VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FAKULTA INFORMAČNÝCH TECHNOLÓGIÍ

Dokumentácia projektu z predmetov IFJ a IAL

Implementácia prekladaču imperatívneho jazyka IFJ18

Tím 124, varianta II 2018/2019

xmicha70 - Ladislav Michalik - 25%
xsalon00 - Marek Saloň - 25%
xsterb12 - Maroš Štěrba - 25%
xsenca00 - Jakub Senčák - 25%

5.12.2018

1 Úvod

Táto dokumentácia bola vytváraná počas vývoja prekladača imperatívneho jazyka IFJ18 a slúži ako sprievodca skrz detailný popis konštrukcie a implementácie prekladača. Dokumentácia obsahuje jednotlivé postupy a prístupy nášho riešenia, ako aj naše formálne podklady k vytvoreniu jednotlivých častí, založených na teoretických základoch z predmetov IFJ a IAL.

2 Štruktúra

(1) Lexikálny analyzátor

(2) Syntaktický analyzátor

(3) Precedentný syntaktický analyzátor

(4) Sémantický analyzátor

(5) Generátor kódu IFJcode18

3 Rozbor jednotlivých častí

3.1 Lexikálny analyzátor

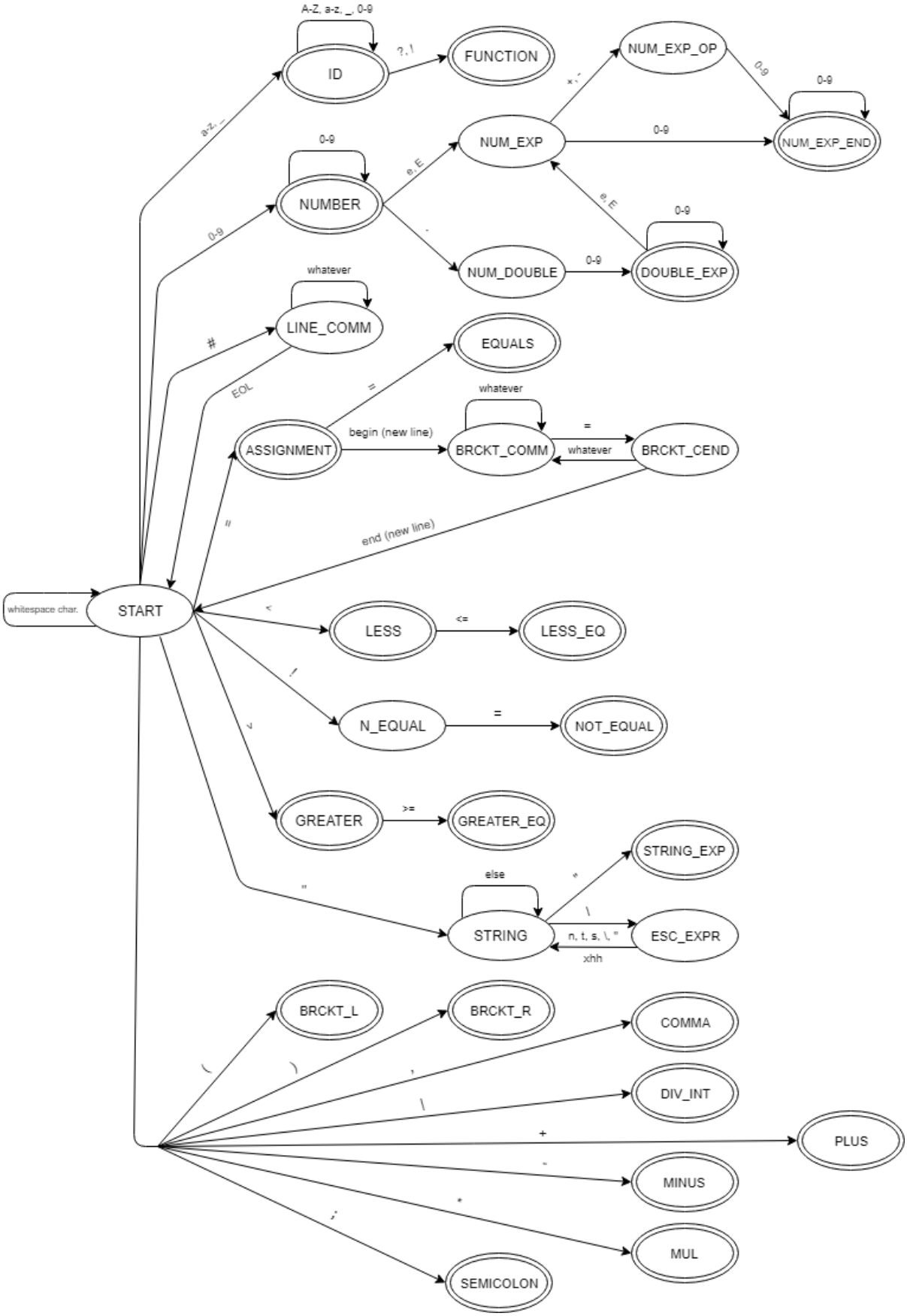
Lexikálny analyzátor, alias skener, je časť prekladača slúžiaca pre identifikáciu jazykových jednotiek tzv. lexémov a následné odosielanie tokenov pre syntaktickú analýzu.

Skener pracuje na základe jednoduchej implementácie deterministického konečného automatu. Sekundárna úloha skeneru je detekcia lexikálnych chýb (kód chyby: 1).

Skener načíta zo zdrojového súboru postupnosť znakov a na základe konečného automatu rozhodne, či sa jedná o konkrétny typ lexému daného jazyka alebo chybu (getToken()). Zarovno s tým z prekladu odstraňuje komentáre. Pri vyhodnotení korektného lexému, zabalí lexém s voliteľným pomocným údajom do štruktúry reprezentujúcej token (makeToken()).

Skener obsahuje aj pomocné funkcie na prácu s dynamickým reťazcom napr. addNextChar() a addEscSeq(), ktorá k danému reťazcu pripojí znak alebo “escape” sekvenciu a prídavné funkcie slúžiace na vyčistenie buffer-u reťazca.

3.1 Lexikálny analyzátor – DKA implementácia



3.2 Syntaktický analyzátor

Syntaktický analyzátor (parser) je nosným prvkom prekladača a má za úlohu simulovať priechod derivačným stromom.

V našej implementácii využívame algoritmus postupného rekurzívneho zostupu, ktorý sme implementovali na základe pravidiel daných LL-gramatikou (viz. 3.2 LL-tabuľka). Komunikuje s ostatnými časťami.

Jeho vstupom je výstup lexikálneho analyzátoru. Predávanie lexému prebieha pomocou globálnej štruktúry token, ktorá obsahuje typ a hodnotu. Podľa typu, ktorý dostane od skeneru, zostúpi v derivačnom strome nižšie, pričom overuje syntaktickú správnosť príkazov. Pri definícií funkcií a premenných volá funkcie z tabuľky symbolov (symtable.h), ktorá rieši časť sémantických chýb. Ak parser narazí na výraz, predá riadenie precedentnej analýze. Po spracovaní výrazu sa mu riadenie znovu vráti.

Z dôvodu možnej definície až po použití ešte nedefinovanej funkcie v rámci inej funkcie, sme sa rozhodli implementovať dva prechody. V prvom prechode si ukladáme záznamy o definíciách funkcii, spracovanie názvu a uloženie počtu parametrov do tabuľky symbolov. V prvom prechode sa chyby nekontrolujú.

V druhom prechode sa vykoná kompletný prechod derivačným stromom. Skontrolujú sa syntaktické a sémantické chyby.

Parser preberá chybové výstupy ostatných častí prekladača a je koncovým interpretom predávania návratovej hodnoty, v prípade úspechu 0 EXIT_SUCCESS, inak chybu podľa jej typu.

3.2 Syntaktický analyzátor – LL-gramatika

```
<type> -> int
<type> -> nil
<type> -> float
<type> -> string
<if-cmd> -> if <expression> then EOL <def-cmds> <elsif-cmd> <else-cmd> end
          EOL
<elsif-cmd> -> elsif <expression> then EOL <def-cmds> <elsif-cmd>
<elsif-cmd> -> €
<else-cmd> -> else EOL <def-cmds>
<else-cmd> -> €
<while-cmd> -> while <expression> do EOL <def-cmds> end EOL
<var-cmd> -> id = <expression> EOL
<var-cmd> -> id = <call-fun> EOL
<call-fun> -> id fun-name <brckt-l> <params> <brckt-r>
<def-fun> -> def id <brckt-l> <params> <brckt-r> EOL <def-cmds> end EOL
<brckt-l> -> €
<brckt-l> -> (
<brckt-r> -> €
<brckt-r> -> )
<params> -> €
<params> -> id
<params> -> id , <params>
<cmd> -> €
<cmd> -> EOL
<cmd> -> <if-cmd> <elsif-cmd> <else-cmd> end EOL
<cmd> -> <while-cmd> end EOL
<cmd> -> <var-cmd>
<cmd> -> <call-fun>
<cmd> -> <def-fun>
<cmd> -> <inputs>
<cmd> -> <inputi>
<cmd> -> <inputf>
<cmd> -> <length>
<cmd> -> <substr>
<cmd> -> <print>
<cmd> -> <ord>
<cmd> -> <chr>
<cmd> -> <expression>
<inputs> -> inputs <brckt-l> <brckt-r>
<inputi> -> inputi <brckt-l> <brckt-r>
<inputf> -> inputf <brckt-l> <brckt-r>
<length> -> length <brckt-l> <expression> <brckt-r>
<substr> -> substr <brckt-l> <expression>, <expression>, <expression>
          <brckt-r>
<length> -> length <brckt-l> <expression> <brckt-r>
<print-param> -> <expression> ,
<print-param> -> <expression>
<print-param> -> €
<print-params> -> <print-param> <print-params>
<print-params> -> €
<print> -> print <brckt-l> <print-params> <brckt-r> EOL
<ord> -> ord <brckt-l> <expression> , <expression> <brckt-r>
<chr> -> chr <brckt-l> <expression> <brckt-r>
<def-cmds> -> <cmd> <def-cmds>
<def-cmds> -> €
<program> -> <def-cmds>
```

3.3 Precedentný syntaktický analyzátor

Správnosť syntaxe výrazov v zdrojovom kóde overuje precedentný analyzátor, ktorého syntaktická analýza je založená na postupu zdola nahor. Analýza implementovaná pomocou hlavnej funkcie int precedenceAnalysis(T_struct* previousToken) pracuje so zásobníkom a precedenčnou tabuľkou.

Zásobník tStack PAStack implementovaný jednosmerným zoznamom, ktorého položkami sú ukazatele tSItemPtr na štruktúru tSItem obsahujúca prvky: štruktúra T_struct symbol predstavujúca token, premenná T_type reduceType obsahujúca typ tokenu po prevedení redukcie, prízank term značiaci terminál, príznak handle značiaci začiatok pravej strany pravidla.

Precedentná tabuľka precedence_table pre výber operácie so zásobníkom je implementovaná ako dvojrozmerné pole charov. Zohľadňuje prioritu, asociativitu matematických (aritmetických a logických) operácií, identifikátory, zátvorky a ukončovací reťazec (EOL, THEN, DO resp. EOF). Symboly v prvom ľavom stĺpci tabuľky predstavujú symboly na vrcholu zásobníku, symboly v záhlaví tabuľky predstavujú aktuálne čítané tokeny v reťazci.

Analyzátor číta tokeny od lexikálneho analyzátoru a porovnáva ich s prvými terminálmi od vrcholu zásobníku. Na základe precedentnej tabuľky vyberie vo funkcii void selectRule(int* psa_state, tStack* PAStack, tSItemPtr* firstTerm, int* var_cnt, bool* firstExp, bool* alone) operáciu, ktorá sa vykoná.

Pri operácii ‘=‘ sa token uloží pomocou funckie void pushNoChange(tStack* PAStack) na zásobník, navyše pri operácii ‘<‘ sa zaznamenáva začiatok “handle“ vo funckii void pushAndChange(tStack* PAStack, tSItemPtr* firstTerm, int* var_cnt, bool* alone). Pri operácii ‘>‘ sa vo funkcii void reduce (int* psa_state, tStack* PAStack, bool* firstExp, bool* alone) redukujú položky na zásobníku a ukladajú sa do zoznamu tDLList rightSideList. Následne sa vo funkcii void derivate (int* psa_state, int i, tDLList* L, tStack* PAStack, bool* firstExp, bool* alone) kontroluje výraz v zozname s pravou stranou pravidiel.

Pre prípad samostatného výrazu na riadku začínajúceho identifikátorom, kedy nie je zrejmé, či sa jedná o výraz alebo priradenie, sú implementované obdobné funckie pracujúce s predchádzajúcim tokenom T_struct previousToken predaným ako argument hlavnej funkcie.

3.3 Precedentná tabuľka

	+	-	*	/	(	)	ID	< <= > >=	= !=	$
+	>	>	<	<	<	>	<	>	>	>
-	>	>	<	<	<	>	<	>	>	>
*	>	>	>	>	<	>	<	>	>	>
/	>	>	>	>	<	>	<	>	>	>
(	<	<	<	<	<	=	<	<	<	
)	>	>	>	>		>		>	>	>
ID	>	>	>	>		>		>	>	>
< <= > >=	<	<	<	<	<	>	<		>	>
= !=	<	<	<	<	<	>	<	<		>
$	<	<	<	<	<		<	<	<	

3.4 Sémantická analýza - tabuľka symbolov

Tabuľka symbolov je implementovaná pomocou hash-ovacej tabuľky. Náš prekladač používa jednu globálnu tabuľku symbolov pre definíciu funkcie. Každá funkcia má potom vlastnú (lokálnu) tabuľku symbolov, v ktorej ukladá kópiu parametrov a ďalšie lokálne premenné. Hlavnou štruktúrou globálnej tabuľky symbolov je štruktúra, do ktorej sa ukladajú údaje o jednotlivých funkciách.

Údaje typu:

- názov
- informácie o parametroch viazaných v lineárnom zozname, pretože ich počet nie je vopred známy
- konečný počet parametrov
- odkaz na lokálnu tabuľku symbolov danej funkcie

Do lokálnej tabuľky symbolov každej funkcie sa automaticky kopírujú parametre z globálnej tabuľky symbolov a ukladajú sa ako lokálne premenné danej funkcie.

Lokálna tabuľka symbolov obsahuje informácie o premenných danej funkcie a ich dátových typoch.

Kontrola sémantických chýb je implementovaná priamo v rámci syntaktickej a precedentnej analýzy.

3.5 Generovanie kódu IFJcode18

Generátor kódu bol implementovaný v tvare funkcií nachádzajúcich sa v súbore generator.c. Syntaktický analyzátor volá jednotlivé funkcie generátoru počas syntaktickej analýzy a generuje kód do príslušného buffer-u a následne tlačí znaky na štandardný výstup. Dynamický buffer bol vytvorení pre zabránenie opätovných definícií.

4 Práca v tíme

Vývoj prekladača postupoval jednotlivými fázami. V prvej fázy sme sa dohodli na spoločnom komunikačnom kanále ako aj na používanom verzovacom systéme. Na komunikáciu sme používali osobné stretnutia podľa potreby, ako verzovací systém nám poslúžil Git, resp. GitHub. Stretnutia sme si plánovali každý týždeň v závislosti od potreby a času.

5 Záver

Projekt nám dal všetkým mnoho skúseností. Získali sme skúsenosti ohľadom tímovej práce, riešení problémov, komunikácie a time managemente. Napriek všetkým prekážkam, sme projekt dotiahli do záverečného stavu.