Implementácia prekladaču imperatívneho jazyka IFJ20

Dokumentácia k projektu

Tím 141, varianta I

Vedúci tímu:

Fabo Adam (xfaboa00) 25%

Ostatní členovia tímu:

Groma Albert (xgroma00) 25%

Gabriš Stanislav (xgabri18) 25%

Országh Roman (xorsza01) 25%

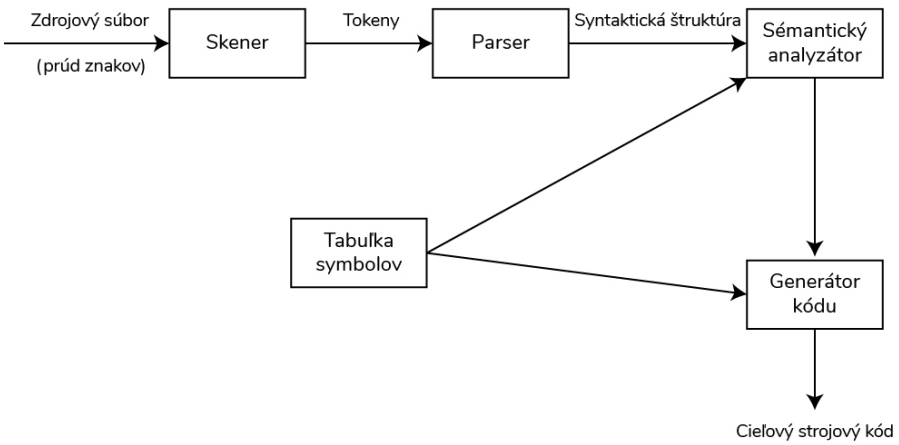
4. December 2020 Rozšírenia: -

Obsah

1 Úvod

V prekladači využívame priame generovanie kódu, t.j. generácia kódu prebehne vždy, keď má program dostatok informácii na to, aby vygeneroval inštrukcie. Lenže nie vždy program po spracovaní tokenu vie, aký kód má generovať, preto v kóde využívame zásobníky.

Štruktúra nášho kompilátora je nasledovná:



Obr. 1: Schéma prekladača

2 Lexikálna analýza

Lexikálny analyzátor dostane od parseru prázdny token, ktorý sa snaží naplniť a vrátiť. Skúma vstup podľa navrhnutého konečného automatu. Ak narazí na znak ktorý už podľa KA nemôže spracovať, vráti ho pomocou funkcie ungetc() naspäť. Aj v niektorých iných prípadoch pužívame funkciu ungetc(), ako napríklad pri prechode zo stavu 1 do stavu štart v konečnom automate (viz obr. 5 a obr. 6 v prílohe).
Následne podľa pravidiel v KA buď vracia naplnený token alebo vracia LEXICAL_ERROR, ak sa nenachádza v konečnom stave. Token drží v type základnú informáciu o lexéme a v data dodatočné informácie. Lexikálny analyzátor nemá prístup k tabuľke symbolov. Keďže lexikálny analyzátor nepozná momentálny kontext nevedel by (v našom riešení) správne pracovať s danou tabuľkou.
Pri blokovom komentáry sme sa rozhodli, ak sa vyskytuje, posielať aj informáciu o NEWLINE (podľa nasledujúcich znakov), inak po prečítaní komentáru pokračujeme v lexikálnej analýze od znova (podľa KA).

3 Parser

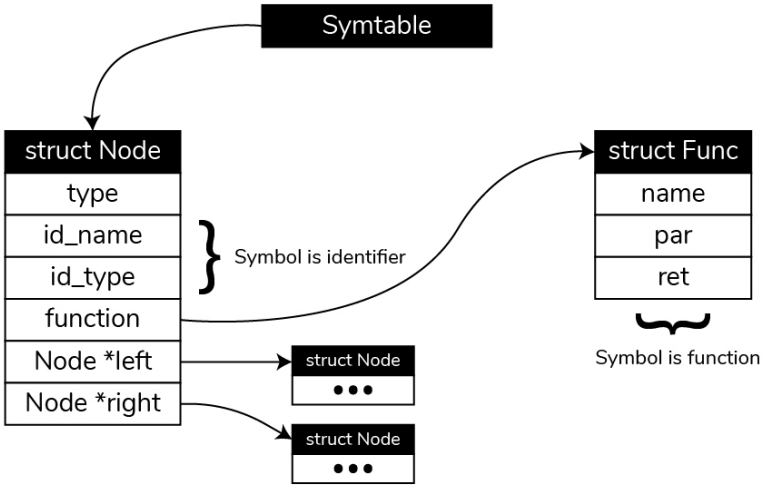
Parser vytvorí token a pošle ho lexikálnemu analyzátoru, ktorý ho naplní pričom, ak vznikne error, tak parser ukončuje program. Všetky tokeny, ktoré sa načítajú sa ukladajú do globálneho zásobníka (toto rozhodnutie padlo na začiatku vývoja kompilátora, keď sme ešte nevedeli, či náhodou nebudeme potrebovat predchádzajúce tokeny). Parser obsahuje okrem funkcii, ktoré predstavujú neterminály v pravidlách, taktiež 2 špeciálne funkcie a to match() a get_next_token().

Predstavme si, že nastane situácia kedy očakávame neterminál ")". Na toto je vytvorená funckia match(RPAREN) (RPAREN je súčasťou enumu), ktorá načíta ďalší token a overí, či ďalší token je naozaj pravá zátvorka, pričom ho "potvrdí (ak je, tak vracia NO_ERROR, ak nie je, tak vracia SYNTAX_ERORR). Nastávajú aj situácie, kedy musíme načítať token a podľa typu tokena sa rozhodnúť, ktoré pravidlo vybrat. Tam používame funkciu get_next_token(), ktorá načíta ďalší token, ale "nepotvrdí"ho. Funkcia get_next_token() načíta najviac jeden token, aj keď je volaná viackrát

a ak je po tejto funkcii volaná funkcia match(), tak nenačítava sa ďalší token, ale "potvrdí" už prednačítaný token.

4 Tabuľka symbolov

Podľa zadania sme implementovali tabuľku symbolov ako tabuľku binárneho vyhľadávacieho stromu. Táto tabuľka slúži na uchovávanie identifikátorov a funkcií, ktoré sa v programe definujú. Každý symbol sa definuje buď ako identifikátor alebo funkcia. Podľa toho o aký typ symbolu ide sa využívajú jednotlivé funkcie na prácu s tabuľkou. Všetky potrebné informácie o daných symboloch sú uložené v univerzálnej štruktúre uzla, ktorý sa používa ako pre identifikátory, tak aj pre funkcie.



Obr. 2: Tabuľka symbolov

Pre jednoduchšiu a efektívnejšiu prácu sme vytvorili globálnu tabuľku pre všetky funkcie a pre identifikátory samostatné tabuľky. Tieto samostatné tabuľky sa vytvoria zakaždým, keď nastane vnorenie do funkcie, cyklu, podmienky apod. Tabuľky sa vždy ukladajú do zásobníku tabuliek. Týmto zabezpečíme, aby sme mali „scope" jednotlivých identifikátorov pod kontrolou. Po vynorení zo „scope" sa tabuľka automaticky vymaže a de-alokuje sa všetka alokovaná pamäť.

Implementovali sme niekoľko funkcií, potrebných na prácu so stromom: inicializácia binárneho stromu, zrušenie binárneho stromu, pridanie identifikátora, pridanie funkcie, vyhľadanie identifikátora/funkcie na základe názvu. Odstránenie jednotlivých položiek nebolo potrebné vzhľadom na implementáciu projektu, keďže nie je potrebné funkcie a ani identifikátory vymazávať. Na konci prekladu sa spustí ešte samostatná de-alokácia tabuliek, ktoré neboli vymazané, aby sme zamedzili memory leak-u.

5 Sémantická analýza a generovanie kódu

Sémantické akcie sú vkladané medzi akcie parsera pričom neovplyvňujú parser, iba spracuvávaju načítané tokeny.
Ako už bolo napísané v úvode, v programe negenerujeme ASS ale používame priame generovanie kódu.
Pri generovaní kódu naplno využívame dátový zásobník inštrukčnej sady, ktorý výrazne uľahčil prácu pri generovaní kódu, hlavne pri výrazoch (5.8).
V nasledujúcich častiach sú vysvetlené jednotlivé príkazy a spôsoby, ktorými ich spracúvavame.

5.1 Príkaz priradenia

Na to aby sme mohli spracovať tento príkaz, musíme ho celý načítať, čo znamená že hodnoty musíme niekde priebežne ukladať a na to nám slúžia zásobniky.
Identifikátory postupne načítavame do zásobniku pričom overujeme či existujú už v tabuľke symbolov. Keď sa dostaneme na koniec príkazu, tak ešte musíme ošetriť typy a ich počet, pričom na to slúži druhý zásobnik, ktorý sa naplní pri tom ako sa vyhodnocujú výrazy.
Vyhodnotenie výrazov zaručuje, že výsledok výrazu bude uložený na dátovom zásobniku, a tým padom je už iba potrebné vybrať hodnoty zo zásobníku v opačnom poradí.
Viz obr. 3.2 v prílohe.

5.2 Definícia premennej

Funguje takmer rovnako ako príkaz priradenia (5.1) až na to, že treba zavolať inštrukciu DEFVAR a uložiť identifikátor do tabuľky symbolov. Viz obr. 4.4 v prílohe.

5.3 Definicia funkcie

Pri spracovaní funkcie sa načíta a uloží meno do premennnej, parametry do zásobníka parametrov a return values do zásobníka return values. Ak sú tieto hodnoty vporiadku (meno funkcie ešte neexituje v tabuľke symbolov, parametry nemajú rovnaké identifikátory), tak sa uloží táto funkcia do tabuľky symbolov a vytvára sa nový strom v zásobníku stromov (viac pri tabulke symbolov (4)). Parametre predávame funkcii cez dátový zásobník v inštručnej sade.
Viz obr. 4.2 a volanie funkcie Viz obr. 4.3 v prílohe.

5.4 Podmienený príkaz (if-else)

If očakáva, že výraz bude boolovského typu, overiť sa to dá zo zásobníku, kam sa uloží typ výrazu a podľa toho sa skáče na návestia.

Problém, ktorý tu vzniká pri viacerých použitiach if-och alebo pri zanorených if-och, je aby sa návestia neopakovali. Toto sme zase vyriešili pomocou zásobníku, do ktorého sa pushne číslo if-u (toto číslo sa každým if-om inkrementuje), a keď dôjde na koniec if-u, tak sa toto číslo pop-ne a vytvorí korektné náveštie.
Viz obr. 3.1 v prílohe.

5.5 For cyklus

For cyklus je svojou konštrukciou podobný if-u, len komplikovanejší.

Na kóde v prílohe na obr. 4.1 v prílohe. si môžete povšimnúť rôzne rozsahy platnosti a taktiež spôsob riešenia toho, že v hlavičke if-u prebieha deklarácia iba raz. Taktiež ako pri if-e, existuje zásobník integerov, pomocou ktorých sa generujú náveštia.

5.6 Rámce

Rôzne rozsahy platnoti riešime tak ako je to napísané pri TS, a to tak, že máme zásobník stromov, pričom unikátne meno premennej je zložené z jej identifikátora a čísla, ktoré určuje poradie stromu v zásobníku. Pri začiatku nového rozsahu platnosti sa všetky premenné, ktoré boli doteraz vytvorené push-nu na dátový zásobník, vytvorí sa nový frame, definuju sa všetky premenné a potom sa pomocou pop-ov nahrajú hodnoty do premenných.
Na konci rozsahu sa pop-ne posledný strom a všetky deklarované premenné sa push-nu, vymaže sa frame a popopujú sa hodnoty naspäť. Názorne je to vysvetlené na príklade v prílohe, viz obr. 3.3. Je nám jasné že takáto konštrukcia nie je legálna v ifj20, ale znázorňuje riešenie napr. pri vnorených if-och bez toho aby ukážkový kód bol zbytočne dlhý.

5.7 Sémantické akcie pri returne

Pri objavení sa returnu vo funkcií sa po vyhodnotení expressions (return expression,expression,...) naplní zásobník typov a kontroluje sa, podla hlavičky funkcie, či sedí počet návratových hodnôt a správnosť ich typov. Tieto hodnoty sa v generovanom kóde pushujú po čom sa podľa zanorenia popujú framy. Zanorenie sa zisťuje zo zásobníku stromov.

Kontrola výskytu returnu:
Po konci funkcie sa kontroluje či funkcia mala return (pomocou globálnej premennej ret_flag, ktorá je buď 1(return present) alebo 0 (return not present)) a či by funkcia mala mať return (return typy v hlavičke)).

5.8 Výrazy

Výrazy spracúvame pomocou precedenčnej tabuľky (8.5). Pri spracovaní používame zásobník, na ktorý si ukladáme znaky, ktoré reprezentujú jednotlivé tokeny a postupujeme podľa algoritmu zdola navrh, ktorý bol preberaný na prednáškach.

Na začiatku, pred spracovaním výrazov si overujeme spätne, či posledný overený token (matched token) je identifikátor, ak áno, tak ho pridáme do spracovania výrazov, ak nie, tak sa pokračuje. Takýmto spôsobom riešime volanie funkcie. Čiže „ID“ ide na volanie funkcie, ale ak je „ID“ a za identifikátorom nie je zátvorka, tak ide do vyhodnotenia výrazov, kde pridá ten identifikátor).

So zásobníkom pracujeme ako so stringom, kde podľa priority operátorou (precedenčnej tabuľky) určujeme, ktorá z možností $<$, $>$, = alebo chyba sa má vykonať.

Pri možnosti $<$ vkladáme znak „$<$“ na zásobník a potom pri $>$ zisťujeme akému pravidlu sa rovná to, čo je za znakom „$<$“, buď skončíme s chybou alebo daným pravidlom, pomocou ktorého generujeme priamo kód.

Typ výrazu ukladáme na zásobník, kde si ho potom overujú iné časti programu.

5.9 Optimalizácie

V kode sme nerobili žiadne optimalizácie, tým pádom samostatný generovaný kód bude pomalý, ale za to, je generácia kódu ľahšia.

6 Práca v tíme

6.1 Rozdelenie práce

Člen tímu	Rozdelenie práce
Fabo Adam (xfaboa00)	organizácia práce, zavedenie štruktúry projektu, syntaktická analýza, sémantická analýza, generovanie kódu, testovanie , dokumentácia
Groma Albert (xgroma00)	syntaktická analýza, syntaktická analýza výrazov, generovanie kódu, sémantická analýza, testovanie, dokumentácia
Gabriš Stanislav (xgabri18)	lexikálna analýza, generovanie kódu, testovanie, sémantická analýza, dokumentácia
Országh Roman (xorsza01)	príprava štruktúr a zásobníkov, generovanie kódu, sémantická analýza, testovanie, dokumentácia

Tabuľka 1: Tabulka rozdelenia práce

7 Záver

Pri riešeni projektu sme si rozšírili skúsenosti s prácou v týme a taktiež sme nabrali množstvo praktických vedomostí. Vypracovavali sme projekt pomocou informácii, ktoré nám poskytli predmety IFJ a IAL, ale taktiež sme sa inšipirovali knihami Crafting a Compiler with C[1] a Algoritmy a štruktúry údajov[2].

8 Prílohy

8.1 Generovanie kódov

```
if (1>2) {
     // true
} else {
     // false
}

LABEL $main
     ...
     PUSHS int@1
     PUSHS int@2
     GTS
     PUSHS bool@true
     JUMPIFNEQS $0IF_LABEL
     CREATEFRAME
     PUSHFRAME
     # true code
     POPFRAME
     JUMP $0ELSE_LABEL

LABEL $0IF_LABEL
     CREATEFRAME
     PUSHFRAME
     # false code
     POPFRAME
LABEL $0ELSE_LABEL
```

Podmienka neplatí

Ukončenie podmienky

Obr. 3.1: Podmienený príkaz (if-else)

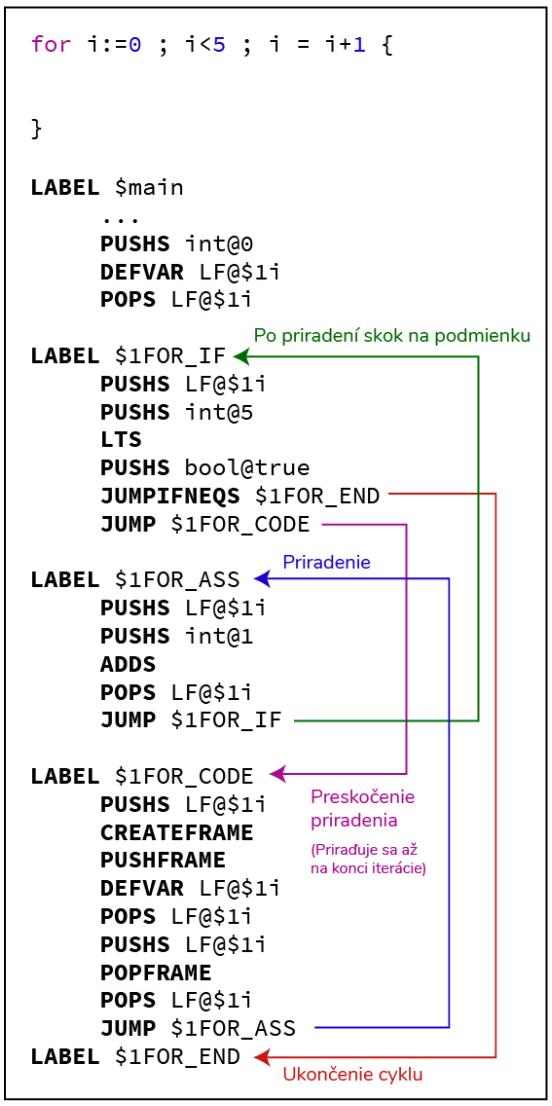
```
s,f,i = "a", 4.20, 9

PUSHS string@a
PUSHS float@daco
PUSHS int@9

POPS LF@$0i
POPS LF@$0f
POPS LF@$0s
```

Obr. 3.2: Príkaz priradenia

```
a := 0
b := 0

{
     a := 0
     a = 4
     b = 5
}

PUSHS int@0
DEFVAR LF@$0a
POPS LF@$0a
PUSHS int@0
DEFVAR LF@$0b
POPS LF@$0b

PUSHS LF@$0a
PUSHS LF@$0b
CREATEFRAME
PUSHFRAME

CREATEFRAME
PUSHFRAME

DEFVAR LF@$0b
DEFVAR LF@$0a
POPS LF@$0b
POPS LF@$0a

PUSHS int@0
DEFVAR LF@$1a
POPS LF@$1a
PUSHS int@4
POPS LF@$1a
PUSHS int@5
POPS LF@$0b
PUSHS LF@$0a
PUSHS LF@$0b
POPFRAME

POPS LF@$0b     # b = 5
POPS LF@$0a     # a = 0
```

Obr. 3.3: Rámce

Obr. 3: Generovanie kódov časť 1.



Obr. 4.1: For cyklus

```
func funkcia(a int) (int) {
    return a
}

LABEL $funkcia
    CREATEFRAME
    PUSHFRAME
    DEFVAR LF@$0a
    POPS LF@$0a

    PUSHS LF@$0a
    POPFRAME
    RETURN
```

Obr. 4.2: Definícia funkcie

```
i = funkica(5)

PUSHS int@5
CALL $funkcia
POPS LF@$0i
```

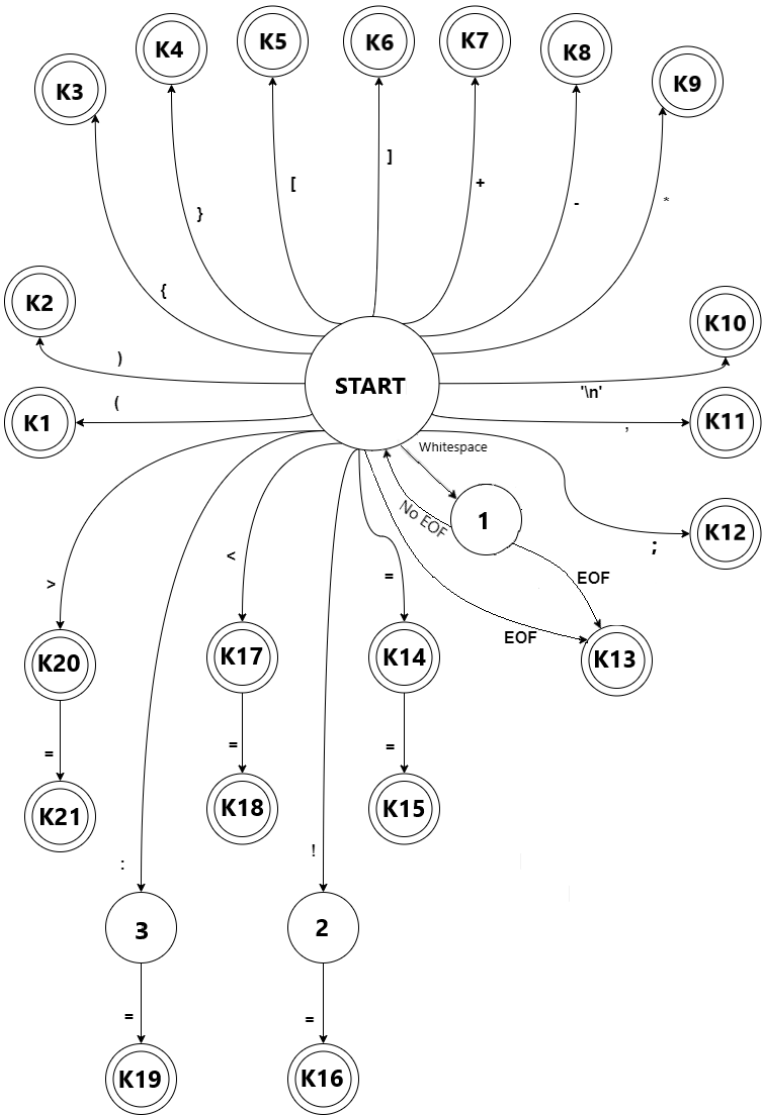
Obr. 4.3: Volanie funkcie

```
i := 0

PUSHS int@0
DEFVAR LF@$0i
POPS LF@$0i
```

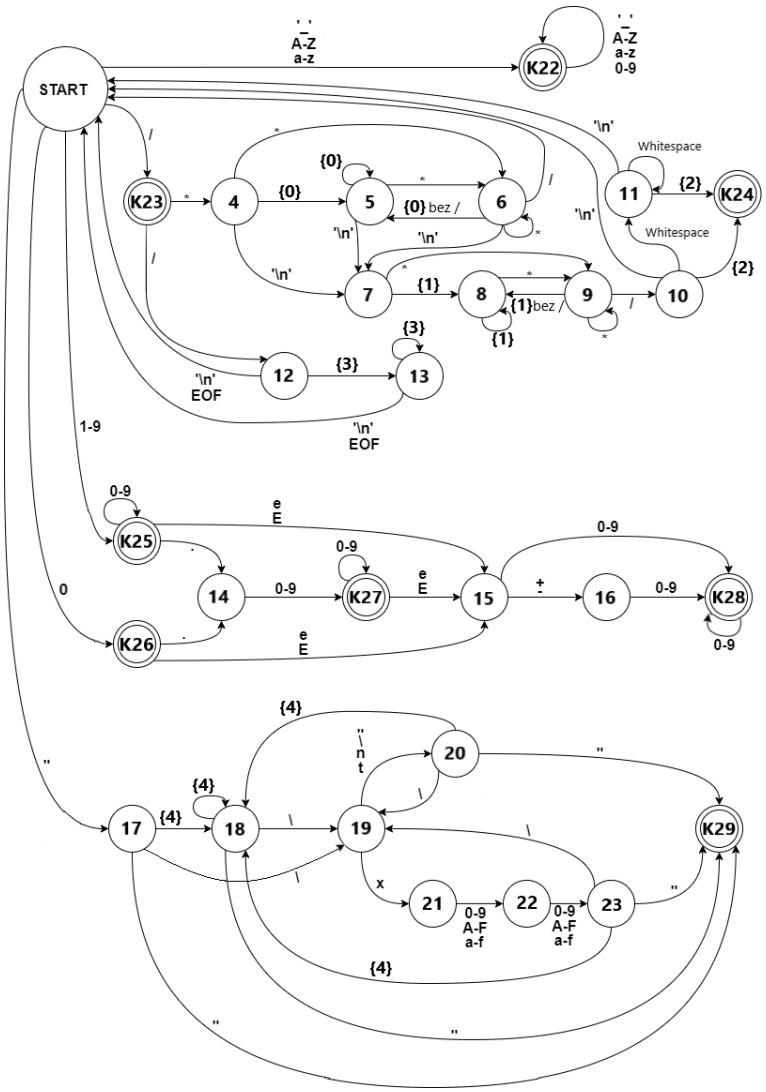
Obr. 4.4: Deklarácia

Obr. 4: Generovanie kódov časť 2.

8.2 Diagram konečného automatu zameraný na lexikálny analyzátor



Obr. 5: Diagram konečného automatu zameraný na lexikálny analyzátor časť 1.



Obr. 6: Diagram konečného automatu zameraný na lexikálny analyzátor časť 2.

Legenda:
{0} - Všetko okrem EOF, '\n', *
{1} - Všetko okrem EOF a *
{2} - Všetko okrem Whitespace a '\n'
{3} - Všetko okrem EOF a '\n'
{4} - ASCII > 31 okrem " a \

K1 - LPAREN
K2 - RPAREN
K3 - LCURL
K4 - RCURL
K5 - LBRACKET
K6 - RBRACKET
K7 - ARITHMOP(ADD)
K8 - ARITHMOP(SUB)
K9 - ARITHMOP(MUL)
K10 - NEWLINE
K11 - COMMA
K12 - SEMICOLON
K13 - SCANEOF
K14 - EQUAL
K15 - RELATIONOP(EE)
K16 - RELATIONOP(NE)
K17 - RELATIONOP(LS)
K18 - RELATIONOP(LSE)
K19 - ASSIGNMENT
K20 - RELATIONOP(GR)
K21 - RELATIONOP(GRE)

K22 - IDENTIFIER/KEYWORD
K23 - ARITHMOP(DIV)
K24 - NEWLINE
K25 - INT_LIT
K26 - INT_LIT
K27 - FLOAT_LIT
K28 - FLOAT_LIT
K29 - STRING_LIT

Obr. 7: Legenda ku diagramu konečného automatu

8.3 LL Tabuľka

	PACKAGE	FUNC	NEWLINE	ID	(	)	,	:=	=	INT	STRING	FLOAT64	IF	ELSE	FOR	RETURN	$
<program>	1		1														
<prolog>	3		2														
<func_list>		4	5														6
<func>				7													
<func_head>				8													
<param_list>					9												
<parameters>				10													11
<param_tail>							12										13
<return_types>					14												15
<return_list>										16	16	16					17
<ret_list_tail>							18										19
<my_type>										20	21	22					
<statement_list>			27	23									24		25	26	28
<id_type>					29												
<id_tail>							31										32
<def>								33	34								
<exp_equal>				35													36
<func_or_exp>					37												
<exp_main_tail>							39										40
<my_return>						43											42
<func_call>						43											
<term>				45						46	47	48					
<term_tail>							49										50
<if_statement>					51												
<if_head>					52												
<if_else>														53			
<for_cycle>				54													
<for_head>				55													
<for1>				56													57
<for2>																	58
<for3>				59													60

Obr. 8: LL Tabuľka

8.4 LL Gramatika

1. <program> → <prolog><func_list>SCANEOF
2. <prolog> → NEWLINE <prolog>
3. <prolog> → package main NEWLINE
4. <func_list> → func <func><func_list>
5. <func_list> → NEWLINE <func_list>
6. <func_list> → ε
7. <func> → <func_head><statement_list>NEWLINE } NEWLINE
8. <func_head> → ID <param_list><return_types>{ NEWLINE
9. <param_list> → (<parameters>)
10. <parameters> → ID <type><param_tail>
11. <parameters> → ε
12. <param_tail> → , ID <type><param_tail>
13. <param_tail> → ε
14. <return_types> → (<return_list>)
15. <return_types> → ε
16. <return_list> → <my_type><ret_list_tail>
17. <return_list> → ε
18. <return_list_tail> → , <my_type><ret_list_tail>
19. <return_list_tail> → ε
20. <my_type> → INT
21. <my_type> → STRING
22. <my_type> → FLOAT64
23. <statement_list> → ID <id_type>NEWLINE <statement_list>
24. <statement_list> → if <if_statement><statement_list>
25. <statement_list> → for <for_cycle><statement_list>
26. <statement_list> → return <my_return>NEWLINE <statement_list>
27. <statement_list> → NEWLINE <statement_list>
28. <statement_list> → ε
29. <id_type> → (<func_call>
30. <id_type> → <id_tail><def>
31. <id_tail> → , ID <id_tail>
32. <id_tail> → ε
33. <def> → := <expression>
34. <def> → = <exp_equal>
35. <exp_equal> → ID <func_or_exp>
36. <exp_equal> → <expression><exp_main_tail>
37. <func_or_exp> → (<func_call>
38. <func_or_exp> → <exp_main_tail>
39. <exp_main_tail> → , <exp_main><exp_main_tail>
40. <exp_main_tail> → ε
41. <my_return> → <expression><exp_main_tail>
42. <my_return> → ε
43. <func_call> →) NEWLINE
44. <func_call> → <term><term_tail>) NEWLINE
45. <term> → ID
46. <term> → STRING
47. <term> → INT
48. <term> → FLOAT64
49. <term_tail> → , <term><term_list>
50. <term_tail> → ε
51. <if_statement> → <if_head>{ NEWLINE <statement_list>NEWLINE } <if_else>
52. <if_head> → (<expression>)
53. <if_else> → else { NEWLINE <statement_list>} NEWLINE
54. <for_cycle> → <for_head>{ NEWLINE <statement_list>}
55. <for_head> → <for1>; <for2>; <for3>
56. <for1> → ID := <expression>
57. <for1> → ε
58. <for2> → <expression>
59. <for3> → ID <id_tail>= <exp_equal>
60. <for3> → ε

Tabuľka 2: LL Gramatika

8.5 Precedenčná tabuľka

Symboly:

`i` značí identifikátor a konštanty INT_LIT, FLOAT_LIT, STRING_LIT, IDENTIFIER (operandy)

`r` značí relačné operátory >, >=, <, <=, ==, != , ktoré majú rovnakú prioritu a zároveň ich je veľa, tak sme sa ich rozhodli dať do jednej skupiny v tabuľke

`$` značí začiatok a koniec vstupu

Vstup / Zásobník	+	-	*	/	(	)	i	r	$
+	>	>	<	<	<	>	<	>	>
-	>	>	<	<	<	>	<	>	>
*	>	>	>	>	<	>	<	>	>
/	>	>	>	>	<	>	<	>	>
(	<	<	<	<	<	=	<	<	
)	>	>	>	>		>		>	>
i	>	>	>	>		>		>	>
r	<	<	<	<	<	>	<	>	>
$	<	<	<	<	<		<	<	

Tabuľka 3: Precedenčná tabuľka

Literatúra

[1] FISCHER, C. N. *Crafting a Compiler with C*. 1. vyd. [b.m.]: Addison Wesley, 1991. ISBN 978-0805321661.

[2] WIRTH, N. *Algoritmy a štruktúry údajov*. [b.m.]: Alfa, 1988. ISBN 063-030-87.