas výpočtu sa raz za čas skontroluje stav premenných - ak sa zistí že nastal pevný bod, vypíše sa obsah premenných a program bude ukončený.

8 viď [DYL]

9 „každý príkaz sa vykoná nekonečne veľa krát"

4.1 Skener[10]

Štruktúra vstupného súboru pre Delphi Lex je nasledovná:

Obrázok 4.1: Štruktúra vstupného súboru pre Delphi Lex

V sekcii definícií sa uvádzajú:

- Delphi deklarácie uzavreté medzi reťazcami **%{** a **%}**, ktoré budú vložené do výsledného súboru na globálnu úroveň

- nerekurzívne regulárne definície v tvare:

názov	*substitúcia*
DIGIT	[0-9]
LETTER	[a-zA-Z_]
INTEGER	{DIGIT}+
IDENTIFIER	{LETTER}({LETTER}|{DIGIT})*
FLOAT	{DIGIT}+\.{DIGIT}+

- definície *stavových identifikátorov*[11] v tvare:

%start	*názov*
%start	DEF
%start	COMMENT1
%start	COMMENT2

10 viď elektronickú prílohu – súbor ***UnityLex.l*** v zložke ***src***

11 umožňujú rozhodovanie medzi pravidlami

Sekcia pravidiel obsahuje samotnú špecifikáciu parsera v tvare:

<stav>regularny_vyraz prikaz

Kde *stav* je stavový identifikátor definovaný v predchádzajúcej sekcii. Skener sa v danom kroku výpočtu rozhodne pre príkaz, pre ktorý je splnený zodpovedajúci regulárny výraz a nachádza sa v uvedenom stave.

Skener UNITY Interpretera rozpoznáva:

1. kľúčové slová: **Program**, **Declare**, **Always**, **Initially, Assign, End**, **Array**, **Of**, **Integer**, **Float**, **Boolean**, **If**, **True**, **False** (zodpovedajúce tokeny začínajú prefixom *LAB_ alebo TOK_*)
2. logické operátory: **And**, **Or**, **Not** (tokeny majú prefix *LOG_*)
3. identifikátory (token *T_ID*) a konštanty (*T_INTEGER* resp. *T_FLOAT*)
4. jednoznakové tokeny + použitie:
   - **:[];** v deklaračnej sekcii
   - **+-*/%** v aritmetických výrazoch
   - **!=<>()** pri relačných operáciách
   - **#** oddelenie príkazov – nedeterminizmus (*TOK_CMD*)
5. dvojznakové reťazce + význam:
   - "**||**" oddelenie priradení – paralelizmus (*TOK_PAR*)
   - "**..**" v deklaračnej sekcii – rozsah poľa (*TOK_RANGE*)
   - "**<<**" začiatok kvatifikovaného príkazu alebo výrazu (*TOK_QB*)
   - "**>>**" koniec kvatifikovaného príkazu alebo výrazu (*TOK_QE*)
6. komentáre a biely priestor – ignorujú sa

V kľúčových slovách a logických operátoroch nerozlišujeme malé a veľké písmená, t.j. sú *case-insensitive*. Naproti tomu, identifikátory musia byť *case-sensitive*.

4.2 Parser[12]

Štruktúra vstupného súboru pre Delphi Yacc je rovnaká ako v prípade skenera, avšak líši sa obsahom jednotlivých sekcií.

V sekcii definícií sa uvádza:

- zoznam tokenov, ktoré rozpoznáva parser v tvare:
  %token *symbol*

- typy atribútov neterminálnych symbolov v tvare:
  %type *<typ> symbol*

- precedencia operátorov:
  %left *symbol* +-*/%
  %right *symbol*
  %nonassoc *symbol*

- počiatočný neterminál:
  %start *symbol*

V druhej časti sa nachádza špecifikácia parsera pomocou pravidiel gramatiky a priradených akcií - sémantických rutín.

Posledná sekcia slúži pre definovanie pomocných funckií. Je tu uvedený i hlavný blok programu, ktorý sa zavolá po spustení výslednej aplikácie. V UNITY Interpreteri je jeho úlohou načítať parametre programu a iniciovať proces parsovania. Ak sa počas neho nevyskytnú žiadne chyby, spustí „hlavná procedúra" výpočet UNITY programu.

12 viď elektronickú prílohu – súbor ***Unity.y*** v zložke ***src***

4.2.1 Pravidlá gramatiky

Neterminály sú písané malými písmenami, terminály veľkými. Počiatočný neterminál je *beginning*.

```
beginning           : LAB_PROGRAM prog LAB_END ;
prog                : T_ID section_declare section_always section_initially section_assign
                    ;
section_declare     : /* empty */
                    | LAB_DECLARE declare_list
                    ;
declare_list        : declare ';'
                    | declare ';' declare_list
                    ;
declare             : dec_var_list ':' dec_type
                    | dec_var_list '=' dec_const
                    ;
dec_var_list        : dec_var
                    | dec_var ',' dec_var_list
                    ;
dec_var             : T_ID
                    ;
dec_type            : dec_simple_type
                    | TOK_ARRAY '[' dec_range_list ']' TOK_OF dec_simple_type
                    ;
dec_simple_type     : TOK_INT | TOK_FLOAT | TOK_BOOL
                    ;
dec_range_list      : dec_range
                    | dec_range ',' dec_range_list
                    ;
dec_range           : dec_range_expr TOK_RANGE dec_range_expr
                    ;
dec_range_expr      : expr
                    ;
dec_const           : T_INTEGER | T_FLOAT | TOK_TRUE | TOK_FALSE
                    ;
section_always      : /* empty */
                    ;
```

```
section_initially       : /* empty */
                        | LAB_INITIALLY in_statement_list
                        ;
in_statement_list       : in_statement
                        | in_statement TOK_CMD in_statement_list
                        ;
in_statement            : in_par_stat_list
                        | TOK_QB TOK_CMD quant_var_list ':' expr ':' ':' in_statement_list TOK_QE
                        ;
in_par_stat_list        : in_par_stat
                        | in_par_stat TOK_PAR in_par_stat_list
                        ;
in_par_stat             : in_enum_assign
                        | TOK_QB tok_par quant_var_list ':' expr ':' ':' in_par_stat_list TOK_QE
                        ;
in_enum_assign          : var_array_list '=' assign_list
                        ;



section_assign          : /* empty */
                        | LAB_ASSIGN as_statement_list
                        ;
as_statement_list       : as_statement
                        | as_statement TOK_CMD as_statement_list
                        ;
as_statement            : as_par_stat_list
                        | TOK_QB TOK_CMD quant_var_list ':' expr ':' ':' as_statement_list TOK_QE
                        ;
as_par_stat_list        : as_par_stat
                        | as_par_stat TOK_PAR as_par_stat_list
                        ;
as_par_stat             : as_enum_assign
                        | TOK_QB tok_par quant_var_list ':' expr ':' ':' as_par_stat_list TOK_QE
                        ;

as_enum_assign          : var_array_list ':' '=' assign_list
                        ;
```

```
var_array_list      : var_array
                    | var_array ',' var_array_list
                    ;
var_array           : T_ID
                    | T_ID '[' expr_list ']'
                    ;
quant_var_list      : quant_var
                    | quant_var ',' quant_var_list
                    ;
quant_var           : T_ID ;

assign_list         : expr_list
                    | cond_expr_list
                    ;
cond_expr_list      : expr_list TOK_IF expr
                    | expr_list TOK_IF expr '~' cond_expr_list
                    ;
expr_list           : expr
                    | expr ',' expr_list
                    ;

expr                : expr_and
                    | expr_and LOG_OR expr
                    ;
expr_and            : expr_not
                    | expr_not LOG_AND expr_and
                    ;
expr_not            : expr_eq
                    | LOG_NOT expr_not
                    ;
expr_eq             : expr_rel
                    | expr_rel '=' expr_eq
                    | expr_rel '!' '=' expr_eq
                    ;
expr_rel            : expr_add
                    | expr_add '<' rel_eq expr_rel
                    | expr_add '>' rel_eq expr_rel
                    ;
rel_eq              : /* empty */
                    | '='
                    ;
```

```
expr_add        : expr_mult
                | expr_mult '+' expr_add
                | expr_mult '-' expr_add
                ;
expr_mult       : expr_fact
                | expr_fact '*' expr_mult
                | expr_fact '/' expr_mult
                | expr_fact TOK_MOD expr_mult
                ;
expr_ fact      : '(' expr ')'
                | '-' expr_fact
                | var_array
                | T_ID '(' ')'
                | T_ID '(' expr_list ')'
                | TOK_QB quant_op quant_var_list ':' expr ':' ':' expr TOK_QE
                | TOK_QB T_ID quant_var_list ':' expr ':' ':' expr TOK_QE
                | T_INTEGER
                | T_FLOAT
                | TOK_TRUE
                | TOK_FALSE
                ;

quant_op        : '+'
                |  '*'
                |  LOG_AND
                |  LOG_OR
                ;
```

Takto definovaná gramatika G je jednoznačná - pre každé slovo z jazyka L(G) existuje najviac jeden strom odvodenia. Tým pádom vo výslednom parseri nedochádza k Shift-Reduce ani Reduce-Reduce konfliktom.

4.3 Sémantika

LALR parser, ktorý Delphi Yacc vygeneruje, používa techniku syntaxou riadeného prekladu, notáciou sú prekladové schémy. K pravidlám gramatiky sa priradia sémantické rutiny, pričom je určené i poradie, v akom sa vykonajú.

Príklad. Booleovské výrazy v UNITY Interpreteri:

```
. . .
expr_and      : expr_not                    { $$ := $1; }
              | expr_not LOG_AND expr_and{ $$ := TBExprOperation.Create ($1, logAnd, $3); }
              ;

expr_not      : expr_eq                     { $$ := $1; }
              | LOG_NOT expr_not            { $$ := TBExprOperation.Create (logNot, $2); }
              ;

expr_eq       : expr_rel                    { $$ := $1; }
              | expr_rel '=' expr_eq        { $$ := TBExprCompare.Create ($1, relEQ, $3); }
              | expr_rel '!' '=' expr_eq    { $$ := TBExprCompare.Create ($1, relNE, $4); }
              ;
. . .
```

expr_and, *expr_not* a *expr_eq* sú inštanciami tried, ktoré majú spoločnú základnú triedu *TExpression*

Kompletný zoznam sémantických akcií sa nachádza v elektronickej prílohe v súbore *Unity.y*, pod adresárom *src* (*source* – zdrojový kód).

4.4 Interpretácia

Beh aplikácie sa dá rozdeliť na dve fázy:

1. parsovanie vstupného programu a ukladanie do dátových štruktúr
2. interpretácia – vykonávanie príkazov programu v pamäti

Počas prvej fázy sa zisťujú syntaktické a sémantické chyby. Ak niektorá z nich nastane, označí sa na výstupe miesto jej výskytu a parser sa snaží zotaviť a pokračovať. K tomu som v gramatike použil niekoľko chybových pravidiel najmä tam, kde očakávam častý výskyt syntaktických chýb.

Vždy, keď dôjde k redukcii pravidla popisujúceho istú sekciu programu, v sémantických rutinách sa načítané dáta v atribútoch uložia do dátových štruktúr a priradia sa atribútu na ľavej strane. Na najvyššej úrovni sa celý program uloží do inštancie triedy *TUnityProgram,* definovanej v module *UnityLib.pas.*

Druhá fáza sa spustí len v prípade, že sa v prvej fáze nevyskytnú žiadne chyby. Z hlavného bloku programu sa spustí metóda *TUnityProgram.Execute*, ktorá postupne volá rovnomenné metódy v jednotlivých sekciách (*TSectionInitially.Execute*, ...).

Prehľad modulov a tried:

- *UnityCom.pas* – obsahuje základné konštrukcie UNITY: triedy *TUType, TUVariable, TExpression*, ...

- *UnityLib.pas* – triedy reprezentujúce jednotlivé sekcie a ich časti (príkazy, priradenia) + hlavná trieda *TUnityProgram*

Kapitola 5

Záver

V diplomovej práci som popísal základné prvky programovej notácie systému UNITY a úvod do teórie kompilátorov. V implementačnej časti práce som naprogramoval UNITY interpreter v objektovom Pascale, využijúc nástroje pre tvorbu lexikálnych a syntaktických analyzátorov.
V ďalších verziách programu by som si vedel predstaviť napríklad:

- grafické rozhranie so zvýrazňovaním syntaxe
- podporu ďalších základných i zložených typov, prácu s reťazcami
- viac zabudovaných funkcií, možnosť zadefinovať vlastné funkcie

No myslím, že i súčasná verzia dostatočne postačuje na dobré zoznámenie sa so systémom UNITY.

Literatúra

[CM88] K. Mani Chandy, Jayadev Misra. *Parallel Program Design: A Foundation*. Prentice Hall, 1988.

[ASU86] A. V. Aho, R. Sethi, J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 1986.

[HU78] John E. Hopcroft Jeffrey D. Ullman. *Formálne jazyky a automaty*. Alfa SNTL, 1978.

[WiUni] Prispievatelia do online encyklopédie Wikipedia. *UNITY programming language*. Prístupné na internete: http://en.wikipedia.org/wiki/UNITY_programming_language

[WiInt] Prispievatelia do online encyklopédie Wikipedia. *Interpreter (computing)*. Prístupné na internete: http://en.wikipedia.org/wiki/Interpreter_%28computing%29

[DYL] Michiel Rook. *Delphi Yacc & Lex. A parser generator toolset for Delphi and Kylix. Version 1.4 User Manual*. December 2005. Prístupné v rámci nástrojov Delphi Yacc & Lex v1.4 na internete: http://www.grendelproject.nl/dyacclex/

Dodatok

Elektronická príloha

Na priloženom CD sa nachádza ZIP archív *unity.zip* s nasledovnou štruktúrou:

- bin - spustiteľný kód s testovacími programami
- doc - dokumenty
- src - zdrojový kód bez Delphi Yacc & Lex knižníc
- build - zdrojový kód pripravený na kompiláciu – spúšťa sa pomocou dávkových súborov *step1.cmd* a *step2.cmd*

Elektronická verzia tejto práce sa nachádza v súbore *DiplomovaPraca.pdf*