Interpreter UNITY

DIPLOMOVÁ PRÁCA

Autor práce: Michal Šuster

UNIVERZITA KOMENSKÉHO V BRATISLAVE
FAKULTA MATEMATIKY FYZIKY A INFORMATIKY
KATEDRA INFORMATIKY

Vedúci záverečnej práce: RNDr. Damas Gruska, PhD.

BRATISLAVA 2006

Ďakujem svojmu diplomovému vedúcemu RNDr. Damasovi Gruskovi, PhD. za cenné rady a pripomienky pri vypracovávaní diplomovej práce. Ďakujem aj Mgr. Ondrejovi Krškovi za užitočné rady pri implementácií.

Čestne prehlasujem, že diplomovú prácu som vypracoval samostatne s použitím literatúry a elektronických dokumentov uvedených v zozname použitej literatúry.

. . . . . .

Michal Šuster

Abstrakt

ŠUSTER, Michal. Interpreter UNITY. [Diplomová práca]. Univerzita Komenského. Fakulta matematiky, fyziky a informatiky. Katedra informatiky.
Školiteľ: RNDr. Damas Gruska, PhD. Obhajoba: Bratislava, 2006. 82 strán.
V diplomovej práci sa zaoberám vytvorením interpretera pre jazyka UNITY. Tento jazyk je súčasťou teórie neohraničených, nedeterministických, iteratívnych transformácií programového stavu (Unbounded Nondeterministic Iterative Transformations - UNITY) navrhnutej za účelom efektívneho navrhovania riešení paralelných problémov.
Práca obsahuje stručný úvod do teórie UNITY a popis konštruktov jazyka UNITY s ich sémantickým významom. Popisuje teórie lexikálnej a syntaktickej analýzy, ktoré stoja za automatickými nástrojmi na generovanie analyzátorov, použitých pri tvorbe interpretera. Oboznamuje s platformou .NET, pod ktorou bol interpreter implementovaný a takisto popisuje jeho samotnú implementáciu. Súčasťou práce sú aj príklady demonštrujúce prístup k riešeniu paralelných problémov v jazyku UNITY.

Kľúčové slová: UNITY, Unbounded Nondeterministic Iterative Transformations, interpreter, automatické nástroje na generovanie analyzátorov, C# Lex, Jay, LR parser

Obsah

Zoznam obrázkov

Zoznam tabuliek

ÚVOD

Unbounded Nondeterministic Iterative Transformations of the program state (UNITY) je výpočtový model a dokazovací systém slúžiaci ako pomôcka pri navrhovaní paralelných programov. V minulosti sa paralelne programy vyvíjali tak že už v úvodných fázach dizajnu využívali črty hardwaru na ktorom mali bežať. Skúsenosti však ukázali že optimalizáciu na konkrétny hardware je dobré ponechať až na posledné fázu vývoja programu. Prvotnou starosťou by malo byť nájdenie dizajnu riešenia problému, ktorý ma program riešiť a až následne toto riešenie implementovať v konkrétnom programovacom jazyku na konkrétnej architektúre. Takto navrhnuté programy sa dajú ľahko implementovať aj po zmene architektúry a sú aj ľahšie pochopiteľné. UNITY bolo vyvinuté práve za účelom navrhovania riešení paralelných problémov.

Mojou snahou, v tejto práci, je vytvoriť interpreter UNITY programu. Interpreter môže nájsť využitie ako pomôcka na rýchlejšie sa zblíženie s týmto jazykom.

Kapitola 1

UNITY

UNITY bol navrhnutý a popísaný v knihe Parallel Program Design – A Foundation ktorej autormi sú K. Mani Chandy a Jayadev Misra z University of Texas. Teória UNITY sa snaží oddeliť (programátorské) rozmýšľanie ohľadne problému – programu od vlastnej implementácie na konkrétnu architektúru. Odlúčiť starosť ohľadne toho čo sa má vykonať od záležitosti kde, kedy a ako sa to má vykonať. Detaily ohľadne implementácie sú zvažované až pri mapovaní navrhnutých programov na konkrétnu architektúru. Tým vlastne ponúka možnosť navrhovať dizajn programu – abstraktné riešenie, nezávislé na konkrétnej paralelnej architektúre, čím sa rieši problém ktorý nastáva ak je program pevne naviazaný na konkrétnu architektúru ktorá časom zastarala. UNITY tak ponúka zjednocujúcu teóriu na vývoj paralelných programov pre rôzne typy architektúr a aplikácií. Výpočtovým modelom sú neohraničené nedeterministické opakované zmeny programového stavu, reprezentované pomocou zložených priraďovacích príkazov (multiple-assignment statement).

Základnými črtami teórie sú: nedeterminizmus, absencia toku riadenia (control-flow), synchrónnosť a asynchrónnosť, stavy a priradenia (prechodový systém), odkazovací systém správnosti programov je oddelený od kódu programu, oddelenie správnosti (závisí len od programu) od zložitosti (závisí od programu jeho implementácie a konkrétnej architektúry na ktorú je implementovaný).

Nedeterminizmus

Vyvíjať program pre rôzne druhy architektúr postupným zjemňovaním základnej koncepcie sa javí ako vhodná stratégia. V prvotnej fáze dizajnu program stačí vykonávanie programu špecifikovať iba málo, čiže to znamená že náš program môže byť nedeterministický. Rôzne spustenia programu môžu vykonávať príkazy v rozdielnom poradí, môžu spotrebovať rôzne množstvá zdrojov, prípadne vyprodukovať rozdielne výsledky. Nedeterminizmus umožňuje vytvárať

jednoduché programy, ktorých jednoduchosť je dosiahnutá vďaka tomu že sa vyhneme zbytočnému determinizmu. Pre architektúry ktoré nedeterminizmus neobsahujú je možné ho na úkor zložitosti odstrániť. Navyše sú niektoré systémy zo svojej povahy nedeterministické (napríklad operačné systémy a delay-insensitive circuits).

Absencia control-flow

Prvé programovacie jazyky boli založene na sekvenčnom toku riadenia, neskôr štruktúrované programovanie uviedlo možnosť podprogramov (subroutine), čím sa problém dekomponoval na sekvencovanie úloh. Ďalším rozšírením je definovanie programu ako množiny procesov. Procesy sú definované ako sekvenčné a tým sa teda ponúka možnosť viacerých súbežných - paralelných tokov riadenia. Rozdielne paralelne modely (kooperujúce sekvenčné procesy, paralelne komunikujúce sekvenčné procesy, alternujúce stroje, paralelne RAM, booleovské obvody, atď.) a z nich vychádzajúce architektúry definujú pre programy rôzne formy toku riadenia.

Dizajn programov by sa v prvotných fázach nemal upínať na konkrétnu formu toku riadenia a ponechať riešenie tejto otázky až na záverečnú fázu mapovania programu na cieľovú architektúru. Je oveľa ľahšie pridať obmedzia toku riadenia v programe ktorý má málo obmedzení ako odoberať nadbytočné obmedzenia z programu, ktorý ich má príliš veľa.

Synchrónnosť a asynchrónnosť

Synchrónne a asynchrónne udalosti musia byť v základoch každej zjednocujúcej teórie paralelného programovania. Delenie systémov na synchrónne a asynchrónne je umelé – neprirodzené. Preto aj UNITY obsahuje synchrónnosť a asynchrónnosť ako základné koncepty.

Stavy a priradenia (prechodový systém)

Systém prechodov stavov (state-transition system) sa ako formálny model používa v rôznych oblastiach inžinierstva, prírodných vied a taktiež i v programovaní. Preto sa stal vhodným základom aj pre UNITY. Avšak spracovanie programu ako systém prechodov stavov pozostávajúci z množiny stavov, počiatočného stavu a prechodovej funkcie stavov ponúka slabé základy pre vývoj programov. Preto je systém prechodov stavov v UNITY reprezentovaný počiatočným stavom a transformáciami, ktoré sú popísané pomocou konštruktov programovacích jazykov a to konkrétne pomocou premenných a priradení. Stav programu je teda definovaný aktuálnymi hodnotami premenných, a prechod do iného stavu sa udeje cez sériu priradení.

Zložené priradenie (multiple-assignment) umožňuje simultánne priradenie do viacerých premenných, pričom premenné samé možu byť komplexné štruktúry. V UNITY sú priradenia jedinými príkazmi, príkazy ktoré by definovali tok riadenia chýbajú, čím sa zabezpečuje už spomínaná črta absencie toku riadenia.

UNITY Program

Program pozostáva z deklarácie premenných, nastavení ich počiatočných hodnôt a množiny zložených priradení (multiple-assignment). Vykonávanie programu začína v ľubovoľnom stave programu ktorý splňuje počiatočné podmienky. Program sa vykonáva donekonečna. V každom kroku výpočtu je nedeterministický vybratý ľubovoľný priraďovací príkaz a je vykonaný. Nedeterministický výber je obmedzený nasledovným pravidlom „nestrannosti“ (fariness rule): Každý príkaz je vykonaný nekonečne veľa krát.

Oddelenie programu od implementácie

UNITY program popisuje čo by sa malo vykonať v takom zmysle, že špecifikuje počiatočný stav a transformácie stavu (priradenia). Nešpecifikuje sa kedy sa má priradenie vykonať – jedným obmedzením je spomínané pravidlo „nestrannosti“. UNITY nešpecifikuje ani kde (napríklad: na ktorom procesore multiprocesorového systému) sa má priradenie vykonať, taktiež neurčuje

ktorému procesu priradenie patrí. Ani otázka ako je priradenie vykonávané či ako implementácia programu môže ukončiť z definície nekonečné vykonávanie programu, nie je riešená v samotnom UNITY kóde.
UNITY oddeľuje starosť čo sa ma vykonať na jednej strane a otázky kedy, kde a ako na druhej strane. Čo sa má vykonať je špecifikované v kóde UNITY programu ostatné otázky sú riešené v mapovaní programu na konkrétnu architektúru. Takýmto oddelením získavame jednoduchý program vhodný pre širokú škálu architektúr. Je zrejmé že táto jednoduchosť ide na úkor toho že namapovanie na konkrétnu architektúru je viac zložitejšie a komplexnejšie ako pri tradičných programoch.

Mapovanie UNITY programu na architektúru

Von Neumann-ová architektúra

Mapovanie na von Neumann-ovú architektúru špecifikuje rozpis pre vykonávanie priradení a spôsob akým je vykonávanie programu ukončené. Implementácia viacnásobného priradenia na sekvenčných zariadeniach spočíva vo vyhodnotení pravých strán a ich následnom priradení korešpondujúcim premenným na ľavých stranách. Rozpis je reprezentovaný konečnou sekvenciou priradení v ktorej sa každé priradenie z programu vyskytuje prinajmenšom raz. Počítač opakovane - donekonečna vykonáva tento rozpis. Takéto mapovanie splňuje pravidlo „nestrannosti“, keďže každé priradenie sa v rozpise nachádza aspoň raz a rozpis sa vykonáva donekonečna tak sa každé priradenie vykoná nekonečne veľa krát.
Stav programu sa nazýva pevný bod (fixed point) vtedy a len vtedy ak vykonanie hocijakého priraďovacieho príkazu z programu v tomto stave zanechá stav nezmenený. Predikát nazvaný FP charakterizuje pevný bod programu. FP je konjunkcia rovností ktoré dostaneme nahradením operátoru priradenia operátorom rovnosti v každom priraďovacom príkaze programu. Z toho dôvodu FP platí vtedy a len vtedy keď sa hodnoty ľavej a pravej strany každého priradenia v programe rovnajú. Keď FP raz platí tak ďalší výpočet programu nemení žiadne premenné a teda je jedno či výpočet programu pokračuje ďalej

alebo zastaví. Teda jedným zo spôsobov ako implementovať ukončenie vykonávania programu je zastaviť ho po dosiahnutí pevného bodu. Stabilný predikát programu je predikát ktorý keď raz platí tak už bude platiť stále. Preto je FP stabilný predikát.

Architektúra synchrónneho systému so zdieľanou pamäťou

V synchrónnom systéme so zdieľanou pamäťou je pevný počet identických procesorov, ktoré zdieľajú spoločnú pamäť z ktorej môžu čítať a do ktorej môžu zapisovať všetky procesory. Sú tam aj spoločné hodiny, na každý tik každý procesor vykoná práve jeden krok výpočtu. Implementácia viacnásobného priradenia je pre synchrónne systémy vlastná: každý procesor vypočíta výraz prevej strany priradenia zodpovedajúceho jednej premennej a následne túto hodnotu tejto premennej priradí. Táto architektúra je tiež veľmi vhodná pre vypočítavanie hodnôt kvantifikovaných výrazov definovaných cez asociatívny operátor (suma, minimum, maximum atď.) aplikovaný na sekvenciu dátových položiek.

Architektúra asynchrónneho systému so zdieľanou pamäťou

Tento systém sa od predchádzajúceho líši absenciou spoločných hodín. Ak dva procesory pristupujú k tomu istému pamäťovému miestu naraz, tak ich prístup je vybavený v ľubovoľnom poradí. UNITY program môže byť namapovaný na túto architektúru rozkúskovaním priradení z programu medzi procesory. Navyše je pre každý procesor definovaný rozpis vykonávania ktorý zabezpečuje pravidlo „nestrannosti“ pre jemu zodpovedajúce priradenia. Ak je pre každú časť programu zabezpečené pravidlo „nestrannosti“ tak je zaručene aj pre celý program. Pri mapovaní na túto architektúru treba zabezpečiť aby dva priraďovacie príkazy neboli vykonávané súčasne ak jeden modifikuje premenné ktoré využíva druhý. Asynchrónny multiprocesor môže simulovať aj spoločné hodiny a teda simulovať synchrónny systém.

Kapitola 2

ŠTRUKTÚRA UNITY PROGRAMOV

Na popis štruktúry je použitá Backus-Naur form-a (BNF) ktorá slúži na vyjadrenie bezkontextových gramatík ktorými sa dá formálne popísať štruktúra väčšiny programovacích jazykov, vrátane UNITY. Neterminálne symboly sú písané italikou, terminálne jednoduchým tučným písmom. Syntaktická jednotka uzatvorená v kučeravých zátvorkách „{“ a „}“ môže byť nahradené nula alebo viackrát.

UNITY Program

```
program →  Program          program-name
             declare        declare-section
             always         always-section
             initially      initially-section
             assign         assign-section
           end
```

program-name je ľubovoľný textový reťazec. Kľúčové slovo sekcie môže byť vynechané ak je daná sekcia prázdna.

declare-section menuje premenné požité v programe a ich typy. Syntax je podobná deklarovaniu premenných v PASCALe a preto ju ďalej nebudeme popisovať. Zväčša stačí používanie typov integer a boolean ako základných typov. Polia, množiny a sekvencie týchto základných typov sa tiež využívajú.

always-section je miesto kde sa definujú isté premenné ako funkcie iných premenných. Táto sekcia nie je nevyhnutná pre písanie UNITY programov, ale jej používanie je v niektorých prípadoch veľmi vhodné.

initially-section sa používa na definovanie počiatočných hodnôt niektorých premenných. Premenné ktoré nie sú inicializované majú ľubovoľnú počiatočnú hodnotu.

assign-section obsahuje množinu assignment statements (priraďovacích príkazov)

Vykonávanie programu začne v stave, kde premenné majú svoje počiatočné hodnoty. V každom kroku je vyhodnotený ľubovoľný statement (príkaz). Statements (príkazy) sú vyberané náhodne, ale pri nekonečnom vykonávaní programu sa každý príkaz vykoná nekonečne veľa krát. Stav programu sa vola fixed point (fixný bod) vtedy a len vtedy ak vykonanie ľubovoľného statement-u (príkazu) zanechá stav nezmenený a teda že všetky premenné budú obsahovať rovnakú hodnotu ako pred vykonaním statement-u.

Program nemá žiadne vstupné ani výstupné statement-y (príkazy). Nie sú povolené ani hierarchické štruktúry programov.

Assignment Statement

Priradenie jednej indexovanej či neindexovanej premennej je už dobre známe, keďže sa vyskytuje v každom programovacom jazyku. UNITY pridáva notáciu ktorá rozširuje tento koncept.

UNITY povoľuje aby sa niekoľkým premenným priradili hodnoty naraz – paralelne v jednom viacnásobnom priradení.

Napríklad assignment (priradenie):

```
x,y := 0,1 || z := 2
```

môže byť rozpísané na množinu assignment-components oddelených paralelným separátorom || ako v nasledujúcom príklade:

```
x,y,z := 0,1,2
```

alebo

```
x := 0 || y := 1 || z := 2
```

Premenné a hodnoty ktoré k nim majú byť priradené môžu byť popísané kvantifikáciou miesto ich vypísania.

Napríklad :

```
< || i : 0 ≤ i ≤ N :: A[i] := B[i] >
```

definuje nasledovnú množinu assignment-components

```
A[0] := B[0] || A[1] := B[1] ||. . .|| A[N] := B[N]
```

nasledujúca notácia cez case analysis (prípadovú analýzu) pre definovanie hodnoty premenných je v matematike bežná:

$$x = \begin{cases} -1 \ if \quad y < 0 \\ 0 \ if \quad y = 0 \\ 1 \ if \quad y > 0 \end{cases}$$

UNITY preberá túto notáciu pre assignments (priradenia), oddeľujúc jednotlivé prípady symbolom **~**

```
x :=  -1    if y < 0 ~
       0    if y = 1 ~
       1    if y > 0
```

Formálny popis assignment-statement-u :

```
assignment-statement →     assignment-component
                           {|| assignment-component}
assignment-component →     enumerated-assignment
                           | quantified-assignment
```

assignment-statement pozostáva z jedného alebo viacerých *assignment-component*s ktoré sú oddelené **||**. Existujú dva druhy *assignment-component*-ov: v *enumerated-assignment*-e sú premenné a ich príslušné výrazy ktorých hodnoty majú byť priradené vymenované. V *quantified-assignment*-e premenné a výrazy ktorých hodnoty majú byť priradené sa získajú tým že vo všeobecnom priradení nahradíme kvantifikované premenné všetkými prípustnými hodnotami.

Premenná sa môže na ľavej strane *assignment-statement*-u objaviť aj viackrát, je však na zodpovednosti programátora zabezpečiť pre akúkoľvek takúto premennú aby všetky možné hodnoty ktoré môžu byť tejto premennej v danom statemente (priradení) priradené boli identické. Z hľadiska vykonávania *assignment-statement*-u je každý *assignment-component* vykonávaný samostatne a nezávisle.

Popis enumerated-assignment-u

```
enumerated-assignment →    variable-list := expr-list
variable-list         →    variable {, variable}
expr-list             →    simple-expr-list
                           | conditional-expr-list
simple-expr-list      →    expr {, expr}
conditional-expr-list →    simple-expr-list    if boolean-expr
                           {~ simple-expr-list if boolean-expr }
```

Syntaktické jednotky *expr* a *boolean-expr* označujú výraz a booleovský výraz tak ako sú definované v jazyku PASCAL. *enumerated-assignment* priradzuje hodnoty výrazov na pravej strane **:=** korešpondujúcim premenným ktoré sú uvedené na ľavej strane **:=**. Toto priradenie je zhodné s tradičným niekoľkonásobným priradením: Najprv sa vyhodnotia výrazy na pravej strane a indexy na ľavej strane a nakoniec sa priradia hodnoty výrazov zodpovedajúcim premenným. Priradenie je úspešné len a len vtedy ak sa počet a typ premenných zhoduje so zodpovedajúcimi výrazmi.

Pri priradení s *conditional-expr-list* sa premenným na ľavej strane priradia hodnoty z toho *simple-expr-list* ktorého prislúchajúci booleovský výraz je pravdivý (true). Ak nie je žiadny z booleovský výrazov pravdivý premenné zostanú nezmenené. Ak je booleovský výrazov pravdivých viac tak všetky zodpovedajúce *simple-expr-list*y musia obsahovať rovnakú hodnotu. Takže ľubovoľný z nich môže byť použitý pri vykonaní priradenia. To tiež garantuje že

každý *assignment-statement* je deterministický : po vykonaní *assignment-statement* z daného stavu sa výpočet môže dostať len a iba do jediného stavu.

Príklady *enumerated-assignment* :

1. výmena x,y.

```
x,y := y,x
```

2. nastav x na absolútnu hodnotu y.

```
x := y   if y ≥ 0 ~ -y  if  y < 0
```

3. pridaj A[i] do premennej sum a zvýš premennú i o 1 za predpokladu že i je menej ako N

```
sum, i := sum + A[i], i+1     if  i<N
```

4. spoj dve usporiadane polia A a B do poľa C (predpokladáme že A a B sú usporiadané)

```
C[k] := min(A[i], B[j])
|| k := k+1
|| i := i+1   if  A[i] ≤ B[j]
|| j := j+1   if  A[i] ≥ B[j]
```

alebo

```
C[k], i  , j  , k  :=
A[i], i+1, j  , k+1   if A[i] < B[j] ~
B[i], i  , j+1, k+1   if A[i] > B[j] ~
A[i], i+1, j+1, k+1   if A[i] = B[j] ~
```

Popis quantified-assignment-u

```
quantified-assignment → <||quantification assignment-statement>
quantification        →    variable-list : boolean-expr ::
```

Premenné vo *variable-list-e* sa nazývajú kvantifikované (quantified) alebo viazané (bounded) premenné. *quantification* má pôsobnosť v priestore ktorý

mu vymedzujú zátvorky „<“ a „>“, tento priestor nazvime sektor. Booleovský výraz vnútri *quantification* môže obsahovať:

- premenné deklarované v programe
- konštanty
- viazané premenné z tohto - daného sektora
- viazané premenné vonkajšieho sektora, to jest sektora ktorý daný sektor obklopuje

Inštancia *quantification* (kvantifikácie) je množina hodnôt viazaných premenných pre ktoré je booleovský výraz vnútri *quantification* (kvantifikácie) pravdivý.

quantified-assignment udáva nula alebo viac *assignment-component* –ov ktoré získame z *assignment-statement* nahradením viazaných premenných ich inštanciami. Pre každu inštanciu dostaneme práve jeden *assignment-component* . Musí byť zaistené že existuje len konečný počet inštancií. Ak neexistuje žiadna inštancia tak príslušný *quantified-assignment* udáva prázdny príkaz (statement).

Príklady *quantified-assignment* :

1. pre dané polia A[0..N] a B[0..N] typu integer priraď max (A[i], B[i]) do A[i], pre všetky i, 0 ≤ i ≤ N.

```
< || i : 0≤ i ≤N :: A[i] := B[i] if A[i] < B[i] >
```

alebo

```
< || i : 0≤ i ≤N :: A[i] := max(A[1],B[i]) >
```

alebo

```
< || i : 0≤ i ≤N & A[i] < B[i] :: A[i] := B[i] >
```

2. vytvor maticu identity do poľa U[0..N, 0..N].

```
<|| i,j : 0≤i≤N & 0≤j≤N :: U[i,j] := 0 if i ≠ j ~ 1 if i = j>
```

alebo

```
<|| i,j : 0≤i≤N & 0≤j≤N & i≠j :: U[i,j] := 0>
|| <|| i : 0≤i≤N :: U[i,i] := 1>
```

alebo

```
<|| i : 0≤i≤N :: U[i,i] := 1
             || <|| j : 0≤j≤N & i≠j :: U[i,j] := 0>
>
```

Assign-section

assign-section	→	*statement-list*
statement-list	→	*statement* {**[]** stetement}
statement	→	*assignment-statement*
		| *quantified-statement-list*
quantified-statement-list	→	**< []** *quantification statement-list* **>**

assign-section špecifikuje statements(príkazy) programu. Symbol [] slúži ako separátor medzi statements(príkazmi).

quantified-statement-list udáva množinu statements(príkazov) nahradením viazaných premenných ich inštanciami v *statement-list*e. Ak neexistuje žiadna inštancia tak príslušný quantified-statement-list udáva prázdnu množinu príkazov (statements). Ako predtým aj tu savyžaduje aby bol počet inštancií konečný. Navyše pridávame nasledujúce obmedzenie na *boolean-expr* nachádzajúci sa v *quantification* : *boolean-expr* nesmie obsahovať premenne ktorých premenne sa môžu počas vykonávania programu meniť. Toto obmedzenie garantuje že množina statementov(príkazov) programu je fixná a žiadne príkazy sa nebudú počas vykonávania programu vytvárať a ani mazať.

Príklady *assign-section* :

1. program ktorý pole U[0..N, 0..N], bude vo fiznom bode obsahovať maticu identity. Tento problém síce riešia už vyššie spomenuté príklady na *quantified-assignment* . Navyše nahradenie ľubovoľného symbolu ||

symbolom [] (nahrádzať však treba tak by neboli porušené syntaktické pravidlá jazyka) tiež rieši daný problém.

Niektoré z riešení s rôznym počtom príkazov:

```
<[] i,j : 0≤i≤N & 0≤j≤N :: U[i,j] := 0 if i ≠ j ~ 1 if i = j>
```

toto riešenie obsahuje $(N+1)^2$ príkazov

alebo

```
<|| i,j : 0≤i≤N & 0≤j≤N & i≠j :: U[i,j] := 0>
[] <|| i : 0≤i≤N :: U[i,i] := 1>
```

toto riešenie obsahuje dva príkazy

alebo

```
<[] i : 0≤i≤N :: U[i,i] := 1
             [] <|| j : 0≤j≤N & i≠j :: U[i,j] := 0>
>
```

toto riešenie obsahuje N+1 príkazov.

Initially-section

Syntax tejto sekcie je taký istý ako v *assign-section* až na drobnú zmenu ktorou je nahradenie znaku priradenia z **:=** znakom **=**. *initially-section* priradzuje počiatočné hodnoty premenným. Počiatočná hodnota premennej je daná ako funkcia počiatočných hodnôt ostatných premenných a konštánt. Priradenia definujúce počiatočné hodnoty nesmú byť v cyklické. Množina priradení počiatočných hodnôt je syntakticky správny vtedy a len vtedy ak splňuje všetky nasledovné podmienky:

- premenná sa nachádza najviac raz na ľavej strane priradenia
- existuje usporiadanie priradení také že ľubovoľná premenná v *quantification* je buď viazaná premenná alebo premenná z ľavej strany priradenia, ktoré sa nachádza skôr v danom usporiadaní

§ existuje usporiadanie všetkých priradení čiže po tom čo boli rozvedené všetky kvantifikované priradenia, také že ľubovoľná premenná nachádzajúca sa na pravej strane priradenia alebo v indexe, sa nachádza aj na ľavej strane priradenia umiestneného v usporiadaní skôr ako dané priradenie.

Príklady *initially-section* :

1.

```
initially
      N = 3 []
      < || i : 0≤ i ≤N :: A[N-k] = k >
```

treba si všimnúť že nahradením symbolu [] symbolom || by v tejto *initially-section* bolo neprípustné, pretože N v kvantifikácií by sa nenachádzalo v žiadnom z predchádzajúcich priradení.

2.

```
initially
      B[0] = 0 || N = 2
      [] < [] i: 0≤i≤N :: A[i] = B[i-1] [] B[i] = A[i] >
```

toto priradenie sa dá rozviesť na nasledujúcu množinu priradení

```
A[1] = B[0] [] B[1] = A[1] [] A[2] = B[1] [] B[2] = A[2]
```

3.

```
initially
      y = … []
      x = 0 if y≥0
```

táto *initially-section* je syntakticky neprípustná, keďže je ekvivalentná s :

```
initially
      y = … []
      x = 0 if y≥0 ~
          x if y<0
```

Tu jasne vidno že premenná x sa nachádza na pravej strane priradenia bez toho aby bola definovaná v nejakom skoršom priradení. Čiže ak pravá strana priradenia obsahuje *conditional-expr-list* tak za každých podmienok vždy aspoň jeden booleovský výraz musí byť pravdivý.

initially-section definuje pre program vstupnú podmienku, čo je najsilnejší predikát ktorý platí nazačiatku. Tento predikát dosiahneme z *initially-section* tak že [] a || nahradíme & (logické and) a priradenie obsahujúce podmienku (*conditional-expr-list*) riešime nasledovne:

```
x := e0 if b0 ~ ... ~ 1 en if bn
```
rozpíšeme na
```
(b0 => (x = e0)) & ... & (bn => (x = en))
```

4. príklad na vstupnú podmienku

```
initially
      < || i, j : 0≤i≤N & 0≤j≤N ::
            U[i,j] =   0 if i ≠ j ~
                       1 if i = j
      >
```
udáva nasledovnú vstupnú podmienku:

```
      < & i, j : 0≤i≤N & 0≤j≤N ::
            ((i ≠ j) => U[i,j] = 0) & ((i = j) => U[i,j] = 1)
      >
```

Always-section

always-section sa používa na definovanie niektorých premenných programu ako funkcií iných premenných. Synatax *always-section* je rovnaká ako v *initially-section*. Premenná vyskytujúca sa na ľavej strane priradenie sa nazýva zrejmá – transparentná premenná (transparent variable). Transparentná premenná je funkciou netransparentných premenných a preto sa nevyskytuje na ľavej strane žiadneho priradenia v *initially-section* alebo *assign-section*, môže sa však vyskytovať na pravej strane. Takže transparentná premenná vlastne určuje invariant pre daný program. Aby bolo zaistené že každá transparentná premenná je dobre definovaná funkcia netransparentných premenných, tak pre ne platia rovnaké obmedzenia ako pre premenne v *initially-section*.

Používanie *always-section* nie je pre písanie UNITY programov nevyhnutné, malo by byť zrejmé že vynechaním *always-section* a nahradením všetkých transparentných premenných vo zvyšku programu ich príslušnými pravými stranami z ich definície vedie k vzniku nového syntakticky správneho programu ktorý je ekvivalentný s pôvodným. Na používanie *always-section* je však niekoľko dobrých dôvodov. Po prvé táto sekcia definuje množinu invariantov. Za druhé sa môžeme na transparentné premenné pozerať ako na makrá čím sa stáva UNITY program prehľadnejší a ľahšie pochopiteľnejší.

Quantified Expression

Ako už bolo skôr spomenuté *expr* a *boolean-expr* sú syntakticky definované ako v jazyku PASCAL. UNITY však ponúka možnosť využiť kvantifikovanie aj vo výrazoch, zavedením quantified expression.

expr → **<** *op quantification expr* **>**

Nahradením viazaných premenných vo vnútornom výraze(*expr*) dostaneme množinu výrazov následným aplikovaním operátora *op* medzi jednotlivé prvky tejto množiny dostaneme výraz ktorého hodnota je hodnotou daného kvantifikovaného výrazu. Keďže operátor *op* musí binárny asociatívny a komutatívny operátor tak je jedno v akom poradí bude aplikovaný na jednotlivé prvky množiny výrazov definovanej kvantifikáciou (*quantification*). Ak neexistuje žiadna inštancia pre danú kvantifikáciu (*quantification*) tak hodnota kvantifikovaného výrazu je jednotkový prvok operátora *op*. Operátory ktoré sú v UNITY používané sú: min, max, +, &(logické and), \/ (logické or), +, * (násobenie), ≡(ekvivalencia). Ich jednotkové prvky sú: +∞ , -∞, true, false, 0, 1, true ,uvedené v tom istom poradí.

Príklady quantified expression:

1. v nasledujúcich príkladoch platí N≥0

```
< \/ i : 0≤ i ≤N :: A[i]>
```

výraz je true(pravdivý) ak je niektorý z prvok poľa pravdivý(true)

```
< min i : 0≤ i ≤N :: A[i]>
```

výraz je rovný najmenšiemu prvku poľa

```
< + i : 0≤ i ≤N & A[i]≤A[j] :: 1>
```

výraz je rovný počtu prvkov z poľa A menších ako prvok A[j]

Príklady programov

Pre prvé tri príklady majú rovnakú *declare-section* a *initially-section* v ktorých sa zadeklaruje pole a jeho prvkom sa priradí náhodná hodnota

```
      declare N:integer;
            A: array[0..10] of integer;
            sorted: boolean
      initially
            N := 10 []
            <|| i : 0≤i≤N :: A[i]:= Random(0,20)>
```

```
Program Sort1
      ...
      assign
        < [] i : 0≤i<N ::
                  A[i],A[i+1] := A[i+1],A[i] if A[i] > A[i+1]
        >
end
```

Sort1 je jednoduchý Bubble sort ktorý vymení dva susedné prvky ak nie sú v poradí. Užitočnou heuristikou je kombinovať priradenia ktoré menujú podobné premenné do jedného priradenia. Zlúčením priradení pre všetky párne i a všetky nepárne i dostávame nasledujúcu obmenu predchádzajúceho programu:

```
Program sort2

     ...
     assign
       < || i : (0≤i<N)&(Even(i)) ::
                 A[i],A[i+1]  := A[i+1],A[i] if A[i] > A[i+1]  >
       []
       < || i : (0≤i<N)&(Odd(i))  ::
                 A[i],A[i+1]  := A[i+1],A[i] if A[i] > A[i+1]  >
       [] sorted := < & i: 0≤i<N :: A[i]≤A[i+1]>
end
```

pre oba *statement*y pre párne aj nepárne sú ich vnútorne *assignment-statement*y zhodné a tak môžme prepísať sort2 na nasledovne:

```
Program Sort3
     ...
     assign
       < [] j : 0≤j≤1 ::
          < || i: (0≤i<N)&(j=i mod 2) ::
                A[i],A[i+1]  := A[i+1],A[i] if A[i] > A[i+1] >
       >
end
```

Nasledujúci príklad počíta binomické koeficienty $\binom{n}{k}$ v poli **c[n,k]** pre všetky k : 0≤k≤n a všetky n: 0≤n≤N pre dané N. Využívame nasledujúce vzťahy: $\binom{n}{0}=\binom{n}{n}=1$ pre všetky n, $\binom{n}{k}=\binom{n-1}{k-1}+\binom{n-1}{k}$ pre všetky n>0 a 0<k<n.

```
Program binomical
      declare N:integer;
              c: array[0..10] of array[0..10] of integer
      initially N := 10
      assign
        < [] n : 0≤n<N ::
            c[n,0]  := 1 ||  c[n,n] := 1
            [] <|| k: 0<k<n :: c[n,k] := c[n-1,k-1]+c[n-1,k] >
        >
end
```

Poradie vykonávanie príkazov je v UNITY náhodné. Preto niektorým prvkom poľa c[n,k] môže byť priradená hodnota aj predtým než boli vypočítané hodnoty c[n-1,k-1] alebo c[n-1,k-1]. Nakoniec sa však všetkým c[n,k] priradia správne hodnoty. V tomto programe môžu byť nahradený ľubovoľný symbol || symbolom [], takéto nahradenie však nie možne vo všetkých UNITY programoch. Pre n = 0 sa prvok c[0,0] vyskytuje v priradení dvakrát ale zakaždým je mu priradená tá istá hodnota a tak je program korektný.

Všetky štyri programy sa nachádzajú v elektronickej prílohe ako súbory a dajú sa interpretovať mojím interpreterom.

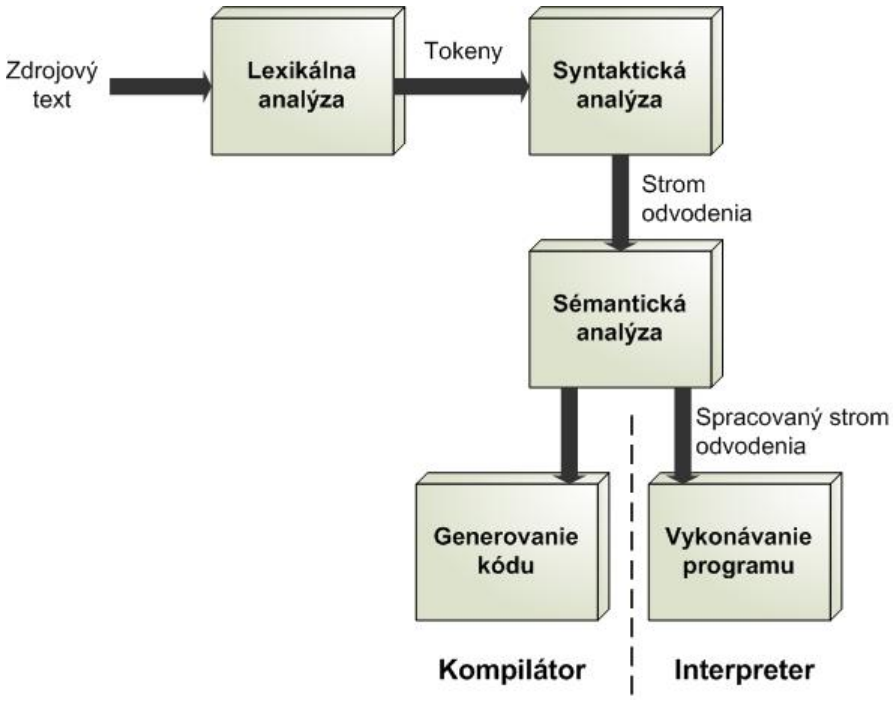
Kapitola 3

ÚVOD DO TEÓRIE KOMPILÁTOROV

Pri implementácií interpretera jazyka UNITY sin využíval poznatky z teórie kompilátorov. Kompilátor sa od interpretera líši len v poslednej fáze, v ktorej kompilátor zo spracovaného stromu odvodenia generuje spustiteľný strojový kód, zatiaľ čo interpreter stromom odvodenia prechádza a príkazy priamo vykonáva - interpretuje. Proces vytvárania spracovaného stromu odvodenia zo vstupného zdrojového textu je rovnaký. Táto kapitola podáva stručný úvod do teórie kompilátorov tých častí ktoré boli dôležité pri mojej diplomovej práci.

Oblasť kompilátorov je už veľmi dobre prebádaná a existuje bežne používaný systémový dizajn ktorý delí kompilátora na niekoľko podsystémov – fáz. Nasledujúci obrázok toto rozdelenie zobrazuje a ukazuje činnosť jednotlivých fáz.

obrázok 1- systémový dizajn kompilátora



V podkapitolách podobnejšie opíšem metódy, ktoré je možné používať na implementáciu jednotlivých fáz kompilácie a mnou vybrané metódy.

Lexikálna analýza

Lexikálny analyzátor je prvá fáza kompilátora. Jeho hlavnou úlohou je prečítať vstupné znaky a vytvoriť postupnosť „tokenov“, ktoré v ďalšej fáze použije syntaktický analyzátor - parser. Lexikálny analyzátor po prijatí príkazu „pošli nasledujúci token“ (get next token) od parseru, začne čítať vstupné znaky až kým nerozozná ďalší token.

Je niekoľko dôvodov prečo oddeliť analytickú fázu kompilácie na lexikálnu analýzu a syntaktickú analýzu:

1. Jednoduchší dizajn kompilátora. Oddelením tejto fázy sa celkový návrh zjednoduší a sprehľadní.
2. Efektivita a rýchlosť kompilátora. Vďaka oddeleniu sa môžeme venovať len konkrétnej činnosti a využiť špecializované techniky na zrýchlenie načítavania vstupu, napríklad použitím existujúcich techník, ako je práca s bufferom.
3. Zvýšená portabilita kompilátora. Lexikálny analyzátor môže odfiltrovať rôzne odlišnosti medzi jednotlivými platformami (napríklad rôzne zakončenie riadkov v operačnom systéme UNIX a Windows)

Tokeny, vzor (patterns), lexémy (lexemes)

Pri lexikálnej analýze sa používajú výrazy token, lexéma, vzor(pattern). V zdrojovom texte sa nachádzajú množiny znakových reťazcov (strings) pre ktoré je vygenerovaný rovnaký token na výstupe. Každá takáto množina je popísaná pravidlom ktorému sa vraví vzor (pattern) pre daný token. Každý zo znakových reťazcov danej množiny zodpovedá tomuto vzoru. Lexéma je sekvencia znakov vstupu ktorá zodpovedá vzoru pre nejaký token.

Príklady tokenov lexém a vzorov sú v nasledujúcej tabuľke:

tabuľka 1 – príklady tokenov lexém a vzorcov

TOKEN	PRÍKLAD LEXÉMY	VZOR (POPISANÝ CEZ REGULÁRNE VÝRAZY)
relation operation	<= , >= , = , < , > , <>	<= | >= | = | < | > | <>
identifier	a0, pi, count, begin	letter(letter|digit)*
number	5, 5.85, 585.7E-2	digit+(.digit+)?(E(+|-)?digit+)?

Vo väčšine programovacích jazykoch sa vyskytujú nasledujúce tokeny: identifikátory (identifiers), operátory(operators), kľúčové slová rezervované programovacím jazykom (keywords), konštanty (constants), znakové reťazce (literal strings) a interpunkčné symboly(punctuation symbols) ako zátvorky, bodka, dvojbodka . Keď jednému vzoru zodpovedá viac lexém je potrebná doplňujúca informácia ktorá nesie informáciu o konkrétnej nájdenej lexéme. Táto informácia sa prenáša v atribúte tokenu (napríklad ak v zdrojom texte nájdeme lexému 5.85 nestačí vrátiť iba token number ale treba poslať aj hodnotu nájdeného čísla).

Špecifikácia tokenov

Regulárne výrazy sú dôležitou a často využívanou notáciou na definovanie vzorov (patterns). Nasledujúca stať ma za úlohu priblížiť regulárne výrazy v žiadnom prípade však nepopisuje ich všetky možnosti.

Klasickým príkladom je regulárny výraz pre identifikátor v jazyku Pascal, ktorý vyzerá nasledovne: **letter(letter | digit)*** , a zodpovedajú mu lexémy, ktoré začínajú písmenom nasledovaným ľubovoľne dlhou (aj nulovou) kombináciou čísel a písmen.

Regulárny výraz je vybudovaný z jednoduchších regulárnych výrazov cez množinu definujúcich pravidiel. Každý regulárny výraz r definuje jazyk L(r), pomocou nasledovných pravidiel nad abecedou Σ:

- **ε** je regulárny výraz definujúci prázdny jazyk *L = {ε}*

- § ak a je symbol z Σ potom regulárny výraz **a** definuje jazyk *L(a) = {a}*
- § predpokladajme že **r** a **s** sú regulárne výrazy potom
  a) **r|s** definuje jazyk *L = L(r) È L(s)*
  b) **rs** definuje jazyk *L = L(r)L(s)*
  c) **r*** definuje jazyk *L = L(r)** nula alebo viac inštancií r
  d) **r+** definuje jazyk *L = L(r)L(r)** jedna alebo viac inštancií r
  e) **r?** definuje jazyk *L = L(r) È {ε}* nula alebo jedna inštancia r
  f) **(r)** definuje jazyk *L = L(r)*
- § a,b,c sú symboly z Σ potom regulárny výraz **[abc]** definuje regulárny výraz **a | b | c** . Regulárny výraz **[a-z]** je skratkou výrazu **a | b | ... | y | z** . Takto sa dajú jednoducho definovať triedy znakov (napríklad: [A-Za-z]|[A-Za-z0-9]* definuje už spomínaný identifikátor jazyka Pascal)

pričom pre operátory platí: *,+,? majú najvyššiu prioritu v danom poradí a sú zľava-asociatívne, spojenie ma druhú najvyššiu prioritu a je zľava-asociatívne, najnižšiu prioritu ma | a je tiež zľava-asociatívne.

Príklady regulárnych výrazov:

1. vzor (pattern) pre identifikátor v jazyku Pascal

```
letter    → [A-Za-z]
digit     → [0-9]
id        → letter | digit
```

2. vzor (pattern) pre číslo v jazyku Pascal

```
digits               → digit*
optional_fraction    → (.digits)?
optional_exponent    → ( E (+|-)? digits)?
number    → digits optional_fraction optional_exponent
```

Je zrejmé že nie všetky jazyky sa dajú popísať pomocou regulárnych výrazov, pre špecifikáciu tokenov veľkej väčšiny programovacích jazykov si s nimi

vystačíme. Trieda jazykov ktorú môžeme pomocou nich definovať je regulárna trieda jazykov *R*, ktorá je ekvivalentná s triedou jazykov konečných automatov L_{KA}. Pre každý regulárny výraz teda existuje konečný automat, akceptujúci rovnaký jazyk ako daný regulárny výraz. Pri implementácii lexikálneho analyzátora si teda vystačíme so simuláciou konečných automatov.

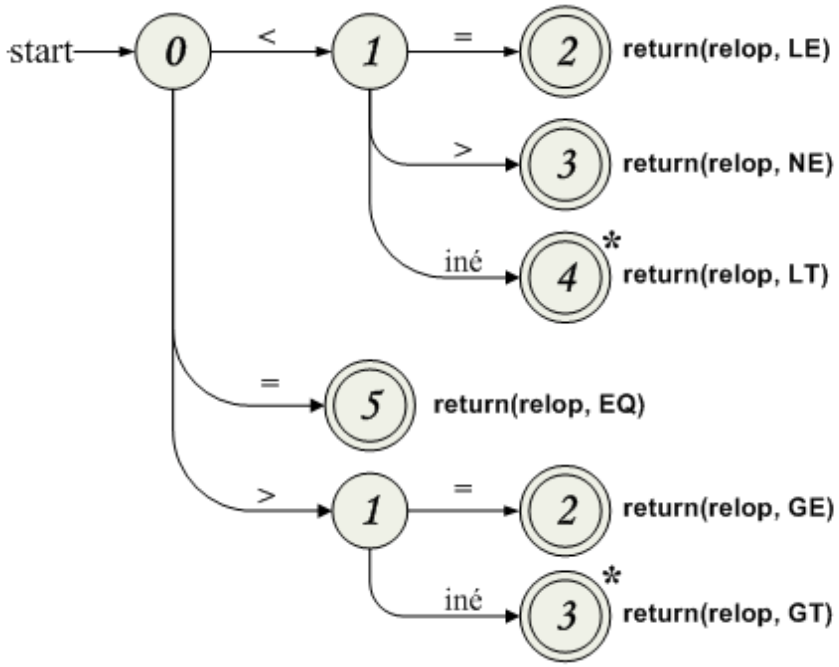
Ako rozoznať tokeny

Konečný automat sa dá zobraziť pomocou prechodového diagramu. Prechodový diagram znázorňuje, ako presne získame slovo (lexému) zo vstupu. Stavy v diagrame označujeme krúžkom. Sú navzájom prepojené šípkami (hranami alebo prechodmi). Pre každú hranu sú určené znaky, pre ktoré sa možno z aktívneho stavu presunúť do cieľového. Jeden stav je počiatočný. Niektoré stavy označíme ako akceptačné(na obrázku sú znázornené dvoma sústredenými kruhmi). Činnosť vyzerá nasledovne:

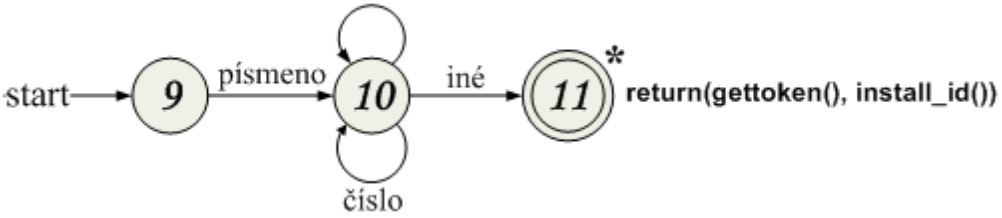
1. Načítaj znak zo vstupu
2. Nájdi hranu, ktorá má na sebe napísaný práve načítaný znak
3. Presuň sa do nového stavu po vybranej hrane
4. Ak prechod na načítaný znak neexistuje, vyhlás chybu
5. Pokračuj od bodu 1

Príklady prechodových diagramov:

obrázok 2 - prechodový diagram pre token relačný operátor



obrázok 3 - prechodový diagram pre token identifikátor



Lexikálna analýza bude používať konečný automat s dvomi hlavami. Takto upravený automat má dva ukazovatele, ktoré sa používajú na čítaný text. Jeden sa bude nachádzať vždy na začiatku čítaného slova, pričom druhý (nazývaný „look ahead") bude samotné slovo čítať. Ak sme dočítali slovo, posunie sa na jeho koniec aj prvý ukazovateľ. Dôvod prečo sa používajú dva ukazovatele je nasledovný: Na príkladoch s prechodovými diagramami sú stavy s číslami 4, 8 a 11 označené hviezdičkou. Mohlo by nastať, že napríklad v stave 1 načítame nejaký znak rôzny od = a >. V takomto prípade sa musíme vrátiť o jeden znak späť, a to je presne tam, kde stojí prvý ukazovateľ. V prípade 11 môžeme načítať

napríklad znak >, ten však už nepatrí k načítavanému slovu a preto sa takisto potrebujeme vrátiť. Vo veľa prípadoch si vystačíme s pozeraním dopredu len na jeden znak.

Lexikálny analyzátor používa viacero prechodových diagramov. Každý diagram spracováva niekoľko tokenov. Ľahko si možno predstaviť jeden čítajúci reálne čísla a jeden operátory (<, <=, <>, ...). Aj v tomto prípade môžeme využiť ukazovatele. Ak počas práce s jedným prechodovým diagramom zlyháme, stačí vrátiť „look ahead“ ukazovateľ na pozíciu prvého a pokračovať s ďalším diagramom, pokiaľ nenájdeme diagram, ktorý token rozpozná. V prípade, keď vhodný diagram neexistuje, slovo nie je z jazyka.

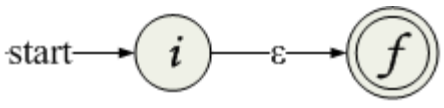
Keďže kľúčové slová (keywords) sú sekvencie znakov a teda takisto zodpovedajú vzoru pre identifikátor avšak identifikátorom nie sú. Tento problém sa vyrieši po prechode do finálneho stavu diagramu, ktorý rozpoznáva identifikátor a tam sa rozhodne či je nájdená lexéma identifikátor alebo kľúčové slovo. Do tabuľky symbolov si uložíme všetky kľúčové slová. Procedúra gettoken() sa pozrie do tabuľky symbolov a ak tam je nájdená lexéma tak vráti token kľúčové slovo inak vráti identifikátor. Procedúra install_id() vráti buď o ktoré kľúčové slovo ide alebo vráti smerník na hodnotu identifikátora.

Vytváranie konečného automatu z regulárnych výrazov

Regulárny výraz je vybudovaný z jednoduchších regulárnych výrazov. Pri vytváraní konečného automatu najprv zostrojíme konečné automaty pre jednoduchšie regulárne výrazy a následne ich spojíme podľa nasledovných pravidiel:

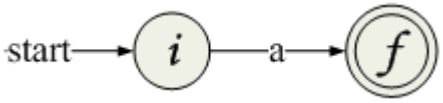
1. pre regulárny výraz **ε** zostrojíme nasledovný konečný automat:

obrázok 4 – automat pre ε



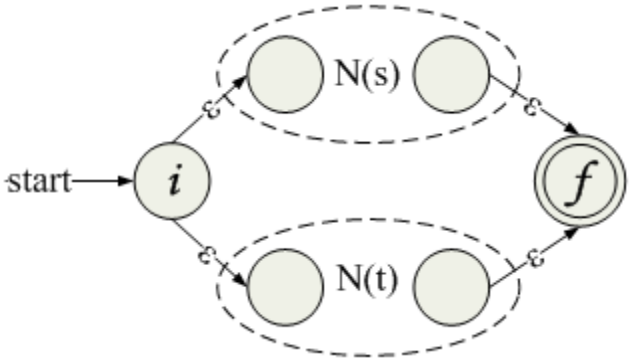
2. pre regulárny výraz **a** zostrojíme nasledovný konečný automat:

obrázok 5 – automat pre a



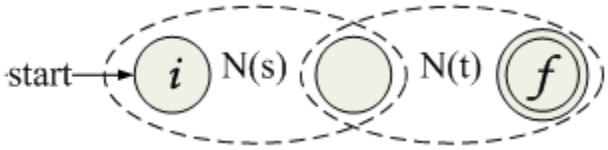
3. predpokladajme že N(s) je konečný automat pre regulárny výraz **s** a N(t) pre regulárny výraz **t**. Potom konečný automat pre **s|t** zostrojíme nasledovne

obrázok 6 – automat pre s|t



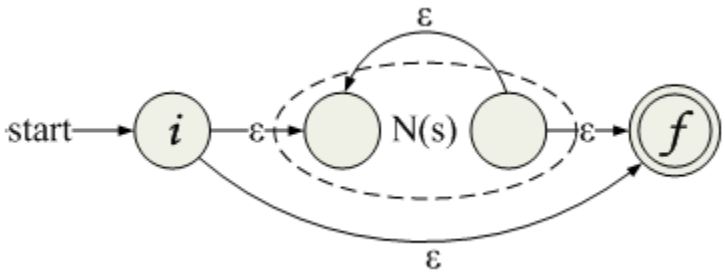
4. predpokladajme že N(s) je konečný automat pre regulárny výraz **s** a N(t) pre regulárny výraz **t**. Potom konečný automat pre **st** zostrojíme nasledovne

obrázok 7 – automat pre st



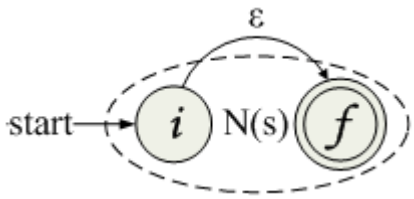
5. predpokladajme že N(s) je konečný automat pre regulárny výraz **s**. Potom konečný automat pre **s*** zostrojíme nasledovne

obrázok 8 – automat pre s*



6. predpokladajme že N(s) je konečný automat pre regulárny výraz **s**. Potom konečný automat pre **s?** zostrojíme nasledovne

obrázok 9 – automat pre s?



Je zrejmé že takto zostavený konečný automat bude akceptovať ten istý jazyk ako regulárny výraz, z ktorého konštrukcia vychádzala. Tento konečný automat je nedeterministický, avšak štandardnou konštrukciou sa dá previesť na deterministický, ktorého implementácia - prevod na kód je jednoduchý. Často využívanou možnosťou je vytvorenie obslužnej rutiny pre každý stav. Ak zo stavu vychádzajú nejaké hrany, potom načítame jeden znak(posunieme druhý ukazovateľ) a podľa neho vyberieme hranu, ktorou budeme pokračovať. Teda budeme pokračovať v kóde, ktorý obsluhuje nový stav. Ak nenájdeme vhodnú hranu a stav, v ktorom sa nachádzame, nie je akceptačný, vrátime druhý ukazovateľ na začiatok slova a pokračujeme rovnako len s ďalším diagramom. Ak už nie je diagramom, s ktorým by sa dalo pokračovať, kompilátor ohlási chybu. Naprogramovanie lexikálneho analyzátora nie je zložité, navyše však existujú automatické generátory lexikálnych analyzátorov ktoré prácu ešte viac uľahčujú a na základe definície regulárnych výrazov vygenerujú hotový lexikálny analyzátor. Týmto nástrojom sa budem venovať v ďalšej kapitole.

Syntaktická analýza

Každý programovací jazyk má pravidlá ktoré predpisujú syntaktickú štruktúru dobre definovaného programu. Na syntax programovacích jazykov sa často používajú bezkontextové gramatiky.

Parser získava od lexikálneho analyzátora tokeny a zisťuje či takto získaná postupnosť tokenov môže byť generovaná gramatikou programovacieho jazyka. Parser zároveň vytvára parsovací strom (respektíve strom ovdovenia pre daný program v gramatike daného programovacieho jazyka) a zároveň by mal ohlasovať syntaktické chyby, ktoré môže vstup obsahovať.

Existujú tri všeobecné typy parserov pre gramatiky. Všeobecné prasovacie metódy ako CYK(nezávisle objavený Cockom a Youngerom v roku 1967 a Kasamim v roku 1965) alebo Erleyho algoritmus dokážu spracovať ľubovoľnú gramatiku. Tieto metódy sú však pomalé neefektívne a náročné na výpočtovú silu. Preto sa na vytváranie kompilátorov zvyčajne nepoužívajú. Metódy bežne používané pri vytváraní parserov sa delia na buď zhora-nadol (top-down) alebo zdola-nahor (bottom-up).

Pri oboch metódach sa vstup spracováva zľava doprava, pričom top-down vytvára syntaktický strom od hora smerom nadol k listom a bottom-up z dola smerom ku koreňu. Obe pracujú pomerne rýchlo ale dokážu rozpoznať len niektoré jazyky, konkrétne tie, ktoré sú generované LL alebo LR gramatikami. Obe gramatiky sú však dostatočne „výrečné", aby pri návrhoch väčšiny programovacích jazykov stačili.

Detekcia chýb

Ak by mal kompilátor spracovávať iba korektné programy jeho dizajn a implementácia by sa výrazne uľahčila. Avšak programátori často píšu programy s chybami a dobrý kompilátor by mal pomáhať v ich identifikácií a lokalizácii. Program môže mať chyby v rôznych úrovniach (lexikálne, syntaktické, sémantické a logické). Ak parser nájde počas svojej práce syntaktickú chybu môže ju ohlásiť a parsovanie ukončiť. To by však znamenalo že kvôli každej jednej chybe by sa opakovane musel proces spúšťať odznova.

Preto sa používajú metódy, ktorými sa kompilátor pokúsi z chyby zotaviť a pokračovať ďalej, čím sa umožní detekovať viacero chýb naraz. Existujú nasledujúce základné typy stratégií:

1. **panický mód** (panic mode): Ak narazí syntaktická analýza na chybu, zahadzujú sa prichádzajúce tokeny dovtedy, kým sa nenájde nejaký, ktorý je označený ako „bezpečný“ (označované aj ako synchronizačný). Až po tomto tokene sa pokúsi pokračovať v spracovávaní ďalej. Ak sa následne podarí úspešne spracovať niekoľko nasledujúcich tokenov tak sa prejde zo stavu zotavovania sa z chyby späť do normálne parsovania. Príkladom bezpečného tokenu môže byť *bodkočiarka* a „*}*” v jazyku C alebo *end* v Pascale.
2. **oprava konštrukcií jazyka** (Phrase-level recovery): Ak nájdeme chybu, pokúsime sa ju opraviť. Častým príkladom môže byť doplnenie bodkočiarky. Opravený výsledok však môže často zmeniť význam programu v porovnaní s tým, čo mal na mysli programátor.
3. **chybové pravidlá** (Error production): V tomto prípade sa doplnia do gramatiky jazyka nové pravidlá, každé pre nejakú známu chybu. Napríklad v jazyku C vždy po if nasleduje zátvorka. Môžeme vytvoriť pravidlo, ktoré v opačnom prípade ohlási chybu „*očakávaná ľavá zátvorka*“.
4. **globálna oprava** (Global correction): Metódy založené na tejto stratégii využívajú špeciálne algoritmy, ktoré dokážu vrátiť pre chybný vstupný text výsledok, v ktorom bolo urobených minimum úprav(doplnenie znaku, ...) a ktorý je gramatikou generovateľný. Tieto algoritmy sú však veľmi náročné na výpočtovú silu a preto sa v praxi nepoužívajú. Zároveň si treba uvedomiť, že program, ktorý dá ako výstup opravný algoritmus nemusí byť to, čo myslel autor.

Syntaktická analýza zdola-nahor (bottom-up)

Nasledujúca časť popisuje parsovaciu metódu zdola-nahor, ktorú využívam aj pri implementácií svojho interpretera.

Tento prístup k syntaktickej analýze sa niekedy označuje aj ako „shift-reduce“ (posunovo redukčná schéma). Metódy založené na tomto prístupe sa snažia vytvoriť strom odvodenia smerom od listov ku koreňu. Celý proces si možno predstaviť ako postupné redukovanie vstupného reťazca na počiatočný symbol gramatiky spätným nahradzovaním pravidiel. V každom redukčnom kroku sa určitý podreťazec zodpovedajúci pravej strane nejakého pravidla gramatiky nahradí neterminálnom z ľavej časti pravidla. Ak sa tento podreťazec v každom kroku vyberie správne, tak získame v opačnom poradí najpravejšie(rightmost) odvodenie (odvodenie kde vždy nahradzujeme najpravejší neterminál).
Existuje viacero metód implementácie, zvyčajne sa pri väčšine z nich využíva zásobník. Vstup sa číta po tokenoch zľava doprava. Krok keď sa práve prečítaný token uloží na vrch zásobníka sa nazýva shift a krok keď na hornej časti zásobníka pravá strana pravidla a nahradí ľavou stranou pravidla sa nazýva reduce.

Existujú gramatiky, pri ktorých sme si nie istý, či treba v istom momente redukovať (a ktorým pravidlom) alebo posunúť symbol zo vstupu do zásobníka. Možné konflikty sú:

1. **shift-reduce** nastáva v prípade, keď sa nevieme rozhodnúť, či treba načítať znak zo vstupu alebo redukovať vetnú formu v zásobníku.
2. **reduce-reduce** konflikt nastáva, keď máme na výber z viacerých pravidiel ktoré možno redukovať (musíme sa rozhodnúť pre jedno pravidlo z viacerých možných).

LR parser

Metóda LR syntaktickej analýzy je veľmi efektívna bottom-up metóda, ktorú možno použiť na rozpoznávanie všetkých programovacích jazykov, pre ktoré vieme napísať bezkontextovú gramatiku. Metóda sa nazýva LR(k), kde *L* je v tom význame, že vstup sa číta zľava doprava, *R* hovorí že sa vytára najpravejšie (rightmost) odvodenie v opačnom poradí a *k* určuje koľko symbolov – tokenov je použitých ako lookahead pri robení parsovacích rozhodnutí (či sa má vykonať

shift alebo reduce krok). Ak je (k) vynechané tak k je rovné 1, čiže LR používa len jeden lookahead token.

Výhody LR syntaktickej analýzy:

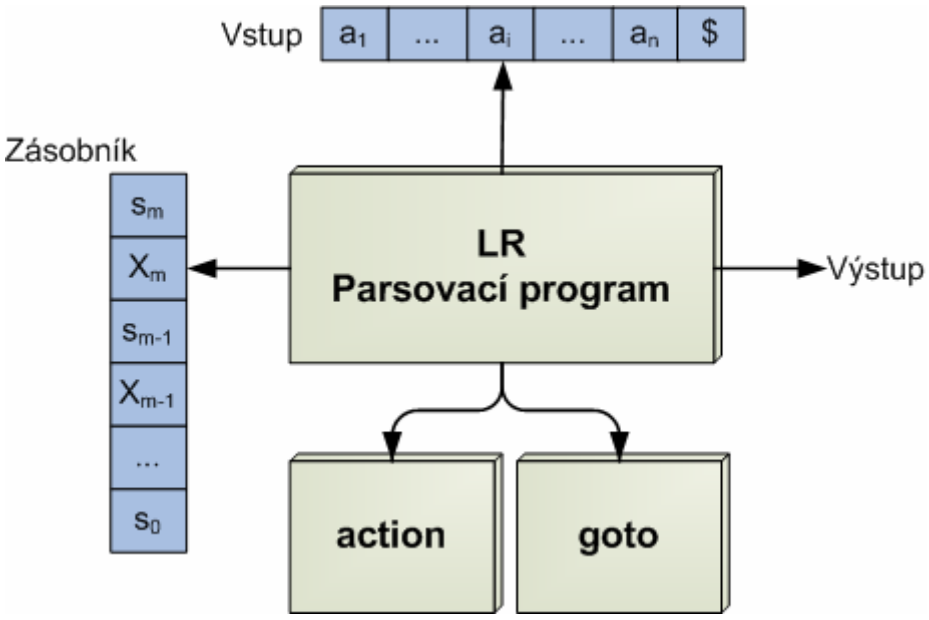
- Možno ju použiť na rozpoznávanie všetkých programovacích jazykov, pre ktoré vieme napísať bezkontextovú gramatiku.
- Dodnes to je jediná známa najvšeobecnejšia metóda nepoužívajúca backtracking založená na shift-reduce.
- Dokáže veľmi rýchlo odhaliť chybu.

Hlavnou nevýhodou je že pre gramatiku typického programovacieho jazyka je príliš pracné napísať LR parser ručne. Avšak existuje viacero LR parser generátorov pre ktoré stačí špecifikovať bezkontextovú gramatiku a generátor automaticky vyprodukuje LR parser pre danú gramatiku. Automatickým generátorom sa budem venovať v ďalšej kapitole, teraz však popíšem princíp, na ktorom LR parser funguje.

Algoritmus LR syntaktickej analýzy

Schematická forma LR parseru je zobrazená na nasledujúcom obrázku:

obrázok 10 – LR parser



LR praser sa skladá z riadiaceho-parsovacieho programu, zásobníka, vstupnej pásky(ukončenej symbolom $) a tabuľky, ktorá ma dve časti : action a goto. Riadiaci program je pre všetky LR parsery rovnaký rozdielne sú len tabuľky action a goto. Riadiaci program načítava symboly zo vstupnej pásky jeden po druhom. Program používa zásobník na uchovávanie reťazcov, ktoré majú formu: $\mathbf{s_0\ X_1\ s_1\ X_2\ s_2}$..... $\mathbf{X_m\ s_m}$ kde s_m je na vrchu zásobníka. X_i označuje symbol z abecedy a s_m označuje stav. Každý stav sumarizuje informácie o všetkom čo sa nachádza nižšie v zásobníku. Na základe kombinácie stavu na vrchu zásobníka a aktuálneho vstupného symbolu sa robia parsovacie rozhodnutia.
Konfigurácia LR parsera je pár ktorého prvú časť tvorí zásobník a druhú zvyšok vstupu:

$$\mathbf{(s_0\ X_1\ s_1\ X_2\ s_2 \ldots\ X_m\ s_m;\ a_i\ a_{i+1}\ a_{i+2}\ \ldots\ a_n\ \$)}$$

Parser pracuje nasledovne: Nech $\mathbf{a_i}$ je práve spracovávaný vstupný symbol a na vrchu zásobníka je stav $\mathbf{s_m}$. Ďalší krok závisí od hodnoty **action[s_m, a_i]**. Do úvahy prichádzajú nasledovné 4 hodnoty:

1. ak tabuľka action[s_m, a_i] = **shift s** (s je stav) tak parser vykoná krok shift s výslednou konfiguráciou:

   $$(s_0\, X_1\, s_1\, X_2\, s_2 \ldots\ldots X_m\, s_m\, a_i\, s; a_{i+1}\, a_{i+2} \ldots\ldots a_n\, \$)$$

   parser pridal na vrch zásobníka aktuálny vstupný symbol $\mathbf{a_i}$ a na neho stav **s** z tabuľky action. $\mathbf{a_{i+1}}$ sa stáva aktuálnym vstupným symbolom.

2. ak tabuľka action[s_m, a_i] = **reduce A →β** (A →β je pravidlo gramatiky) tak parser vykoná krok reduce s výslednou konfiguráciou:

   $$(s_0\, X_1\, s_1\, X_2\, s_2 \ldots\ldots X_{m-r}\, s_{m-r}\, A\, s;\ a_i\, a_{i+1}\, a_{i+2} \ldots\ldots a_n\, \$)$$

   kde s = goto[s_{m-r} , A] a r je dĺžka pravej strany pravidla (r = |β|). Parser vybral z vrchu zásobníka 2r symbolov (r stavov + r symbolov gramatiky) čím sa na vrch zásobníka dostal stav s_{m-r}. Následne sa na vrch zásobníka vložila ľavá strana pravidla podľa ktorého sa redukovalo teda neterminál A. Nakoniec sa na vrch zásobníka pridal sumarizujúci stav s ktorý sa získal z tabuľky goto (s = goto[s_{m-r} , A]). Aktuálny vstupný symbol sa pri kroku reduce nemení takže ním zostáva $\mathbf{a_i}$. Tabuľky LR parseru sú vytvorené tak aby sa sekvencia gramatických symbolov $X_{m-r+1}...X_m$ vybratých zo zásobníka vždy rovnala pravej strane pravidla β podľa ktorého sa redukuje. (z tohto dôvodu sa pri reálnej implementácií v zásobníku udržujú iba stavy)

3. ak tabuľka action[s_m, a_i] = **accept** aktuálny vstupný symbol je **$** a v zásobníku je počiatočný symbol gramatiky tak je parsovanie úspešne ukončené.

4. ak tabuľka action[s_m, a_i] = **error** tak vstupný text nevyhovuje gramatike.

Na vytváranie LR parsovacích tabuliek existuje viacero techník: Simple LR (SLR), Canonical LR, Lookahead LR (LALR) a Generalized LR (GLR). V praxi najčastejšou technikou je LALR pretože tabuľky, vyprodukované touto technikou sú značne menšie ako tie ktoré vyprodukuje Canonical LR. Syntaktické konštrukty väčšiny bežne používaných programovacích jazykov sa dajú vyjadriť pomocou LALR gramatík, na rozdiel gramatík SLR ktoré majú výpočtovú silu o niečo menšiu. Na porovnanie SLR a LALR majú vždy rovnaký počet stavov a ten je zvyčajne niekoľko stoviek pre programovací jazyk typu Pascal, na rozdiel od nich to je pre Canonical LR až niekoľko tisíc stavov. LR praser generátor, ktorý som využil vo svojej diplomovej práci tiež používa LALR techniku. GLR je rozšírením klasického LR prasovania a umožňuje parsovanie nedeterministických (gramatiky, pri ktorých nastáva reduce-reduce konflikt) a nejednoznačných (gramatiky, pri ktorých nastáva shift – reduce konflikt) gramatík. Prasovacia tabuľka GLR môže obsahovať konflikty a ak sa má vykonať konfliktný prechod tak sa parsovací proces rozvetví a pre každú vetvu sa udržuje vlastný zásobník.

Cieľom tejto kapitoly nebolo podrobne vysvetliť teóriu kompilátorov ale iba uviesť čitateľa do problematiky tých častí teórie ktoré súvisia s mojou diplomovou prácou. Detailný popis techník vytvárania LR parsovacích tabuliek, teóriu stojacu za top – down syntaktickou analýzou a aj teóriu zvyšných častí kompilátora je možné nájsť v knihe Compilers: Principles, Techniques, and Tools z ktorej som čerpal aj ja (referencia je uvedená na konci mojej diplomovej práce).

Kapitola 4

NÁSTROJE NA AUTOMATICKÉ GENEROVANIE ANALYZÁTOROV

Lexikálna analýza

Napísať ručne lexikálny analyzátor nie je príliš ťažké, zvyčajne je však táto robota pracná a nezáživná. Preto boli vyvinuté automatické generátory lexikálnych analyzátorov, ktorých väčšina je založená na teoretickom základe popísanom v predošlej kapitole. Tieto nástroje urýchlili a zjednodušili vytvorenie lexikálnych analyzátorov, navyše takto vygenerovaný analyzátor je zvyčajne efektívnejší ako ručne napísaný. Najznámejší a najbežnejšie používaný takýto nastroj je LEX.

LEX je štandardným generátorom lexikálnych analyzátorov na systémoch UNIX a je obsiahnutý v POSIX štandarde. Autormi LEX-u sú Eric Schmidt a Mike Lesk. LEX na základe vstupnej špecifikácii vyprodukuje lexikálny analyzátor, ktorého kód je v cieľovom programovacom jazyku(existujú viaceré verzie s podporou rôznych cieľových jazykov: C, C++, Perl,…). Populárna voľne dostupná verzia LEX-u je flex (fast lexical analyzer), ktorý možno nájsť na väčšine Linux distribúcií. Vstupná špecifikácia LEX-u je s menšími obmenami využívaná aj v generátore JLex, a C# Lex. JLex generuje kód lexikálneho analyzátora v programovacom jazyku Java a aj samotný JLex je v jazyku Java napísaný. C# Lex je prepis JLex-u do programovacieho jazyka C# čím sa ponúkajú možnosti automatického generovania lexikálnych analyzátorov aj na platforme .NET.

Keďže som sa rozhodol vytvoriť svoj interpreter v jazyku C# logicky som použil nástroj C# Lex. Vstupný súbor používa nasledovnú špecifikáciu:

```
user code
%%
C#Lex  directives
%%
regular expression rules
```

direktíva %% oddeľuje jednotlivé sekcie ktorých úloha je nasledovná:

- **user code** – táto časť je prekopírovaná do výsledného kódu analyzátora bez zmeny. V tejto časti sa importujú triedy ktoré sa v analyzátore využívajú.
- **C#Lex directives** – v tejto časti sa deklarujú direktívy napríklad:
  - **cup** : generuje kód ako triedu na priame použitie s C#Cup parserom
  - **line** : zapne počítanie riadkov
  - **class <name>** : premenuje vygenerovanú triedu (implicitne: Ylex)
  - **function <name>** : premenuje funkciu cez ktorú parser žiada o token (implicitne: yylex())
  - **type <name>** : mení návratový typ funkciu cez ktorú parser žiada o token (implicitne: Ytoken)
  - **state state0[,state1....]** : definuje jeden alebo viac stavov v ktorých sa môže lexikálny analyzátor nachádzať. Ak nie je žiadny definovaný tak je analyzátor stále v stave YYINITIAL
- **regular expression rules** – táto sekcia obsahuje definíciu tokenov pomocou regulárnych výrazov a akcií asociovaných s týmito tokenmi. Všetky majú nasledujúcu formu: [<*states*>] <*expression*> { <*action*>}. expression je v tvare regulárneho výrazu a action obsahuje užívateľský kód v jazyku C#.

Podrobný popis a manuál **C#Lex** sa nachádza na internetovej adrese: http://www.infosys.tuwien.ac.at/cuplex/lex.htm#1.%20About%20C#%20CUP.

Syntaktická analýza

Existuje pomerne veľké množstvo automatických generátorov parserov (v anglickej literatúre sa uvádzajú aj pod názvom Compiler-compiler) odlišujúcich sa od seba hlavne prasovacou metódou ktorú implementujú, výstupným programovacím jazykom v ktorom je vygenerovaný kód parsera. Niektoré zároveň umožňujú špecifikovať aj tokeny a následne vytvárajú aj lexikálny analyzátor. Stručný prehľad prináša nasledujúca strana.

tabuľka 2 – automatické generátory parserov

Produkt	parsovacia metóda		výstupný jazyk	Gramatika / kód	Lexer	CC platforma	IDE	Licencia
AnaGram	bottom - up	LALR	ANSI C, C++			Win32	áno	Proprietary
ACCENT	bottom - up	GLR	C	zmiešaný	externý (Lex)	všetky (ANSI C)	nie	Open source (GPL)
ANTLR	top - down	LL(k)	C++, C#, Java, Python	zmiešaný	vytváraný	všetky (Java)	nie	Open source (BSD)
Beaver	bottom - up	LALR	Java	zmiešaný	externý	všetky (Java)	nie	Open source (BSD)
Bison	bottom - up	LALR, GLR	C, C++	zmiešaný	externý (Flex)	GNU/Linux, Unix, Win32	nie	Open source (GPL)
BYACC	bottom - up	LALR	C	zmiešaný	externý	Unix, Win32	nie	Public domain
BYACC/J	bottom - up	LALR	C, Java	zmiešaný	externý (JFlex)	Irix, Linux, Solaris, Win32	nie	Public domain
Coco/R	top - down	LL(k)	C++, C#, Java	zmiešaný	vytváraný	všetky (Java), .NET, Linux	nie	Open source (GPL)
CppCC	top - down	LL(k)	C++				nie	Open source (GPL)
CSTools	bottom - up	LALR	C#	zmiešaný	vytváraný	.NET	nie	Open source
CUP	bottom - up	LALR	Java	zmiešaný	externý (JLex)	všetky (Java)	nie	Open source (GPL)
Elkhound	bottom - up	GLR	C++, Ocaml	zmiešaný	externý	Unix, Win32	nie	Open source (BSD)
GOLD	bottom - up	LALR	ANSI C, C#, Delphi, Java, Python, Visual Basic, Visual C++	oddelený		Win32	áno	Open source (zlib/libpng)
Grammatica	top - down	LL(k)	C#, Java	oddelený	vytváraný	všetky (Java)	nie	Open source (GPL)
jacc	bottom - up	LALR	Java	zmiešaný		všetky (Java)	nie	Open source (BSD)
JavaCC	top - down	LL(k)	Java	zmiešaný	vytváraný	všetky (Java)	áno	Open source (BSD)
jay	bottom - up	LALR	Java			Unix	nie	
LEMON	bottom - up	LALR	C				nie	Public domain
LRgen	bottom - up	LALR	C++, hociaký ak sa prepíše skeleton	oddelený	vytváraný	Win32	nie	Proprietary
Parser Objects	top - down	LL(k)	Java	zmiešaný		všetky (Java)	nie	Open source (ZLib/LibPNG)
PRECC	top - down	LL(k)	C			Dos, Unix	nie	
Rats!	Packrat		Java	zmiešaný	vytváraný	všetky (Java)	nie	Open source (LGPL)
SableCC	bottom - up	LALR	Java, s "altgen" engine-om: C, C++, C#, OCAML a Python	oddelený	vytváraný	všetky (Java)	nie	Open source (LGPL)
SLK	top - down	LL(k)	C, C++, C#, Java	oddelený	externý	Win32, Linux	nie	Proprietary
Spirit	top - down	LL(k)	C++	zmiešaný			nie	Open source (Boost)
TextTransformer	top - down	LL(k)	C++	zmiešaný	vytváraný	Win32	áno	Proprietary (free version)
Visual Parse++	bottom - up	LALR	C++,C++, C#, Java	oddelený	vytváraný	Win32	áno	Proprietary
Yacc (AT&T)	bottom - up	LALR	C ,existuju aj porty pre iné jazyky	zmiešaný	externý	Unix	nie	Open source (various)
YooParse	bottom - up	LR	C++		externý (YooLex)		nie	Open source (MIT)

Medzi najpoužívanejšie generátory parserov patri YACC (Yet Another Compiler Compiler), ktorý vytvoril Stephen C. Johnson vo firme AT&T pre operačný systém Unix. Na jeho základe boli vytvárané mnohé ďalšie generátory parserov používajúce techniku LALR a veľa z nich s miernymi úpravami prevzalo aj špecifikáciu vstupného súboru medzi ne patria aj: GNU bison (napísaný pre GNU projekt a dostupný na väčšine linuxových distribúcií); jacc, javacc, Cup, Jay (generujúce kód parsera v jazyku Java). Keďže špecifikácia programovacích jazykov C# a Java si je veľmi podobná netrvalo po uvedení platformy .NET dlho kým niekto prepísal niektorý z open source generátorov (s otvoreným kódom) tak aby generoval kód parsera v jazyku C# miesto jazyku Java. Pri mojej implementácií som sa rozhodoval medzi generátormi C# Cup a portom Jay pre C# z projektu Mono.

Autorom Jay portu pre C# je Miguel de Icaza a je súčasťou open source projektu Mono zastrešeného firmou Novell, ktorý poskytuje sadu nástrojov pre vývoj aplikácií pre platformu .NET. Z týchto nástrojov je to hlavne C# kompilátor vytvorený pomocou tohto Jay portu a Common Language Runetime. Mono je na rozdiel od .NET nástrojov od firmy Microsoft spustiteľný pod viacerými operačnými systémami (Linux, FreeBSD, UNIX, Mac OS X, Solaris a Windows).

C# Cup vznikol spolu s C# Lex na Technical University Vienna a ich autorom je Samuel Imriška. C# Cup je preklad populárneho generátora parserov - Cup do jazyka C#. Napriek tomu že pre lexikálnu analýzu používam C# Lex, ktorý sa dá jednoducho cez direktívu prepojiť na C# Cup, rozhodol som sa nakoniec použiť port Jay pre C# (ďalej len Jay) a o hlavne z dôvodu aby som si v praxi lepšie precvičil prepojenie lexikálnej analýzy na syntaktickú.

Špecifikácia vstupného súboru je obdobná ako u generátora Yacc a vyzerá nasledovne:

```
declarations
%%
grammar rules
%%
subroutines
```

direktíva %% oddeľuje jednotlivé sekcie. Z prvej sekcie sa na začiatok výstupného súboru prekopíruje bez zmeny všetok kód ktorý je uzavretý v zátvorkách %{ a %{. Pomocou tejto direktívy môžme do parseru importovať potrebné knižnice zadať namespace a triedu samotného parsera. Ďalej sa v prvej sekcii definujú tokeny pomocou direktívy:

```
%token   nazovtokenu1 nazovtokenu2 . . .
```

Každý názov nezadefinovaný ako token (čiže terminálny symbol abecedy) je v pravidlách gramatiky braný ako neterminál a musí sa vyskytovať na ľavej strane aspoň jedného pravidla.

Ďalej v tejto sekcii možno určiť prioritu (precedence) a asociáciu (association) tokenom čím sa môže predísť parsovacím konfliktom. Umožňuje to písanie jednoduchších gramatík, ktoré sú potom lepšie čitateľné ako by boli tie ktoré by zaisťovali tu istú funkčnosť ale bez použitia direktív priority a asociácie. Ako príklad môže slúžiť jednoduchý jazyk matematických výrazov s binárnymi operátormi (+,-,/,*,=):

```
%token '='  '+'  '-'  '*'  '/' NAME

%right '='
%left  '+'  '-'
%left  '*'  '/'
```

Priorita a asociácia je definovaná sériou riadkov ktoré začínajú na `%left`, `%right` alebo `%nonassoc` a sú nasledované zoznamom tokenov. Všetky tokeny na rovnakom riadku majú rovnakú prioritu. Tokeny na nižších riadkoch majú vyššiu prioritu a asociatívnosť je určená kľúčovým slovom na začiatku riadku. Vďaka tomu bude vstup tvaru:

a = b = c*d - e - f*g

vypočítaný ako:

a = (b = (((c*d)-e) - (f*g))

Nakoniec v tejto sekcii môžeme cez nasledovnú direktívu zadať počiatočný neterminál gramatiky:

```
%start   symbol
```

ak nie je počiatočný symbol zadaný tak sa zaň považuje ľavá strana prvého pravidla zo sekcie určujúcej pravidlá gramatiky (grammar rules section).

Druhá sekcia - grammar rules section definuje gramatiku. Pravidlo A → a B c zapíšeme cez:

```
A       : a B c;
```

Ak je niekoľko gramatických symbolov, ktoré majú rovnakú ľavú stranu môžeme použiť direktívu využívajúcu symbol „|“:

```
A       : B C D;
A       : E F;
A       : G;
```

sa teda dá zapísať ako

```
A       : B C D
        | E F
        | G;
```

Pre každé pravidlo sa dá zadefinovať akcia (užívateľom definovaný kód, uzavretý v „{“„}“ uvedený za pravidlom), ktorá sa vykoná pri redukcii daným pravidlom. Návratová hodnota tejto akcie je dostupná cez pseudo-premennú $$. Hodnoty terminálov(získané z lexikálneho analyzátora) a neterminálov(získané z predošlých akcií) pravej strany pravidla sú dostupné v pseudo-premenných $1, $2, $3 ... podľa ich poradia zľava doprava. Napríklad sekcia definujúca gramatiku pre jednoduchý jazyk matematických výrazov s binárnymi operátormi (+,-,/,*,=), ktorý bol spomenutý pri vyššom príklade by mohla vyzerať nasledovne:

```
expr   :expr '=' expr { symTable[$1] = symTable[$3];}}
       |expr '+' expr {$$ = (int)$1 + (int)$3;}
       |expr '-' expr {$$ = (int)$1 - (int)$3;}
       |expr '*' expr {$$ = (int)$1 * (int)$3;}
       |expr '/' expr {$$ = (int)$1 / (int)$3;}
       |NAME {$$ = symTable[$1];}
       ;
```

Kde symTable je hashovacia tabuľka typu integer.

Posledná sekcia – subroutines slúži na definovanie funkcií, ktoré môžu byť využívane v akciách priradených k pravidlám, prípadne sa tam môže nachádzať kód lexikálneho analyzátora.

Podrobný manuál k špecifikácií vstupného súboru generátora parserov Yacc možno nájsť na adrese http://dinosaur.compilertools.net/yacc/index.html . Ako už bolo vyššie spomenuté tato špecifikácia je použiteľná aj v generátore Jay.

Prepojenie nástrojov C# Lex a Jay

Prepojenie ľubovoľného lexikálneho analyzátora na Jay vyžaduje aby tento lexikálny analyzátor implementoval nasledovné jednoduché rozhranie:

```
      interface yyInput {
            bool advance ();
            int token ();
            Object value ();
      }
```

metóda `advance()` sa posúva na ďalší token, ak nastane chyba pri čítaní vstupu tak vyvolá výnimku (exception) ak už nemá čo čítať na vstupe tak vráti hodnotu false. Metóda `token()` vracia identifikátor načítaného tokenu a metóda `value()` vracia jeho hodnotu. Toto rozhranie implementuje aj môj lexikálny analyzátor vygenerovaný cez nástroj C# Lex.

Kapitola 5

PLATFORMA .NET

Od svojej implementácie som požadoval aby skompilovaná verzia interpretera bola spustiteľná pod rôznymi operačnými systémami. To je dosiahnuteľné len v moderných jazykoch, pre ktoré existuje run-time (behové) prostredie fungujúce ako prostredník medzi aplikáciou a operačným systémom. Tým som vylúčil možnosť použiť jazyk C++, pre ktorý síce existuje najviac nástrojov na automatické generovanie analyzátorov, ale nedá sa vyhnúť nutnosti skompilovať zdrojové súbory pre rôzne operačné systémy zvlášť. Dve najrozšírenejšie platformy, ktoré ponúkajú nezávislosť od operačného systému sú platforma JAVA a platforma .NET. Funkcionalitou sú si tieto dve platformy veľmi podobné avšak aplikácie pod platformou JAVA musia byť naprogramované v jedinom programovacom jazyku tejto platformy, ktorý sa volá tiež JAVA. Na rozdiel od toho platforma .NET ponúka celú množinu programovacích jazykov, v ktorých môžu byť aplikácie naprogramované. Jazyky platformy .NET sú si rovnocenné a preto výber programovacieho jazyka závisí prakticky len od firemných, alebo osobných preferencií toho ktorého jazyka. Jedným z programovacích jazykov z tejto množiny je aj programovací jazyk C#, ktorý bol uvedený spolu s touto platformou a práve tento jazyk som sa rozhodol použiť pri implementácii interpretera.

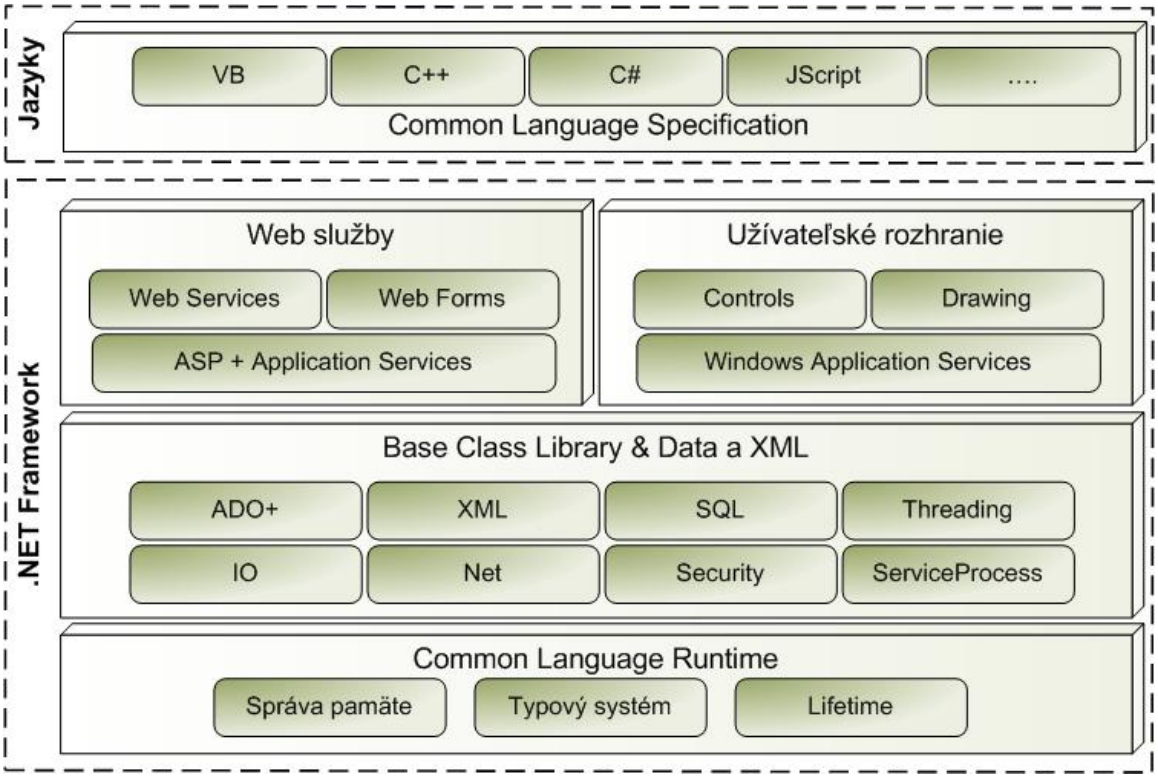
Popis platoformy .NET

.NET je platforma od spoločnosti Microsoft a ktorej zámerom je nahradiť dnes už vyše desaťročie starú platformu Win32. Platforma .NET funguje ako nadstavba nad operačným systémom a jej jadrom je .NET Framework.

Systém .NET Framework sa skladá z behového prostredia Common Language Runtime (CLR) a knižníc. Behový systém CLR si možte predstaviť ako virtuálny stroj, ktorý realizuje základnú infraštruktúru. Nad CLR je vybudovaný celý zvyšok .NET Frameworku.

Jednotlivé súčasti .NET Framework-u zobrazuje nasledujúci obrázok:

obrázok 11 - .NET Framework



Nad CLR sa nachádza niekoľko sád hierarchicky umiestnených knižníc. Základom je knižnica nazvaná Base Class Library. Nad ňou nasleduje podpora pre prístup k dátam a pre prácu s XML. Obidve kinižnice vytvárajú objektovo orientovanú podoporu pre efektívne a jednotné budovanie aplikácií.

Poslednú vrstvu tvoria dve knižnice uľahčujúce programovanie webových aplikácií a klasických aplikácií s užívateľským rozhraním. Treba zdôrazniť fakt že ide o jazykovo nezávislé knižnice, dostupné z ľubovoľného vývojového nástroja. Poslednú horizontálnu vrstvu tvorí nelimitovaná množina programovacích jazykov. V spodnej časti sa nachádza Common Language Specification (CLS) definujúca základné vlastnosti, ktoré sú očakávané v každom programovacom jazyku platformy .NET. Nad ňou nasledujú jednotlivé programovacie jazyky. V súčasnej dobe sú firmou Microsoft podporované štyri - Visual Basic, C++, JScript (Microsoft implementácia jazyku Java) a celkom nový programovací

jazyk C#. Množina nie je uzatvorená a ľubovoľný výrobca ju môže rozšíriť o novú položku. Počet jazykov tretích strán dnes presahuje hodnotu desať a určite nie je konečný. [1]

Na pochopenie služieb dostupných na celej platforme je potrebné trocha detailnejšie popísať najnižšiu vrstvu – Common Language Runtime.
Koncepcia run-time prostredia a knižníc nie je nová. V jazyku Visual Basic je to napríklad knižnica VBRUNxxx.dll, ktorá musí byť prítomná na každom počítači, ktorý sa bude snažiť spúšťať programy v ňom napísané. Takisto existujú run-time knižnice funkcií jazykov C a C++ a to napríklad objektovo orientované knižnice MFC či ATL. Pre jazyk JAVA je run-time prostredie dopracované na vyššiu úroveň a tvorí ho takzvaný virtuálny stroj (Java Virtual Machine).

Dôvodov na vytvorenie nového run-time prostredia bolo hneď niekoľko:

- § výrazne zjednodušiť vývoj aplikácií
- § pripraviť robustné a bezpečné prostredie pre beh aplikácií
- § podporiť veľké množstvo programovacích jazykov
- § zjednodušiť nasadenie a administráciu aplikácií [2]

Výsledkom je behové prostredie, v ktorom môžu byť bez problémov spúšťané aplikácie vytvorené v rôznych programovacích jazykoch. Ide o princíp, ktorý nazývame možnosť spolupráce medzi jazykmi respektíve cross-language interoperability. To je dosiahnuté vďaka tomu že všetky programovacie jazyky systému .NET splňujú kritéria CLS (Common Language Specification). Behové prostredie ponúka sadu run-time služieb, dostupných vo forme tried a štruktúr implementujúcich rôzne verejné rozhrania. Množinu služieb môžeme rozdeliť do dvoch kategórií. Prvá zaobaľuje existujúce Win32 funkcie a služby a druhá ju doplňuje o služby vyžadované modernými vývojovými postupmi a sú to najmä služby:

---

[1] Programujeme .NET aplikace, str. 20-21, autor: Dalibor Kačmář; Computer Press, 2001

[2] Programujeme .NET aplikace, str. 22, autor: Dalibor Kačmář; Computer Press, 2001

- garbage-collection na systémovej úrovni riadiaci životnosť objektov alokovaných v pamäti
- kompilátory z jazyka Microsoft Intermediate Language (MSIL) na natívny kód procesoru
- typovú kontrolu založenú na prítomnosti metadat popisujúcich všetky typy použite v aplikácií
- deklaratívny bezpečnostný model
- podpora pre pokročilé ladenie a profilovanie aplikácií
- integrácia s existujúcim COM modelom [3]

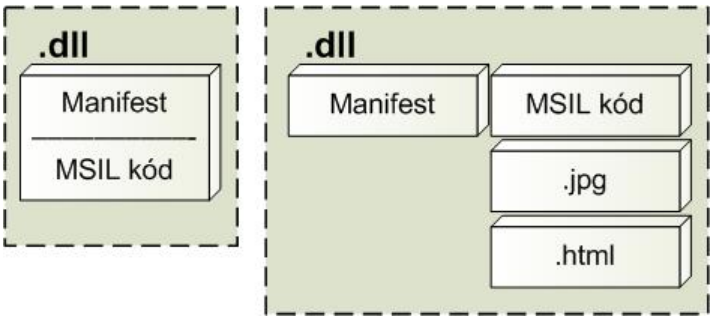
Veľmi dôležitým princípom, s ktorým sa v prostredí CLR stretneme je tvz. riadený kód čo je kód spúšťaný pod ochranou CLR a zároveň je týmto prostredím riadený. Riadený kód využíva výhody CLR napríklad automatickú správu pamäti (garbage collector), jednotného typového systému a metadat. To zaručuje aby sa všetky aplikácie chovali rovnako nezávisle od programovacieho jazyka v ktorom boli vytvorené.[4]
Niektoré z jazykov platformy .NET sú schopné generovať aj kód, ktorý sa explicitne vzdáva výhod a služieb ponúkaných CLR, takýto kód nazývame neriadeným.

Výstupom kompilátoru každého z jazykov schopných generovať riadený kód je Microsoft Intermediate Language (MSIL) čo je plnohodnotný procesorovo nezávislý jazyk (podobný assembleru) v ktorom je možné písať vlastné aplikácie. Oproti assembleru je však oveľa vyspelejší a vyniká schopnosťami charakteristickými pre vyššie programovacie jazyky, akými sú práca s objektmi, volanie virtuálnych metód, priama manipulácia s prvkami poľa či techniky zaisťujúce generovanie a zachytávanie výnimiek. Dôvodom na zavedenie medzijazyka je snaha o jednoduchú prenositeľnosť a spustiteľnosť existujúceho

[3] Programujeme .NET aplikace, str. 22, autor: Dalibor Kačmář; Computer Press, 2001
[4] myslíme v jazyku C#, str. 33-34, autor: Tom Archer, preklad: Jri Hynek; Grada Publishing, 2002

kódu medzi rôznymi hardwarovými platformami. Snaha o preklad do medzijazyka môže byť známa z platformy JAVA kde je obdobou MSIL takzvaný ByteCode.

obrázok 12 - Jednosúborová a multisúborová verzia assembly



.NET Framework zavádza koncepciu Assembly – programovej jednotky určenej k nasadeniu a opakovanému použitiu. Skladá sa z jedného alebo viacerých súborov obsahujúcich MSIL kód, zo súborov obsahujúcich ďalšie zdroje a povinného manifestu. Assembly môže mať príponou .exe alebo .dll. Formát týchto súborov je však už odlišný, ako bol pri platforme Win32 a bez nainštalovaného .NET Frameworku assembly nemožno spustiť. Assembly sú spravované a ukladané v assembly cache, ktorá umožňuje používanie viacerých verzií jedného súboru, dokonca súčasne z jedného programu. Assembly môže na prvý pohľad pripomínať dnešné dynamicky linkované knižnice (knižnice dll). Na assembly a ich správu sú však kladené ďaleko vyššie požiadavky vďaka čomu sa konečný užívateľ aplikácii vyhne problému, ktorý sa na platforme Win32 často označoval ako „DLL Hell". Ten vznikal keď rôzne aplikácie využívajú tu istú knižnicu ale v jej rôznych verziách pričom tieto verzie majú rôzne rozhrania.

MSIL určuje jednotný normalizovaný typový systém CTS (Common Type System), ktorý môže byť používaný všetkými programovacími jazykmi splňujúcimi kritéria CLS (Common Language Specification). Typový systém je flexibilný a umožňuje rozširovanie o nové dátové typy, ktoré vyzerajú a pracujú úplne rovnako ako vlastné typy systému. Programátori tak majú možnosť

pracovať so všetkými typmi rovnakým spôsobom bez nutnosti rozlišovania medzi predefinovanými typmi a vlastnými typmi. Dôsledkom je že assembly napísané v jednom jazyku môžu byť bez problémov využívané v aplikáciách napísaných v inom jazyku systému .NET. Napísať podobnú knižnicu DLL v systéme Win32, ktorej funkcie by mohli využívať rôzne programovacie jazyky bolo obtiažne, keďže knižnice DLL nešpecifikovali presné dátové typy a tak vznikali nekompatibility.

Keďže v súčasnosti neexistuje procesor, ktorý by dokázal priamo vykonávať inštrukcie jazyka MSIL, tak CLR systém sa využíva aj pre preklad kódu MSIL do strojového kódu na príslušnej hardwarovej platforme. Na to slúži prekladač JIT, takzvaný JITter (z anglického jsut-in-time). V závislosti od okolností môžu byť ku konverzii kódu z jazyka MSIL do strojového kódu použité tri rôzne JITtery (Generovanie kódu pri inštalácií, metóda JIT, metóda EconoJIT) .

Platforma .NET svojou koncepciou v mnohom pripomína platformu Java, ponúka však množinu viacerých programovacích jazykov. Navyše táto množina nie je uzavretá a je možne ju pri dodržaní CLS rozšíriť o ľubovoľný programovací jazyk. Trendom je prejsť z platformy Win32 na platformu .NET tak ako sa v minulosti prešlo z platformy DOS na Win32.
V auguste 2000 firma Microsoft, Hewlett-Packard a Intel spolupodali návrhy špecifikácie pre Common Language Infrastructur (CLI) a špecifikácie programovacieho jazyka C# medzinárodnej štandardizačnej organizácií Ecma. Následne sa za spoluúčasti ďalších firiem a organizácií (IBM, Fujitsu Software, Plum Hall, Monash University and ISE) pretransformovali na štandardy.[5]
So štandardizáciou sa rýchlo začali objavovať projekty tretích strán čím sa platforma .NET dostáva aj na iné operačné systémy ako Windows a tým sa zvyšuje jej atraktívnosť u programátorov. Existujú aj implementácie s otvoreným

[5]ECMA C# and Common Language Infrastructure Standards
http://msdn.microsoft.com/netframework/ecma/

zdrojovým kódom, kde treba spomenúť napríklad DotGNU Project a hlavne Mono Project ktorého súčasť Jay som využil aj v tejto diplomovej práci.

Implementovanie môjho UNITY interpretera v jazyku C# nad platformou .NET sledovalo nasledovné dôvody. Prvým bolo moja snaha hlbšie sa oboznámiť s touto platformou a konkrétne s jej novým jazykom C#. Ďalším bola možnosť skompilovania zdrojových súborov pre rôzne operačné systémy. Pri implementácií sa navyše ponúkla možnosť prezentovať možnosť spolupráce medzi jazykmi a tak si používateľ interpretera môže doprogramovať funkcie, ktoré mu v interpreteri chýbajú v ľubovoľnom jazyku platformy .NET a následne ich pri spustení interpretácií importovať.

Kapitola 6

IMPLEMENTÁCIA

Príbuzné implementácie

Od vytvorenia teórie UNITY bolo vyvinutých viacero implementácií, ktoré ju využívajú. Implementácie o ktorých som získal informácie možno rozdeliť na dve hlavné kategórie. V prvej kategórií sú prekladače a kompilátory zväčša ako súčasť rozsiahlejších systémov, do druhej patria dokazovacie systémy, ktoré slúžiace na dokazovanie správnosti UNITY programov, avšak negenerujú žiadny spustiteľný kód.

Implementácie kompilátorov a prekladačov:

- Paralelná implementácia Davida DeRoure z Southampton University kompiluje jazyk UNITY (napísaný v jazyku EuLisp) do koncového jazyka BSP-occam, kde kompilátor samotný vystupuje ako súčasť ESPRIT projektu PUMA.
- Martin Huber z University of Karlsruhe vytvoril prekladač z jazyka UNITY do jazyka MPL. Prekladač je navrhnutý pre MasPar MP-1 a MP-2 SIMD.
- S. Radha a C. R. Muthukrishnan uverejnili prenosnú verziu implementácie UNITY pre Von Neumanové počítače.
- Implementácia od autorov Adam Granicz, Daniel M. Zimmerman, a Jason Hickey prekladá UNITY program do C abstraktnej syntaxe a za použitia front-end nástroja Phobes Mojave kompilátora. Výhodou tejto implementácie je možnosť verifikovania správnosti UNITY cez MetaPRL – dokazovací systém teorém, ktorý je súčasťou Mojave kompilátora.
- Najpríbuznejší mojej implementácii je Toy UNITY compiler - kompilátor, ktorého autorom je Nils Grimsmo z Norwegian University of Science and Technology. Obdobne ako v mojej práci je paralelizmus simulovaný sekvenčne, UNITY špecifikácia je skompilovaná do jazyka C++. Pri vývoji boli taktiež použité automatické generátory z rodiny generátorov Lex a Yacc.

Implementácie dokazovacích systémov:

- Dokazovací systém HOL-UNITY autora Flemminga Andersena, ktorý využíva všeobecný dokazovací systém teorém HOL implementovaný na University of Cambridge.
- Coq-UNITY autorov Heyda a Creguta, využívajúci dokazovací systém Coq
- Paulsonova implementácia UNITY v prostredí Isabelle.

Lexikálna analýza

Vstupný súbor môjho interpretera pre C# Lex možno nájsť v elektronickej prílohe v súbore **scanner.lex**.

V user code časti sú zadefinované menné priestory (nemaspace) a importujú sa potrebné menné priestory, ktorých triedy sú potrebné pre lexikálny analyzátor. Jedným z dôležitých menných priestorov je `UNITY.Language.Parser`, v ktorom sú v triede Token zadefinované menné konštanty pre číselné identifikátory tokenov (napríklad číselná hodnota tokenu „+“ je `285` a je definovaná v konštante `Token.PLUS`). Tento menný priestor, tiež definuje rozhranie `yyParser.yyInput`, ktoré je lexer-om implementované, čím sa zabezpečuje prepojenie s parserom.

V časti C#Lex directives deklarujem direktívy - medzi hlavné patria:

- **`%implements`** `UNITY.Language.Parser.yyParser.yyInput` – určuje rozhrania, ktoré musí trieda lexera implementovať
- **`%type`** `int` – definuje návratový typ funkcie ktorá vracia tokeny (`Yytoken()`)
- **`%line`** – zapína počítanie riadkov prístupné cez premennú `yyline`
- **`%char`** – zapína počítanie znakov prístupné cez premennú `yychar`
- **`%state`** `COMMENT1, COMMENT2, STRING` - definujem stavy, v prvé dva slúžia na odfiltrovanie komentárov zo vstupu a tretí na načítanie textového reťazca

Ďalej je v tejto sekcii samotná implementácia rozhranie `yyParser.yyInput` a definícia funkcie `GetLocation()`,ktorá vráti pozíciu práve spracovávaného tokenu vo vstupnom súbore, kde pozícia je definovaná riadkom, počtom znakov

od začiatku riadku a počtom znakov od začiatku vstupu. Túto funkciu používam pri ohlásení chyby aby pomohla k jej lokalizácií. V poslednej časti tejto sekcie sú definované triedy znakov, ktoré sa využívajú v nasledujúcej sekcii definícií regulárnych výrazov pre jednotlivé tokeny.
Výsledná trieda implementujúca lexikálny analyzátor je v súbore **scanner.lex.cs**.

Syntaktická analýza – budovanie syntaktického stromu

Vstupný súbor môjho interpretera pre Jay port do jazyku C# možno nájsť v elektronickej prílohe v súbore **jparser.jay**.
V časti ***declarations*** definujem menný priestor a importujem potrebné menné priestory, ktorých triedy lexikálny analyzátor bude využívať. V mennom priestore `UNITY.Language.Tokenizer` je implementovaný lexikálny analyzátor a v `UNITY.Language.ILTree` sú implementované triedy, z ktorých inštancií sa počas syntaktickej analýzy zostaví parsovací strom. Ďalej sa v tejto časti definujú tokeny ich priorita a asociácie. Na konci je definovaný počiatočný neterminál `program`.
V ***grammar rules*** časti je definovaná gramatika jazyka UNITY. Ku každému pravidlu je zároveň priradená akcia, ktorá vytvára príslušnú časť parsovacieho stromu.
Napríklad:

```
quantification  : variable_list COLON  boolean_expr DOUBLE_COLON
            {
             $$ = new Quantification((ArrayList)$1, (Expression) $3);
            }
```

akcia sa vyvolá pri redukovaní pravidla:

quantification → variable_list **:** boolean_expr **::**

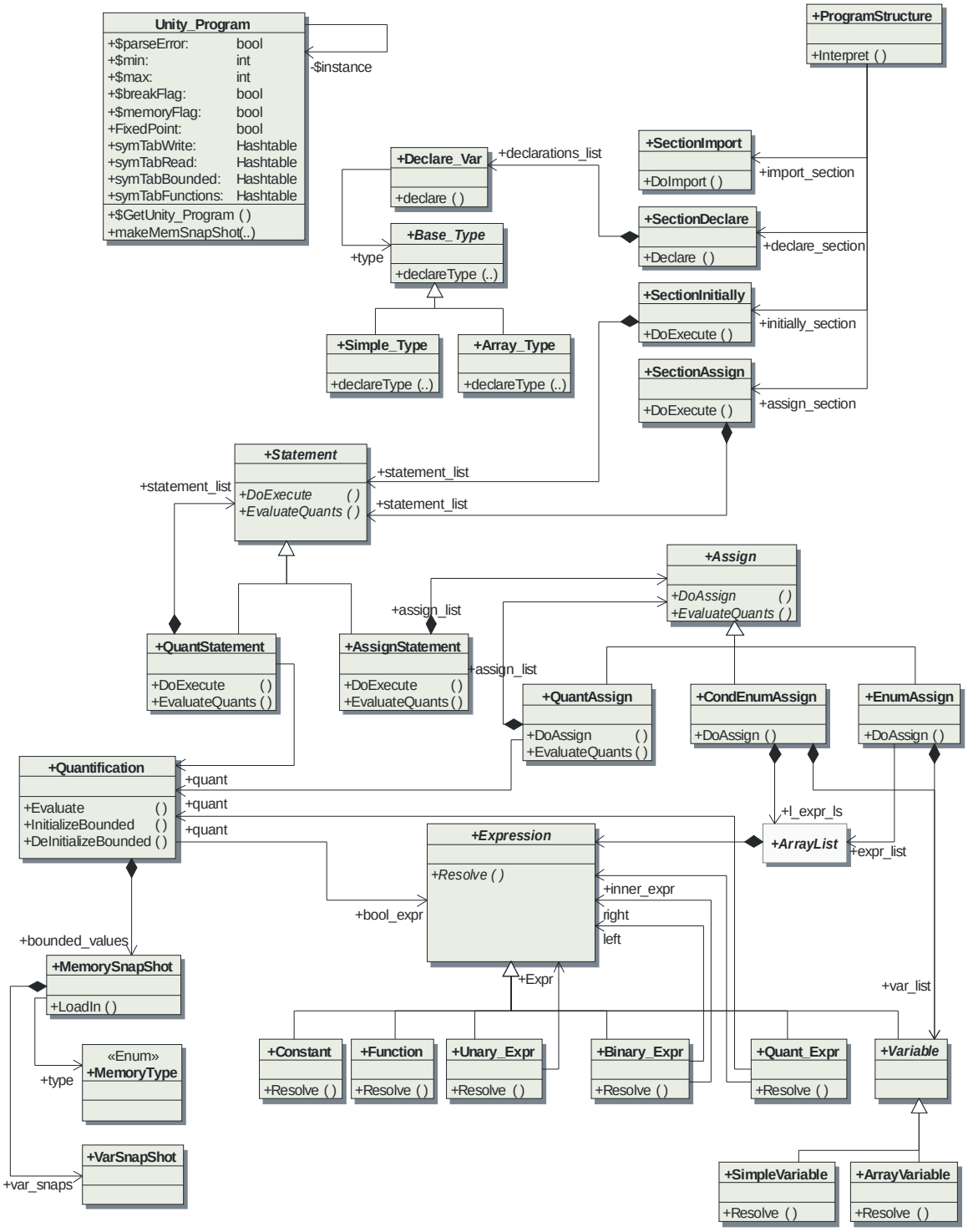
pričom sa vytvorí nový objekt - inštancia triedy `ILTree.Quantification` cez konštruktor
`Quantification(ArrayList bounded_vars, Expression bool_expr)`.
Neterminálny `variable_list` a `boolean_expr` sa pri bottom-up parsovaní

spracovali skôr, takže príslušne inštancie tried `ArrayList` a `ILTree.Expression`, vytvorené pri ich redukovaní, sú určite k dispozícií.

Ak je vstupný súbor korektným programom UNITY tak je po ukončení syntaktickej analýzy (dosiahnutí počiatočného neterminálnu) vytvorený prasovací strom dostupný cez inštanciu triedy `ILTree.ProgramStructure`. Štruktúra parsovacieho stromu by mala byť zrejmá z nasledujúceho UML diagramu tried, ktorý popisuje vzťahy medzi triedami ktoré ho tvoria.

UML Diagram tried (Class diagram)

obrázok 13 – UML Diagram tired



Ak nie je vstupný UNITY program syntakticky korektný tak sa interpreter pokúsi nájsť čo najviac chýb. Na zotavenie sa po chybe používam nasledovné stratégie:

1. **panický mód** (panic mode): v pravidlách používam špeciálny neterminál `error`. Ak sa pri parsovaní vyskytne chyba, tak parser vyberá stavy zo zásobníku pokiaľ nenarazí na stav, v ktorom je možné na `error` token vykonať ďalší parsovací krok. Tento krok následne aj vykoná, s tým že sa zo vstupu sa zahadzujú tokeny, až pokiaľ sa nenarazí na token, ktorý môže po vykonanom kroku legálne nasledovať. Pri detekovaní chyby prejde parser do chybového stavu a zotrvá v ňom pokiaľ sa mu úspešne nepodarí prepársovať tri tokeny za sebou. Ak sa narazí na chybu a parser je v error stave, tak sa negeneruje žiadne chybové hlásenie. Takéto ošetrenie chýb netreba programovať ručne, túto funkcionalitu zabezpečujú všetky generátory parserov odvodené z Yacc-u. Treba iba vhodne zvoliť miesta kam umiestniť error token.
2. **chybové pravidlá** (Error production): Medzi pravidlá som dal aj zopár chybových pravidiel na miesta kde som predpokladal možnosť častého výskytu chýb (hlavne pri separátoroch (**||**, **[]**, **;**), ktoré v UNITY oddeľujú niektoré programové elementy, čiže za posledným elementom by sa nemali vyskytovať).

Poslednú časť - ***subroutines*** špecifikácie vstupného súboru pre Jay vo svojej implementácii nevyužívam, keďže jej funkcionalitu zabezpečuje importovanie menných priestorov, čiže subrutiny sú definované v separátnych súboroch.

Interpretovanie

Stav programu sa udržiava v inštancii triedy `Unity_Program`, ktorá je zadefinovaná ako singleton (design pattern), a teda počas behu programu existuje iba jediná inštancia tejto triedy. Singleton zabezpečuje pohodlný prístup k jej jedinej inštancii z ľubovoľného bodu programu, volaním statickej metódy `GetUnity_Program()`. Stav UNITY programu je definovaný obsahom premenných, ktoré sú udržované v hašovacích tabuľkách (hashtables):

- globálne premenné v `symTabWrite` a `symTabRead`.

§ kvantifikované premenné v `symTabBounded`.

Trieda `Unity_Program` ďalej deklaruje hašovaciu tabuľku funkcií, ktorá udržuje informácie o preddefinovaných funkciách a funkciách, ktoré naimportoval užívateľ.

Po vytvorení úplného parsovacieho stromu (teda po dosiahnutí začiatočného neterminálu) sa spustí metóda `Interprete()` triedy `ProgramStructure`.

```
public void Interpret()
{
      if(this.import_section != null)
            this.import_section.DoImport();
      if(this.declare_section != null)
            this.declare_section.Declare();
      if (!Unity_Program.parseError)
      {
            if(this.initially_section != null)
```

V nej sa najprv volaním metódy `DoImport()` triedy `SectionImport` naimportujú .NET assembly s užívateľom definovanými funkciami, ktoré chce užívateľ v UNITY používať a nie sú v interpreteri obsiahnuté v preddefinovaných funkciách. Import sekcia v definícii UNITY jazyka neexistuje, je to črta mojej implementácie, ktorá je užitočná a zároveň sa ňou dá demonštrovať prepojiteľnosť programovacích jazykov v systéme .NET (cross-language interoperability).

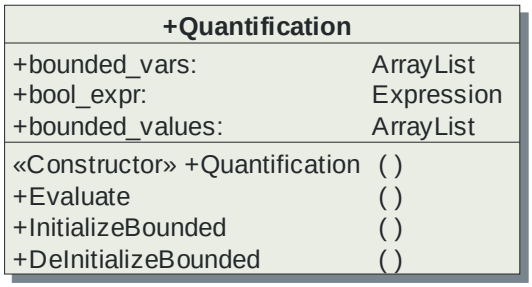
V *`declare-section`* implementovanej v triede `SectionDeclare` sa volaním metódy `Declare()` zadeklarujú premenné – naplnia sa hašovacie tabuľky globálnych premenných `symTabWrite` a `symTabRead`. Interpreter implementuje len typy boolean, integer a jednorozmerné/viacrozmerné polia týchto typov ich deklaráciou sa vytvoria ich obdoby v programovacom jazyku C# (int; bool; int[ ]; bool[, ,];). Takto vytvorené dátové štruktúry sa uložia do hašovacej tabuľky pod kľúčom, ktorý tvorí názov premennej.

Interpretovanie priraďovacích príkazov sa deje v initially a assign section cez metódy `DoExecute()` príslušných tried `SectionInitially` a `SectionAssign`. Nakoľko samotné priraďovacie príkazy (Statements) sa v oboch sekciách vykonávajú rovnako, tak predtým než popíšem implementáciu týchto sekcií, podrobnejšie rozpíšem samotné interpretovanie priraďovacích príkazov (Statements), priradení (Assigns) a vyhodnotenie viazaných premenných kvantifikátora (Quantification).

Vyhodnotenie viazaných premenných kvantifikátora (Quantification)

Kvantifikátor je udržiavaný v inštancii triedy `Quantification`.

obrázok 14 – UML diagram triedy Quantification



Na vyhodnotenie kvantifikovaných premenných používam primitívny algoritmus implementovaný v metóde `Evaluate()`. Keďže viazané(kvantifikované) premenné môžu byť len typu integer, tak algoritmus prejde všetky možné kombinácie hodnôt viazaných premenných a vyskúša či pre danú kombináciu je booleovský výraz pravdivý. Neprechádza sa celý rozsah typu integer, ale len rozsah určený dolnou a hornou hranicou, tie sú implicitne nastavené na hodnoty -50 a 50 ale je možne ich pre každý program nastaviť na želané hodnoty. Každá kombinácia pre ktorú booleovský výraz platí je uložená do poľa `bounded_values` (ako inštancia triedy `MemorySnapShot`). Pri vyhodnocovaní programových štruktúr obsahujúcich kvantifikátor sa vyhodnocuje ich vnútorná štruktúra postupne so všetkými predpočítanými hodnotami viazaných premenných. V booleovskej podmienke `quantified-statement-list`-u (implementovaný v triede QuantStatement) sa nesmú vyskytovať premmené, ktoré počas behu menia svoju hodnotu a preto stačí ich viazané premenné vyhodnotiť iba raz. V

`quantified-assignment` (trieda QuantAssign) síce táto podmienka nie je, avšak implicitne je nastavené aby sa viazané premenné vyhodnocovali tiež iba raz čím sa zvýši efektivita interpretera a nezníži sa výpočtová sila keďže premenné ktoré menia svoju hodnotu počas výkonu programu je možné pretlačiť do podmienok vnútorných priradení (assignments). Ak je však toto správanie pre daný UNITY program nevhodné dá sa potlačiť a viazané premenné sa budú vyhodnocovať zakaždým. Pri vyhodnocovaní `quantified expression` (trieda Quant_Expr) sa viazané premenné vyhodnocujú zakaždým.

Interpretovanie priradení (Assigns)

V UNITY sú možné nasledovné typy priradení:

- *`quantified-assignment`* implementované v triede `QuantAssign`
- *`enumerated-assignment`* toto priradenie je implementované v dvoch triedach podľa toho či jeho *`expr-list`* je *`simple-expr-list`* (trieda `EnumAssign`) alebo *`conditional-expr-list`* (trieda `CondEnumAssign`)

Triedy `CondEnumAssign`, `EnumAssign` a `QuantAssign` sú odvodené od abstraktnej triedy `Assign` a implementujú jej abstraktnú metódu `DoAssign()`, ktorá samotné priradenie vykonáva.

V `EnumAssign` sa zložené priradenie typu rozloží na jednotlivé jednoduché priradenia (klasické priradenia s jednou premennou na ľavej strane a jedným výrazom na pravej strane) a tie sa následne vykonajú v danom poradí. Pri `QuantAssign` sa najprv vyhodnotia booleovské výrazy, následne sa vyhodnotia pravé strany priradení pre ktoré je príslušný booleovský výraz pravdivý. Ak sa hodnoty vyhodnotených pravých strán rovnajú tak postupujeme ako v prípade `EnumAssign`, ak sa však nerovnajú vyhlási sa sémantická chyba. V `QuantAssign` sa ako už bolo vyššie uvedené vykonávajú vnútorné priradenia postupne so všetkými predpočítanými hodnotami viazaných premenných kvantifikátora.

Interpretovanie priraďovacích príkazov (Statements)

V UNITY sú možné nasledovné typy priraďovacích príkazov:

- *`assignment-statement`* implementovaný v triede `AssignStatement`

§ *quantified-statement-list* implementovaný v triede `QuantStatement`

Obe triedy sú odvodené od abstraktnej triedy `Statement` a implementujú jej abstraktnú metódu `DoExecute()`, ktorá priraďovacie príkazy vykonáva.

Priraďovací príkaz *assignment-statement* sa skladá z kolekcie priradení (assignments), ktoré sú v rámci daného príkazu vykonávané paralelne. Toto paralelne vykonávanie simulujem pomocou zdvojenej pamäte pre globále premenné (`symTabWrite` a `symTabRead`). `DoExecute()` metóda triedy `AssignStatement` postupne vyhodnocuje priradenia jedno za druhým. Na vyhodnocovanie výrazu z pravej strany priradenia sa používajú hodnoty premenných zo `symTabRead` (získané metódou `Resolve()` triedy `Variable`) a výsledná hodnota je priradená premennej z ľavej strany priradenia do pamäte `symTabWrite` (metódou `setValue()` triedy `Variable`). Po vykonaní všetkých priradení z priraďovacieho príkazu *assignment-statement* sa obsah pamäte `symTabWrite` prekopíruje do `symTabRead`, o čo sa stará nasledovný úsek kódu:

```
MemorySnapShot memcopy = unity_prog.makeMemSnapShot(MemoryType.Write);
memcopy.type = MemoryType.Read;
memcopy.LoadIn();
```

V triede `QuantStatement` metóda `DoExecute()` vykonáva vnútorné priraďovacie príkazy postupne so všetkými predpočítanými hodnotami viazaných premenných.

Sekcia initailly a sekcia assign

Sekcie initially a assign sa skladajú z kolekcie priraďovacích príkazov (statements) sú implementované v triede `InitiallySection` a `AssignSection` o ich interpretovanie sa stará metóda `DoExecute()`.

V initially sekcii sa vykonávajú priraďovacie príkazy podľa poradia ako sú zadané v programe, metóda DoExecute() je jednoduchá a vyzerá nasledovne:

```
public void DoExecute()
{
      Unity_Program unity_prog = Unity_Program.GetUnity_Program();
      foreach(Statement stm in this.statement_list)
      {
            stm.EvaluateQuants();
```

Pred samotným vykonaním priraďovacích príkazov sa vyhodnotia kvantifikátory ktoré môžu obsahovať.

Assign sekcia sa z definície UNITY vykonáva po krokoch do nekonečna. V každom kroku je vyhodnotený ľubovoľný statement (príkaz). Statements (príkazy) sú vyberané náhodne, ale musí byť zabezpečená pravidlo nestrannosti a to, že pri nekonečnom vykonávaní programu sa každý príkaz vykoná nekonečne veľa krát. Realizácia môjho interpretera je deterministická a príkazy sú vykonávané stále dookola, v takom poradí ako sú zadané v programe. Interpreter teda opakovane vykonáva cykly - postupnosť pevne definovaných krokov. Takáto realizácia zachováva pravidlo nestrannosti. Z definície UNITY sa stav programu vola fixed point (fixný bod) vtedy a len vtedy ak vykonanie ľubovoľného príkazu zanechá stav nezmenený, teda že všetky premenné budú obsahovať rovnakú hodnotu ako pred vykonaním príkazu. Z tejto definície vyplýva, že keď vykonám všetky príkazy assign sekcie čiže jeden cyklus a hodnota premenných sa nezmení, tak program dosiahol fixný bod. Po dosiahnutí fixného bodu ďalší výpočet programu nemení žiadne premenné a teda je jedno či výpočet programu pokračuje ďalej alebo zastaví. Toto využívam aj v mojej implementácií interpretera kde nekonečný beh programu nahradzujme konečným. Ak sa teda po vykonaní cyklu nezmenila hodnota žiadnej premennej môžem vykonávanie assign sekcie ukončiť. Metóda `DoExecute()` triedy `AssignSection` vyzerá nasledovne:

```
public void DoExecute()
{
      Unity_Program unity_prog = Unity_Program.GetUnity_Program();

      foreach(Statement stm in this.statement_list)
            stm.EvaluateQuants();

      while(!unity_prog.FixedPoint)
      {
            unity_prog.FixedPoint = true;
```

Pred samotným vykonaním priraďovacích príkazov sa vyhodnotia kvantifikátory ktoré môžu obsahovať. Následne vykonávam cyklus kým sa nedosiahne fixný bod. Premenná `FixedPoint` detekuje zmenu hodnôt globálnych premenných. Pred každým cyklom predpokladám že sa hodnota žiadnej premennej nezmení. Zmeniť sa môže iba priradením. Ako bolo skôr spomenuté, premennej sa hodnota priradzuje metódou `setValue()` triedy `Variable`. Ak sa priradzuje iná hodnota ako už premenná obsahuje tak program fixný bod ešte nedosiahol a je nevyhnutné vykonať ďalší cyklus.

ZÁVER

Cieľom moje diplomovej práce bolo vytvoriť interpreter programov napísaných v jazyku UNITY. Myslím si, že tento cieľ sa mi podarilo dosiahnuť. Vytvoril som jednoduchý interpreter spúšťaný z príkazového riadku, použiteľný vo všetkých operačných systémoch pre ktoré existuje implementácia behového prostredia pre programy systému .NET.

Na záver by som chcel poukázať, ktorým smerom by sa moja práca dala ešte rozšíriť. Môj interpreter implementuje iba základný jazyk UNITY, ktorý pracuje iba s dátovými typmi integer, boolean a jedno-viacrozmernými poľami týchto typov. Pre niektoré paralelne problémy to však nemusí stačiť a z tohto dôvodu existuje aj rozšírená verzia jazyka UNITY, ktorá pracuje s väčším množstvom dátových typov a umožňuje aj hierarchickú štruktúru UNITY programu. Myslím si že moja implementácia by mohla slúžiť ako dobrý základ pre vytvorenie interpretera, ktorý by programy rozšíreného UNITY jazyka dokázal vykonávať. Taktiež by bolo užitočné vytvoriť editor pre ľahšie používanie interpretera s funkciami ako: zvýrazňovanie syntaxe, vyznačenie pozície ohlásených chýb v zdrojovom súbore prípadne možnosťou krokovania.

Súčasťou mojej diplomovej práce je aj návod na používanie môjho interpretera a pár príkladov ktoré demonštrujú funkcionalitu interpretera. Tieto programy som používal aj pri testovaní a neobjavil som žiadne chyby a nedostatky v programe. Príklady sa nachádzajú na priloženom CD disku a návod je v tejto diplomovej práci v časti prílohy.

REFERENCIE

[1] CHANDY, K. Mani; MISRA Jayadev. Parallel Program Design: A Foundation. Menlo Park: Addison-Wesley, 1988. ISBN 0-201-05866-9. s. 3-79.

[2] KAČMÁŘ, Dalibor. Programujeme .NET aplikace: ve Visual Studiu .NET. Praha: Computer Press, 2001. ISBN 80-7226-569-5

[3] HYNEK, Jiří. Myslíme v jazyku C# : knihovna programátora. Praha: Grada Publishing, 2002. ISBN 80-247-0301-7

[4] AHO, Alfred V.; SETHI, Ravi; ULLMAN, Jeffrey D. Compilers: Principles, Techniques and Tools. Harlow: Addison-Wesley, 1985. ISBN: 0201101947.

[5] Wikipedia prispievatelia. Compiler-compiler [online]. Wikimedia Foundation, August 2002. aktualizované Marec 2006. [cit. 2006-04-10; 20:00 SEČ]. Dostupné na internete: <http://en.wikipedia.org/wiki/Compiler_compiler>.

[6] Wikipedia prispievatelia. List of compiler-compilers [online]. Wikimedia Foundation, August 2002. aktualizované Apríl 2006. [cit. 2006-04-10; 20:00 SEČ]. Dostupné na internete: < http://en.wikipedia.org/wiki/List_of_compiler-compilers>.

[7] Wikipedia prispievatelia. Yacc [online]. Wikimedia Foundation, August 2002. aktualizované Apríl 2006. [cit. 2006-04-10; 20:00 SEČ]. Dostupné na internete: < http://en.wikipedia.org/wiki/Yacc>.

[8] BERK, Elliot. JLex:A lexical analyzer generator for Java(TM) [online]. Department of Computer Science, Princeton University, Máj 1997. aktualizované September 2000. [cit. 2006-04-10; 20:00 SEČ]. Dostupné na internete: <http://www.cs.princeton.edu/~appel/modern/java/JLex/current/manual.html>.

[9] JOHNSON, Stephen C. Yacc: Yet Another Compiler-Compiler [online]. AT&T Bell Laboratories. [cit. 2006-04-10; 20:00 SEČ]. Dostupné na internete: <http://dinosaur.compilertools.net/yacc/index.html>

[10] SCHREINER, Axel-Tobias ; KÜHL, Bernd. Jay: a yacc for Java [online]. 1997. [cit. 2006-04-10; 20:00 SEČ]. Dostupné na internete: < http://www.informatik.uni-osnabrueck.de/alumni/bernd/jay/>

[11] GRANICZ, Adam; ZIMMERMAN, Daniel M.; HICKEY, Jason. Rewriting UNITY [online]. California Institute of Technology, 2002. [cit. 2006-04-10; 20:00 SEČ].Dostupné na internete: < http://mojave.caltech.edu/papers/rta03.pdf >

[12] Grimsmo, Nils. Toy UNITY compiler [online]. október 2005. [cit. 2006-04-10; 20:00 SEČ]. Dostupné na internete: < http://www.idi.ntnu.no/~nilsgri/unity/ >

PRÍLOHY

Manuál k UNITY interpreteru

Interpreter je navrhnutý ako nástroj spúšťaný z príkazového riadku, ktorý ako parameter dostane adresu k textovému súboru, ktorý obsahuje UNITY program.

```
> UnityInterpreter input1.unity
```

Vstupom je teda jednoduchý textový súbor. Na niektoré znaky z UNITY definície, sa kvôli jednoduchosti používa sekvencia znakov, čo ilustruje nasledujúca tabuľka:

znak v UNITY interpreter	znak UNITY definície	
!=	**≠**	
>=	**≥**	
<=	**≤**	
=	**=**	ekvivalencia
:=	**=**	priradenie v initally section
[ ]	□	separátor príkazov
||	**‖**	separátor priradení
<<	**<**	začiatok kvantifikovanej sekcie*
>>	**>**	koniec kvantifikovanej sekcie*
and	logický **AND**	
or	logický **OR**	
not	logický **NOT**	

* kvantifikované sekcie: `quantified-statement-list` `quantified-assignment` `quantified-expression`

UNITY program by mal obsahovať section declare, section initially a section assign. V sekcii initailly je podľa definície znakom priradenia = kôli zjednodušeniu interpretera je v mojej implementácií potrebné použiť znak := tak ako v assign sekcii, navyše sa kód stáva ľahšie čitateľným keďže nemožno znak priradenia

zameniť za znak ekvivalencie. Sekcia always nie je v interpreteri implementovaná, keďže nie je pre UNITY program nutne potrebná.

Declare section používa na deklarovanie premenných syntax z jazyka Pascal. Interpreter ponúka dátové typy: integer, boolean a jedno/viacrozmerné polia týchto typov. Jednotlivé deklarácie sú oddelené bodkočiarkou **;**.

Príklady deklarácií:

```
declare
      a: boolean;
      N,k:integer;
      A: array[0..N] of integer;
      U: array[0..3,0..4] of boolean;
      K: array[0..4] of array[0..5] of integer
initially
...
```

Zdrojový súbor môže obsahovať aj komentáre tie sú pri lexikálnej analýze ignorované a tak nemajú vplyv na chod programu. Jednoriadkový komentár nasleduje po znakoch // a teda všetky znaky až po koniec daného riadka sú odignorované. Taktiež sa odignorujú všetky znaky medzi /* a */ vrátene týchto znakov čo sa dá využiť na komentár prechádzajúci cez viac riadkov.

Interpreter obsahuje nasledovné preddefinované funkcie:

funkcia	**návratový typ**	**popis**
minimum		
Min()	integer	vráti nulový prvok funkcie minimum
Min(integer a0, integer a1)	integer	vráti hodnotu menšieho z dvoch čísel
Min(integer a0, integer a1, integer a2)	integer	vráti hodnotu najmenšieho z troch čísel
Min(integer[] arr)	integer	vráti hodnotu najmenšieho prvku poľa
maximum		
Max()	integer	vráti nulový prvok funkcie maximum
Max(integer a0, integer a1)	integer	vráti hodnotu väčšieho z dvoch čísel
Max(integer a0, integer a1, integer a2)	integer	vráti hodnotu najväčšieho z troch čísel
Max(integer[] arr)	integer	vráti hodnotu najväčšieho prvku poľa
párnosť		
Even(integer a)	boolean	vráti true ak je číslo párne

Odd(integer a)	boolean	vráti true ak je číslo nepárne
náhodná hodnota		
Random()	integer	vráti náhodnú hodnotu v rozsahu typu integer
Random(integer maxValue)	integer	vráti nezápornú náhodnú hodnotu do veľkosti maxValue
Random(integer minValue, integer maxValue)	integer	vráti náhodnú hodnotu v danom rozsahu

Interpreter umožňuje používateľovi dodefinovať vlastné funkcie, cez sekciu import. Import sekcia musí byť zadeklarovaná pred samotným UNITY programom, čiže ešte pred kľúčovým slovom **`Program`**. Štruktúra import sekcie je nasledovná

```
import-section  →   import dll_list
dll_list        →   dll_list ; string
dll_list        →   string
```

za kľúčovým slovom import nasledujú bodkočiarkami oddelené znakové reťazce (znakový reťazec je ľubovoľná sekvencia znakov okrem znaku nového riadku uzavretá medzi dvojitými uvodzovkami "), definujúce umiestnenie .NET assembly, ktorá obsahuje implementáciu užívateľských funkcií. Je možné použiť absolútnu i relatívnu cestu k assembly.

Príklad import sekcie:

```
import "../../dlltest/customfuncVB.dll"; "mathfunc.dll"
Program ...
```

.NET assebly definujúca funkcie môže byť napísaná v ľubovoľnom .NET jazyku funkcie musia byť verejnými statickými metódami nejakej statickej triedy X s návratovou hodnotou typu System.Int32 alebo System.Boolean systému CTS (Common Type System). Funkcie sú potom v UNITY programe dostupné pomocou svojich mien. Ak existuje predefinovaná funkcia alebo už skôr importovaná funkcia s rovnakým menom treba pred meno uviesť aj názov triedy, v ktorej je deklarovaná (názov triedy je od názvu metódy oddelený bodkou).

Príklad zdrojových súborov assembly definujúcich užívateľské funkcie:

- V jazyku C#

súbor customfuncCS.cs

```
using System;

  public class cmt
  {
      public static int custTime( int a , int custNum)
      {
        int retval = ((int)Math.Ceiling((double)a
                        /(double)custNum)) * custNum;
        return (retval < custNum) ? custNum : retval;
      }
  }
```

- V jazyku VB.NET (Visual Basic .NET)

  súbor customfuncVB.vb

```
Imports System

  Public Class cmt

    Public Shared Function custTime(ByVal a As Integer, ByVal
custNum As Integer) As Integer
      Dim retval As Double = (Math.Ceiling(a / custNum)) *
custNum
      If (retval < custNum) Then
        Return custNum
      Else
        Return retval
      End If
    End Function

  End Class
```

Zdrojové súbory treba následne skompilovať do .NET assembly. V .NET Frameworku možno jednoducho využiť riadkové (púšťané z príkazového riadku) kompilátory pre príslušné jazyky, napríklad:

- pre jazyk C#:

  `> csc /nologo /t:library /out:customfuncCS.dll customfuncCS.cs`

- pre jazyk VB.NET:

  `> vbc /nologo /t:library /out:customfuncVB.dll customfuncVB.vb`

vytvorenú assembly (**`customfuncCS.dll`** alebo **`customfuncVB.dll`**) možno následne importovať a do UNITY programu a funkciu `custTime()` využívať:

```
import "../../dlltest/customfuncVB.dll";
Program CommMeatTime
      declare r:integer;
      initially r := 0
      assign
            r:= cmt.custTime(r,11) []
            r:=custTime(r,5) []
            r:=custTime(r,3)
end
```

Užívateľsky definované funkcie možno využívať aj v kvantifikovanom výraze (Quantified Expression) ak splňuje nasledovné podmienky:

1. funkcia definuje binárny asociatívny a komutatívny operátor a teda musí mať práve dva parametre. Typ parametrov i návratovej hodnoty musí byť rovnaký.
2. v danej triede musí existovať aj preťažená (overloaded) metóda ktorá nemá žiadne parametre. Tá vracia jednotkový prvok funkcie, ktorému sa kvantifikovaný výraz rovná ak neexistuje žiadna inštancia pre danú kvantifikáciu.

Napríklad:

funkcia definovaná v jazyku C# nasledovne:

```
public static int Add(int a0, int a1)
{
      return a0+a1;
}
public static int Add()
{
      return 0;
}
```

môže byť následne využitá v kvantifikovanom výraze

```
<< Add i : ( 0<=i<=N )and (A[i]≤A[j]) :: 1 >>
```

pričom má tú istú funkcionalitu ako kvantifikovaný výraz

```
<< + i : ( 0<=i<=N )and (A[i]≤A[j]) :: 1 >>
```

Za uvedením cesty k textovému súboru, ktorý obsahuje UNITY program môžeme definovať ako ďalšie parametre modifikátory, ktoré menia správanie interpretera. Možno použiť nasledujúce modifikátory:

-i, **-initially** umožňuje zadefinovať a nastaviť počiatočná hodnoty premenných typu integer a boolean. Takto zadefinované premenné netreba následne definovať v sekcii declare samotného UNITY programu. Využíva sa ak chceme spustiť ten istý UNITY program s rôznymi parametrami, pričom nemusimé modifikovať zdrojový súbor s kódom UNITY programu

```
> UnityInterpreter input.unity -i N=15 b=true c=8
> UnityInterpreter input.unity -initially N=10 c=8
```

-m, **-memory** vypíše obsah pamäte po každom príkaze (statement-e)

-b, **-break** slúži na krokovanie. Po vykonaní každého príkazu (statement-u) čaká na vstup. Po zadaní run sa krokovanie ukončí, ak sa zadá exit ukončí sa interpretovanie. Logické je používať krokovanie, len so zapnutým vypisovaním obsahu pamäte.

```
> UnityInterpreter input.unity –b -m
```

-e, **-evalbounded** potlačuje implicitné správanie interpretera, keď sa viazané premenné `quanitified-assigment`–u vyhodnocujú iba raz a teda sú vyhodnocované zakaždým keď sa priradenie vyhodnocuje.

-min určuje spodnú hranicu rozsahu, pre ktorý sa vyhodnocujú viazané premenné kvantifikátora

-max určuje hornú hranicu rozsahu, pre ktorý sa vyhodnocujú viazané premenné kvantifikátora

```
> UnityInterpreter input.unity –e –min -20 –max 80
```

Elektronická príloha

K práci je priložený CD disk s nasledujúcim obsahom podľa adresárov:

- **build 1.0** - skompilovaná verzia interpretera
- **ethesis** - elektronická verzia diplomovej práce
- **input** – vstupné súbory pre nástroje na automatické generovanie analyzátorov
- **sourcecode** - zdrojové súbory interpretera
- **test** - príklady, ktoré sú uvedené v diplomovej práci, plus Floyd-Warshall algoritmus na nájdenie najkratšej cesty medzi dvoma bodmi na grafe, Ranksort a algoritmus na nájdenie spoločného času stretnutia troch procesov.