Projekt do předmětu IFJ

TÝM 057, VARIANTA I

Jakub Zapletal (xzaple36) - vedoucí - 25%
Michal Zilvar (xzilva02) - 25%
Martin Studený (xstude23) - 25%
Radek Wildmann (xwildm00) - 25%

Obsah

1 Lexikální analýza

Lexikální analyzátor je implementován pomocí konečného stavového automatu dle zadání.

V naší implementaci pro generování tokenu slouží funkce **int getToken(String *s, int *cursor)**, která vrací integerovský identifikátor tokenu (viz. *define.h*), prvním parametrem vrací identifikátor, případně klíčové slovo a pro možnost opakovaného čtení slovo a vrácení zpět ještě druhým parametrem počet přečtených znaků pro identifikaci tokenu.

Vracení tokenů na vstup jsme původně dělali tak, že jsme funkci, která toto obstarávala, předali řetězec reprezentující token a podle něj byl posunut kurzor vstupu. Toto bylo ovšem problematické, jelikož to nebralo v potaz whitespace, který se na vstupu vyskytoval a proto jsme se uchýlili k předávání a uchovávání informace a tom, kolik bylo přečteno znaků pro identifikaci tokenu. Toto náš problém vyřešilo.

Další mírně problematickým úkolem bylo zpracování řetězcových literálů. pro potřeby interpretu bylo nutné některé znaky nahrazovat escape sekvencemi. toto jsme původně dělali až během sémantických akcí, ale nakonec jsme se rozhodli, že to provedeme již během lexikální analýzy.

2 Syntaktická a sémantická analýza

Syntaktická analýza je prováděna pomocí rekurzivního sestupu, během nějž zároveň dochází k sémantické analýze a generování instrukcí pro interpret. Zvolený způsob řešení přesně odpovídá metodě, která byla probírána na přednáškách.

2.1 Precedenční syntaktická analýza

Precedenční analýza byla implementována přesně podle algoritmu v prezentaci, tedy za použití zásobníku. V průběhu tohoto algoritmu jsou prováděny i veškeré nutné typové konverze a generování instrukcí.

Pro potřeby precedenční analýzy byla vytvořena datová struktura *tokenparam*. Tato struktura reprezentuje token s informacemi, které jsou nápomocné při precedenční analýze. Struktura obsahuje identifikátor tokenu, který se mírně liší od identifikátoru, který je používaný ve zbytku překladače a to proto, aby se tímto identifikátorem zároveň mohla indexovat precedenční tabulka, která byla realizována jako dvourozměrné pole. Dále je tam uložen datový typ, přepínač, indikující, zda se jedná o proměnnou či literál, a hodnota literálu či proměnné, která byla použita při debuggování.

Struktura je definována takto:

```
typedef struct TokenParam {
    char token;
    String *data;
    int type;
    bool identifier;
} tokenparam;
```

Při řešení syntaktické analýzy byla problematickou pouze analýza precedenční. Vyzkoušeli jsme více způsobů uchovávání informace o tokenech, ale nakonec jsme zvolili definování tokenu jako struktury a potřebné informace si uchovávali v ní. Ve výsledku to bylo mnohem vhodnější řešení než prve používané globální proměnné.

2.2 Sémantická analýza

Sémantická analýza a generování sémantických akcí probíhá spolu se syntaktickou analýzou. V průběhu vývoje jsme měli 2 různé iterace sémantické analýzy.

V naší první iteraci probíhaly všechny kontroly a sémantické akce až po tom, co jsme syntakticky zkontrolovali celý příkaz. Například pokud se jednalo o definici proměnné s její inicializací, nejprve jsme syntakticky

zkontrolovali celý příkaz až po konec řádku a pak zkontrolovali zda proměnná neexistuje, vložili ji do tabulky symbolů a vygenerovali příslušnou instrukci. Postupně jsme však naráželi na problém toho, že jsme si museli pamatovat zbytečně mnoho informací zpět, které jsme museli ukládat do globálních proměnných nebo si je předávat parametry. Obzvláště krkolomné se toto ukázalo být u definice funkce s parametry. Tyto parametry jsme si ukládali postupně do globálního lineárního seznamu, který jsme potom najednou vložili do tabulky symbolů fce.

Řešení, které jsme implementovali poté, a u nějž jsme i zůstali, je předčasné vkládání symbolů do tabulky a následné upravování jejich atributů. Toto se ukázalo jako příjemnější řešení. Například při zpracování definice funkce vložíme symbol do tabulky hned po přečtení id funkce a následně vkládáme parametry po jednom už přímo do tabulky a nemusíme si je pamatovat.

Samotné instrukce jsou generováný v průběhu analýzy a okamžitě vypisovány na stdout tak, jak jsou generovány.

3 Tabulka symbolů

Jelikož zadání jinou formu ani nepřipouštělo, je tabulka symbolů implementována pomocí binárního vyhledávacího stromu. Binární strom jsme již dělali v domácí úloze v předmětu IAL. Pro potřeby překladače jsme tento strom museli podstatně změnit. Základní funkce jsou však v principu stejné. Složitější je především z důvodu velkého množství informací, který musí každý uzel obsahovat, včetně například vnořeného stromu.

3.1 Struktura uzlu

Každý uzel představuje buď funkci, nebo proměnnou. Proměnná ani funkce nesmí mít stejný název, je tedy zajištěna jedinečnost. Proto název představuje přímo klíč uzlu. klíč je vždy malým písmem, bylo totiž potřeba ošetřit fakt, že překládaný jazyk je case-insensitive.

Princip řazení je obdobný abecednímu řazení. Toho je docíleno porovnáváním hodnot znaků v ASCII tabulce. Hodnoty s menší hodnotou jsou vlevo, s vyšší vpravo.

Všechny důležité informace jsou v uloženy v datové struktuře *load*, na kterou odkazuje *loadPtr symbol*.

```
struct node {
    char key[64];
    struct node *lPtr;
    struct node *rPtr;
    loadPtr symbol;
};
```

3.2 Informace o uzlu

Informace o tom, zda uzel představuje proměnnou, nebo funkci, je uložena ve struktuře *load*. Konkrétně v proměnné *metaType* - pokud se rovná 1, jde o proměnnou, pokud se rovná 2, jde o funkci.

Proměnná *type* zase určuje datový typ. Pro *type=1* je datový typ integer, pro *type=2* double a pro *type=3* je datový typ string.

Ve struktuře *value* se pak nachází konkrétní hodnota.

Booleovské proměnná *defined* je pouze pro funkce a ukazuje, jestli už byla funkce definovaná. *Initialized* je naopak pouze pro proměnné a ukazuje, jestli už proměnná byla inicializovaná.

Struktura *functInfo* je opět pouze pro funkce a nese informaci o tom, jestli již byla funkce deklarovaná, jestli už se v kódu narazilo na *return*, a především obsahuje parametry, které byly funkci předány. Ty jsou obsaženy v lineárním seznamu *parameters*. Zároveň se v *functInfo* nachází ukazatel na vnořený strom, který eviduje lokální proměnné.

```
typedef struct load {
    bool defined;
    bool initialized;
    int type;
    int metaType;
    val value;
    functInfo function;
} *loadPtr;

struct parameters{
    char name[64];
    int type;
    param next;
};
```

```
typedef struct value {
    int i;
    double d;
    String *s;
    bool b;
} val;

typedef struct {
    param parameters;
    param declaredParameters;
    nodePtr functTable;
    bool hasReturn;
    bool declared;
} functInfo;
```

4 Operace s pamětí

Vzhledem k množství operací s dynamicky alokovanou pamětí bylo velmi náročné ošetřit ztráty paměti, takzvané **memory leaky**. Z toho důvodu jsem se rozhodl o systematickou správu a dohled nad jakoukoliv alokací a uvolněním. Byl implementován jednosměrný lineární seznam a dceřiná funkce pro alokaci a uvolnění. S jakýmkoliv zavoláním alokace paměti ukládáme nový ukazatel na konec lineárního seznamu. Při uvolnění paměti pak procházíme seznam a je nutné ukazatel vymazat i z něj. Při jakémkoliv konci programu jsou vyčištěny všechny aktuální dynamicky alokované ukazatelé. Díky tomu se nám podařilo zabránit všem memory leakům na úkor paměti a procesoru. Překladač je program, který se spouští pouze příkazem uživatele a nezpracovává data non-stop. Z toho důvodu si takové ošetření můžeme dovolit, leč není nejlepším řešením.

5 Vývoj

Od loňského roku jsme se s týmem poučili jelikož jsme minulý rok s projektem jako tým pohořeli, rozhodli jsme se pro jiný přístup. Loni byl náš postup poněkud chaotický, na začátku jsme si každý zadali téma a pracovali na něm ve vakuu, což se nám extrémně vymstilo při pokusu o spojení částí do jednoho celku.

Letos jsme však pracovali na projektu zásadně společně a to okamžitě po odhalení zadání, všichni pohromadě, tak, aby každý věděl na čem dělají ti druzí a mohli se podle toho zařídit a případně hned říct co potřebuje od toho druhého a části kompilátoru kterou programuje. Díky tomu, že jsme programovali zásadně pohromadě se nám dařilo mnohem lépe jednotlivé části propojit a setkali jsme se s menším množstvím problémů typu ”musíme revertovat každý druhý commit protože si neustále mažeme soubory”. I tak jsme se ale párkrát setkali s nutností řešit kolize a podobně. Takové problémy se ale pohromadě řeší mnohem lépe.

Celkově jsme s týmovou prací i s postupem vývoje letos spokojeni a dařilo se nám mnohem lépe jak po komunikační tak po implementační stránce.

6 Závěr

Ve výsledku jsme s projektem spokojeni. I přesto, že byl projekt zjednodušen oproti loňskému zadání, které jsme nezvládli, se nám zdá být projekt stále tím nejtěžším, který jsme museli na FITu dělat. Nejtěžší na projektu je rozhodně týmová práce, nustnost koordinovat svoje úsilí a dát jednotlivé části projektu dohromady. I přes obtížnost je to ovšem rozhodně jeden z nejzajímavějších projektů na kterém jsme pracovali.

7 Rozdělení práce

Při práci jsme si jednotlivé úseky rozdělili jen orientačně, často se stalo, že jsme na jedné části pracovali ve více lidech. Naše původní rozdělení práce ovšem vypadalo takto:

- **Martin Studený** - Martin pracoval převážně na tabulce symbolů a na funkcích pro sémantickou kontrolu a sémantických akcích.
- **Radek Wildmann** - Radek implementoval precedenční syntaktickou analýzu a základ knihovny *string.c*.
- **Michal Zilvar** - Michal pracoval na knihovně *string.c* a na funkcích pro správu a úklid paměti alokované našimi funkcemi. Dále vytvořil funkci pro generování instrukcí interpretu a v neposlední řadě pracoval na lexikálním analyzátoru.
- **Jakub Zapletal** - Jakub pracoval na syntaktické analýze, sémantické analýze a generování instrukcí interpretu.

Každý si svoji část implementace dokumentoval sám.

8 LL tabulka

	eps	if	eof	eol	decl	funct	expr	(	)	then	as	else	id	lit	int	double	str	scope	end	,	ret	=	dim	in	print	do	while	loop
PROGRAM	1				1	1																						
FUNCTIONS	3				2	2																						
FUNCTION					4	5																						
PARAMS	7												6															
NPARAM	9																			8								
PARAM													10															
SCOPE																		11										
FCOMMANDS	15	13											13								12		14	13	13	13		
FCOMMAND																					16							
SCOMMANDS	19	18											18										17	18	18	18		
SCOMMAND																							20					
COMMANDS	22	21											21											21	21	21		
COMMAND		26											23											24	25	27		
VARDEF																							28					
DEFINIT	30																						29					
INIT							31						32															
FCALL													33															
CPARAMS	35												34	34														
NCPARAM	37																			36								
CPARAM													38	38														
INPUT																								39				
PRINT																									40			
NEXPR	42						41																					
BRANCH		43																										
LOOP																										44		
TYPE															45	46	47											
TERM													49	48														

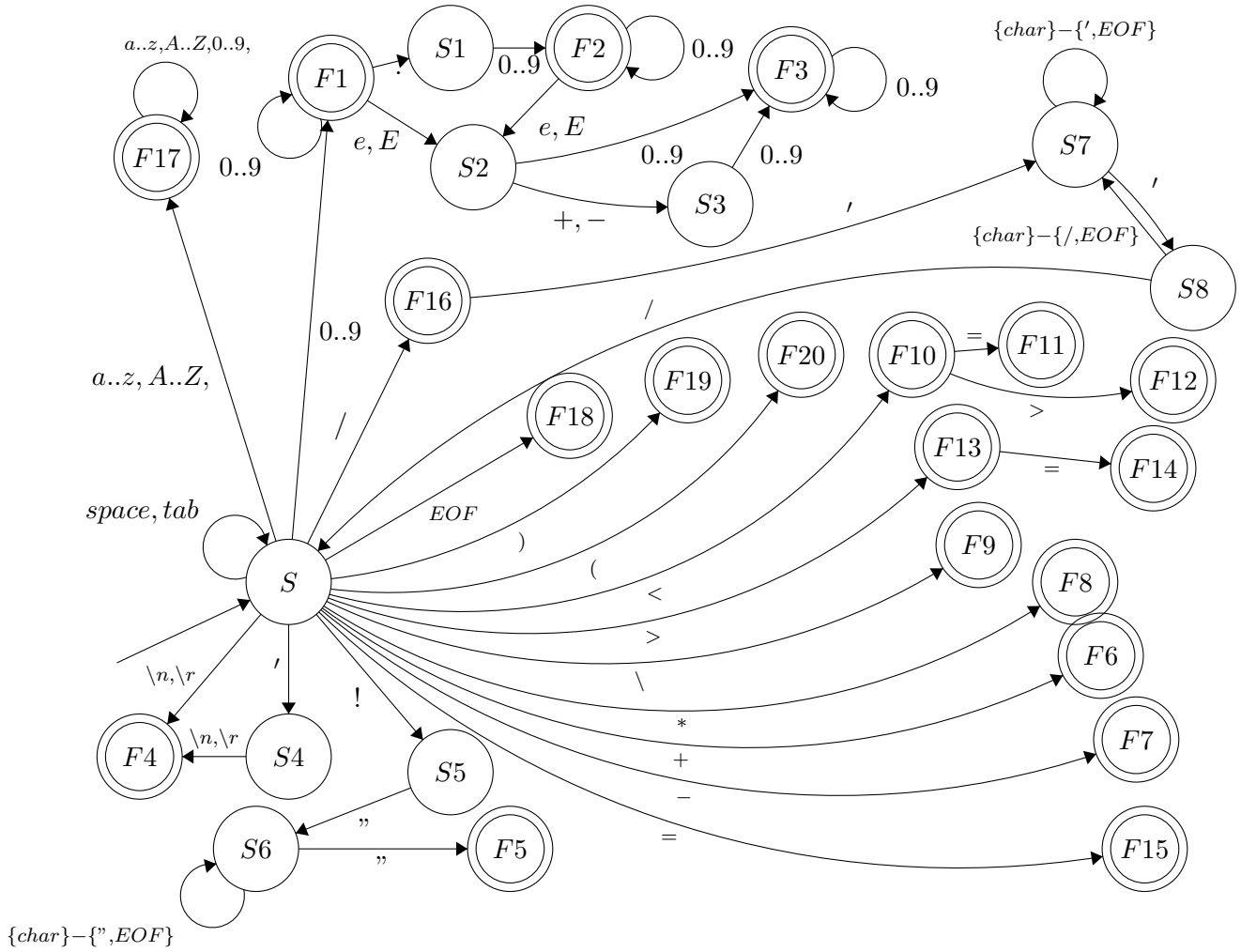
9 Pravidla gramatiky

1	PROGRAM → FUNCTIONS SCOPE eof
2	FUNCTIONS → FUNCTION FUNCTIONS
3	FUNCTIONS → epsilon
4	FUNCTION → Declare Function id lb PARAMS rb As TYPE eol
5	FUNCTION → Function id lb PARAMS rb As TYPE eol FCOMMANDS End Function
6	PARAMS → PARAM NPARAM
7	PARAMS → epsilon
8	NPARAM → comma PARAM NPARAM
9	NPARAM → epsilon
10	PARAM → id As TYPE
11	SCOPE → Scope eol SCOMMANDS End Scope
12	FCOMMANDS → FCOMMAND FCOMMANDS
13	FCOMMANDS → COMMAND FCOMMANDS
14	FCOMMANDS → SCOMMAND FCOMMANDS
15	FCOMMANDS → epsilon
16	FCOMMAND → Return EXPR eol
17	SCOMMANDS → SCOMMAND SCOMMANDS
18	SCOMMANDS → COMMAND SCOMMANDS
19	SCOMMANDS → epsilon
20	SCOMMAND → VARDEF eol
21	COMMANDS → COMMAND COMMANDS
22	COMMANDS → epsilon
23	COMMAND → id equals INIT eol
24	COMMAND → INPUT eol
25	COMMAND → PRINT eol
26	COMMAND → BRANCH eol
27	COMMAND → LOOP eol
28	VARDEF → Dim id As TYPE DEFINIT
29	DEFINIT → equals INIT
30	DEFINIT → epsilon
31	INIT → EXPR
32	INIT → FCALL
33	FCALL → id lb CPARAMS rb
34	CPARAMS → CPARAM NCPARAM
35	CPARAMS → epsilon
36	NCPARAM → comma CPARAM NCPARAM
37	NCPARAM → epsilon
38	CPARAM → TERM
39	INPUT → Input id
40	PRINT → print EXPR semicolon NEXPR
41	NEXPR → EXPR semicolon NEXPR
42	NEXPR → epsilon
43	BRANCH → If EXPR Then eol COMMANDS Else eol COMMANDS End If
44	LOOP → Do While EXPR eol COMMANDS Loop
45	TYPE → Integer
46	TYPE → Double
47	TYPE → String
48	TERM → literal
49	TERM → id

10 Tabulka precedenční analýzy

	ID	(	)	+	-	*	/	\	<	>	<=	>=	==	<>	$
ID			]	]	]	]	]	]	]	]	]	]	]	]	]
(	[	[	=	[	[	[	[	[	[	[	[	[	[	[	
)			]	]	]	]	]	]	]	]	]	]	]	]	]
+	[	[	]	]	]	[	[	[	]	]	]	]	]	]	]
-	[	[	]	]	]	[	[	[	]	]	]	]	]	]	]
*	[	[	]	]	]	]	]	]	]	]	]	]	]	]	]
/	[	[	]	]	]	]	]	]	]	]	]	]	]	]	]
\	[	[	]	]	]	[	[	]	]	]	]	]	]	]	]
<	[	[	]	[	[	[	[	[							]
>	[	[	]	[	[	[	[	[							]
<=	[	[	]	[	[	[	[	[							]
>=	[	[	]	[	[	[	[	[							]
==	[	[	]	[	[	[	[	[							]
<>	[	[	]	[	[	[	[	[							]
$	[	[		[	[	[	[	[	[	[	[	[	[	[	

11 Stavový automat pro lexikální analýzu



12 Koncové stavy automatu pro lexikální analýzu

Stav	Programová konstanta	Znaky
F1	L_INT	celočíselný literál
F2	L_FLOAT	desetinný literál
F3	L_FLOAT	desetinný literal s exponentem
F4	T_EOL	konec řádku
F5	L_STRING	řetězcový literál
F6	T_ADD	+
F7	T_SUB	-
F8	T_TIMES	*
F9	T_IDIV	\
F10	T_LT	<
F11	T_LTE	<=
F12	T_NEQ	<>
F13	T_GT	>
F14	T_GTE	>=
F15	T_EQ	=
F16	T_DIV	/
F17	T_ID nebo T_*KEYWORD*	identifikátor nebo klíčové slovo
F18	T_EOF	konec souboru
F19	T_RB	)
F20	T_LB	(