VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Dokumentácia k projektu z predmetov IFJ a IAL

Implementácia prekladača imperatívneho jazyka IFJ23

Tým xtothf00, varianta vv-BVS

Frederik Tóth	`xtothf00`	25%
Tomáš Široký	`xsirok10`	25%
Michael Babušík	`xbabus01`	25%
Andrej Mikuš	`xmikus19`	25%

2023/2024

Obsah

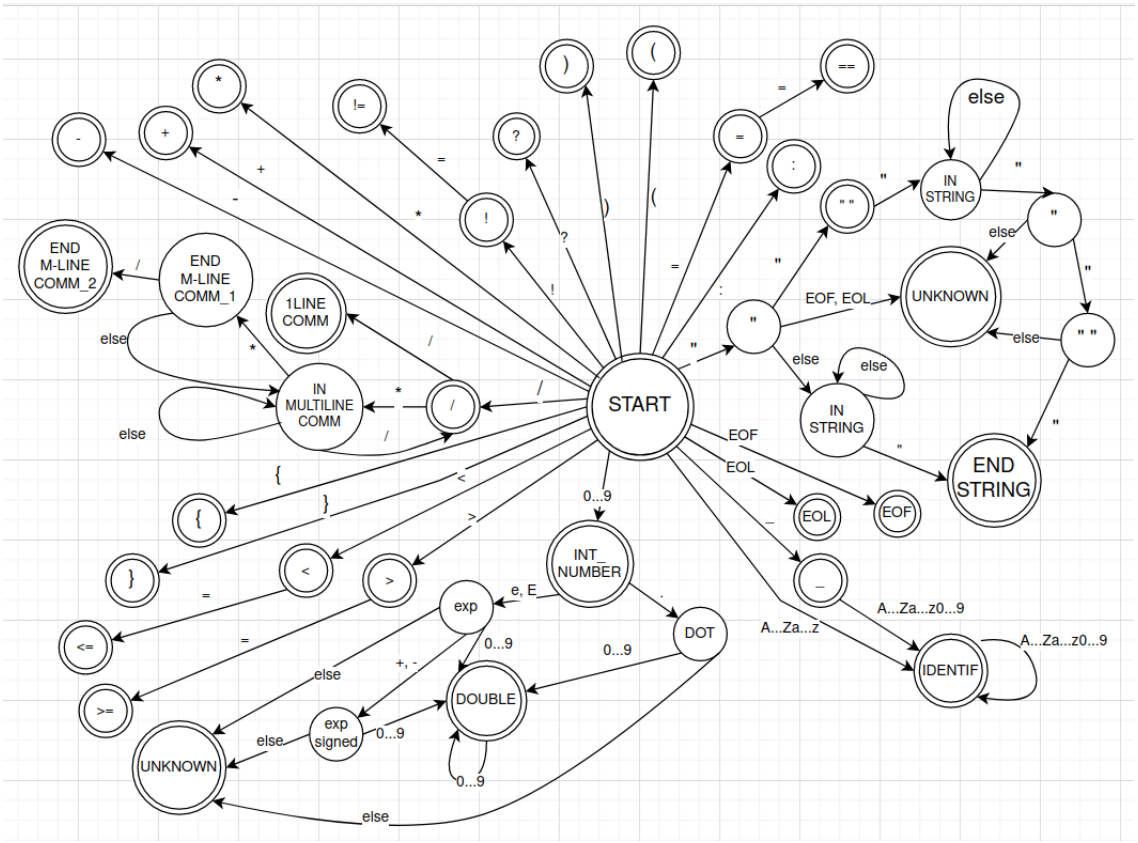
1 Úvod

Cieľom projektu bolo vytvoriť program v jazyku C, ktorý načíta zdrojový kód zapísaný v zdrojovom jazyku IFJ2023 a preloží ho do cieľového jazyka IFJcode23.

2 Štruktúra prekladača

2.1 Lexikálna analýza

Podstatou lexikálnej analýzy je rozpoznávanie a klasifikácia lexémov v zdrojovom kóde. Je implementovaná pomocou deterministického konečného automatu v module `scanner.c`. Kľúčovou funkciou je funkcia `getToken()`,ktorá vráti štruktúru Token, ktorá má prvky *symbol* (reťazec predstavujúci token) a *ID* (atribút tokenu). Pomocnou funkciou k funkcii `getToken()` je funkcia `stringanoff()`, ktorá je zvlášť vyhradená na spracovávanie reťazcov. Obe funkcie používajú konštrukciu *switch* jazka C. Súčasťou lexikálneho analyzátora je zásobníkový automat určený na spracovanie vnorených komentárov.



Obrázek 1: Automat lexikálnej analýzy

2.2 Syntaktická analýza

Syntaktické chyby sú rozpoznávané takisto v module `scanner.c`. Syntaktická analýza je založená na LL-gramatike a LL-tabuľke, ktoré sú implementované rekurzívnym zostupom. Nezrovnalosti v praktickom a teoretickom pohľade na implementáciu boli spôsobené tým, že jeden z implementujúcich mal na to „pohľad architekta“ a druhý „pohľad murára“.

2.3 Precedenčná syntaktická analýza

Funkciu *expression()* definovaná v module `expression.c` je zavolaná parserom, následne sa porovná vstupný token s vrcholom zásobníku. Funkcia *reduce()* na základe nasledujúceho tokenu a vhodného pravidla vytvorí uzol stromu.

```
<program> = {statement}
<statement> = <mod-var> | <nomod-var> | <assignment> | <func-decl> | <func-call> | <if-else> |
<while> | <return>

(assigment)
assignment = <id>, “=”, <expr> | <func-call>

(keywords)
<keywords> = “else” | “func” | “if” | “let” | “nil” | “return” | “var” | “while” | “string” | “int” | “double”

(functions)
<func-decl> = <func-head>, <func-body>
<func-head> = “func”, <id>, “(”, <param-list-def>, “)”, [“→”, <type>]
<param-list-def> = <id>, <id>, “:”, <type>, {“,”, id, id, “:”, <type>}
<func-body> = “{”, {<statement>}, “}”

(func-call)
<func-call> = <id>, “(”, <param-list-call>, “)”
<param-list-call> = {<param-name> | <param>}
<param-name> = <id>, “:”, <expr>, [“,”]
<param-underscore> = <expr>, [“,”]

(return)
<return> = “return”, [expr]

(if else statement)
<if-else> ::= “if”, <if-expr>, “{”, {statement}, “}”, else, “{”, {statement}, “}”
<if-expr> ::= <expr_bool>

(while)
while = “while”, <if-expr>, “{”, {statement}, “}”

(expression)

<expr_int> ::= (<digit>, (<bi_opernad>), <digit>), ((<digit>, (<bi_opernad>), <digit>))*
<expr_str> ::= (<string>, (“+”), <string>), ((“+”), <string>)*
<expr_bool> ::= ((<digit>, (<bool_opernad>), <digit>), ((<digit>, (<bool_opernad>), <digit>))*)
<expr> ::= [“(”] <expr1> [“)”]
<expr1> ::= <expr2> | <expr1> “*” <expr> | <expr1> “/” <expr>
<expr2> ::= <expr3> | <expr2> “+” <expr1> | <expr2> “-” <expr1>
<expr3> ::= <expr4> | <expr2> “==” <expr3> | <expr2> “!=” <expr3> | <expr2> “<” <expr3> | <expr2> “>” <expr3> | <expr2> “<=” <expr3> | <expr2> “>=” <expr3>
<expr4> ::= <term> | <term> “??”

(inicializace expression)
<init-expr> ::= <id> [":" <type>] "=" <expr> | "nil"

(variables)
<mod-var> ::= "var" <init-expr>
<nomod-var> ::= "let" <init-expr>
let a → let a = “nil” (optimalizacie)

(operandy)
<bool_operad> ::= “==” | “!=” | “<” | “>” | “<=” | “>=”
<un_operand> ::= “++” | “- -”
<bi_operand> ::= “=” [“=”] | “+” | “-” | “/” | “*” |
<term> ::= <integer> | <double> | <string> | "nil" | <id>
<id> ::= (<letter>)+(<letter> | <digit> | “_”)*
<type> ::= "Int" [“?”] | "Double" [“?”] | "String" ["?"]
<double> ::= <integer> "." <integer> | <integer> "e" <integer>
| <integer> "E" <integer> | <integer> "." <integer> "e" <integer>
| <integer> "." <integer> "E" <integer>
<integer> ::= <digit>+
<string> ::= '"'<printable-ascii_char>'"'
<multiple_line_string> ::= """\n hocičo \n"""
<letter> ::= "a" | ... | "z" | "A" | ... | "Z"
<digit> ::= "0" | "1" | ... | "9"
<printable-ascii_char> ::= ASCII character 32-126
<ascii-char> ::= any ASCII character
```

Obrázek 2: Gramatika precedenčnej syntaktickej analýzy

	+	-	*	/	(	i	)	$	??
+	M	M	L	L	L	L	M	M	M
-	M	M	L	L	L	L	M	M	M
*	M	M	M	M	L	L	M	M	M
/	M	M	M	M	L	L	M	M	M
(	L	L	L	L	L	L	Q	R	L
i	M	M	M	M	R	R	M	M	M
)	M	M	M	M	R	R	M	M	M
$	L	L	L	L	L	L	R	R	L
??	L	L	L	L	L	L	R	M	L

Obrázek 3: Precedenčná tabuľka

2.4 Dynamické reťazce

Modul `dynamic.c` máme na uchovávanie záznamu o dynamicky alokovaných prostriedkoch a slúži na prípady, v ktorých nastane chyba a program potrebuje predčasne ukončiť – je potreba okamžite uvoľniť alokované prostriedky.

2.5 Tabuľka symbolov

Tabuľka symbolov bola implementovaná ako výškovo vyvážený binárny vyhľadávací strom v súbore `symtable.c`. Slúži na predanie názvov funkcií a identifikátorov. Po každom vkladaní kontrolujeme vyváženosť stromu a pokým nie je vyvážený, robíme pravú alebo ľavú rotáciu.

2.6 Generovanie kódu

Ukazateľ na AVL strom, ktorý obsahuje blok príkazov sa rozloží na jednotlivé príkazy a tie sa potom posielajú do funkcie *generator()* a spracovávajú sa po jednom.

3 Testovanie

Prekladač sme testovali pomocou shell skriptu a zdrojových kódov v jazyku Swift. Testy boli vytvorené iným tímom, ale doplnili sme ich aj o vlastné testy.

4 Práca v tíme

S implementáciou projektu sme mali v tíme menšie problémy, nakoľko niektoré osoby v tíme riešili dlhodobé osobné/rodinné dôvody. Z týchto dôvodov bola naša komunikácia prevažne vo virtuálnej forme a osobne sme sa stretávali len výnimočne.
Komunikovali sme cez platformy Discord a Messenger a pri implementácii sme využívali verzovací systém Git.

4.1 Rozdelenie práce v tíme

Vzhľadom na približne rovnako veľké úsilie venované projektu, sme sa dohodli na rovnomernom rozdelení bodov. Nižšie je uvedené rozdelenie práce na projekte:

Frederik Tóth – vedenie tímu, implementácia lexikálnej analýzy, syntaktickej analýzy
Tomáš Široký – implementácia dynamických reťazcov, chybové stavy, generovanie kódu
Michael Babušík – implementácia tabuľky symbolov, spracovanie výrazov, zásobník stromu a tokenov
Andrej Mikuš – implementácia lexikálnej analýzy, dokumentácia, testy

5 Záver

Vypracovaním projektu sme bližšie pochopili, ako funguje prekladač a aké algoritmy sa pri jeho tvorbe využívajú. Práca na projekte bola pre nás zaujímavá aj tým, že to bola naša prvá skúsenosť s tímovou spoluprácou.