Vysoké učení technické v Brně
Fakulta informačních technologií

Dokumentace projektu z předmětu IFJ a IAL

Implementace překladače jazyka IFJ20

Tým 101, varianta I

Jakšík, Aleš	**(xjaksi01)**	**25%**
Vlasáková, Nela	(xvlasa14)	25%
Mráz, Filip	(xmrazf00)	25%
Bělohlávek, Jan	(xbeloh08)	25%

5. 12. 2020

Obsah

1 Úvod **3**

2 Návrh **4**

3 Implementace **4**
3.1 Lexikální analýza - `scanner.c` 5
3.1.1 Dynamický řetězec - `dynamicString.c` 5
3.2 Syntaktická a sémantická analýza 5
3.2.1 Syntaktická analýza - `parser.c` 5
3.2.2 Sémantická analýza- `parser.c` 5
3.2.3 Precedenční analýza výrazů - `expression.c` 6
3.3 Generování kódů - `generator.c` 7

4 Konečný automat **8**

5 LL gramatika **9**

6 LL tabulka **10**

7 Zdroje **11**

1 Úvod

Zadáním projektu bylo vytvořit překladač imperativního jazyka IFJ20 a na jeho vytvoření se podíleli všichni členové týmu. Naživo jsme se viděli jen jednou, pravidelně jsme však komunikovali pomocí Messengeru a hovory jsme realizovali na společném Discordovém serveru. Práci jsme si rozdělili následovně:

Aleš Jakšík

Vedoucí týmu dostal na práci syntaktickou a sémantickou analýzu vyjma analýzy výrazů, navíc také implementoval tabulku symbolů.

Soubory:

- `parser.c/parser.h`
- `symtable.c/symtable.h`

Nela Vlasáková

Na práci dostala syntaktickou a sémantickou analýzu výrazů, k čemuž patří i implementace obousměrně vázaného seznamu, dokumentaci a testování kódu.

Soubory:

- `expression.c/expression.h`
- `exprList.c/exprList.h`

Filip Mráz

Na starost dostal generování výsledného kódu a implementaci dynamického řetězce, dále se podílel na implementaci obousměrně vázaného seznamu pro lexikální analyzátor.

Soubory:

- `generator.c/generator.h`
- `tokenList.c/tokenList.h`
- `dynamicString.c/dynamicString.h`

Jan Bělohlávek

Honzovým úkolem bylo vytvořit lexikální analyzátor, pro který také vytvořil konečný automat.

Soubory:

- `scanner.c/scanner.h`

2 Návrh

Celý překladač je složen z několika spolupracujících modulů. Každý má většinou jednu hlavní funkci, která je z jiného modulu volána, a tak tedy dojde k jakési komunikaci a předání informací. Kromě vrácení návratové hodnoty mohou takové funkce také zapisovat do různých proměnných (pomocí předání ukazatele na danou proměnnou), a oznamovat tak jiným modulům různé skutečnosti, příkladem by mohlo být například získání datového typu výrazu. Chybová návratová hodnota je předávána vždy zpět od místa, kde se vyskytla, až do `main.c`, kde je vyhodnocena, je na standartní chybový výstup vytištěna chybová hláška a program je ukončen s touto předávanou hodnotou.

K implementaci jsme využili dvou datových struktur - **binárního stromu** (pro tabulku symbolů) a **oboustranně vázaného seznamu**, který je používán jak pro nahrání celého vstupního souboru, tak například pro předávání výrazů k analýze, nebo následně předání výrazu ve formě postfixu k tištění.

Pro lexikální analýzu jsme využili **konečného automatu**, při syntaktické kontrole jsme využili LL-gramatiky a **metody rekurzivního sestupu**. Výrazy jsme zpracovali podle **precedenční syntaktické analýzy**.

3 Implementace

Jako první je ze souboru `main.c` volán syntaktický/sémantický analyzátor - `parser.c`. Ten si zavolá lexikální analyzátor (jde tedy o syntaxí řízený překlad), který projde celý vstupní soubor, provede lexikální analýzu podle konečného automatu a vytvoří obousměrně vázaný seznam S tzv. *Tokeny* [Kód 1].

```
typedef struct Token {
    TokenType t_type;
    tStr *atribute;
    struct Token *lptr;
    struct Token *rptr;
} *TokenPtr;
```

Kód 1: Implementace struktury tokenu

Tento seznam pak předá zpět syntaktickému/sémantickému analyzátoru, který jej projde a provede syntaktickou kontrolu podle LL gramatiky, následně pak zkontroluje sémantiku. Pokud narazí na místo, kde by se měl nacházet výraz, vytvoří z něj další obousměrně vázaný seznam a pošle jej na precedenční analýzu do `expression.c`, kde je výraz zkontrolován jak po sémantické, tak syntaktické stránce, a v případě, že je vše v pořádku, je vstupní seznam reprezentující výraz převeden na postfix a odeslán dále do generátoru výstupního kódu.

3.1 Lexikální analýza - `scanner.c`

Lexikální analýza je řízena konečným automatem [Obrázek 1]. Ze vstupu načte vždy jeden znak a postupně je přidává do předem připraveného řetězce, dokud nedojde do stavu, kdy může tento řetězec přijmout nebo zamítnout. Pokud jej může přijmout, vytvoří z něj *Token* a přidá jej do obousměrně vázaného seznamu, který poté, co narazí na `EOF` (v místě, kde `EOF` může podle automatu přijít), považuje soubor za lexikálně korektní a předá jej zpět do `parser.c`.

3.1.1 Dynamický řetězec - `dynamicString.c`

Práci s řetězci máme řešenou pomocí implementace vlastního dynamického řetězce. Při inicializaci je alkováno místo o předem dané velikosti `STRING_LEN` a při každém přidání nového znaku do takto inicializovaného řetězce se v případě, že místo v paměti již existujícího řetězce není dostačující, zvětší vyhrazený prostor o dalších `STRING_LEN`, což je předem vytvořená konstanta o hodnotě 10.

3.2 Syntaktická a sémantická analýza

Syntaktická kontrola je řízena pravidly **LL gramatiky**, informace potřebné k sémantické kontrole (existence proměnných a funkcí, datové typy a podobně) jsou uchovávány v **tabulce symbolů** implementované pomocí binárního stromu.

3.2.1 Syntaktická analýza - `parser.c`

Začátkem syntaktické analýzy je kontrola, jestli vstupní soubor obsahuje povinnou hlavičku. Následně se celý seznam, který obsahuje *Tokeny* reprezentující vstupní soubor, projde poprvé. Funkce `buidInFunc` projde všechny hlavičky funkcí, uloží je do tabulky symbolů spolu s informacemi o vstupních a výstupních parametrech (počet a datový typ). Také zkontroluje, že poslední funkcí je `main`. Po provedení tohoto prvního běhu se pak celý seznam projde znova a kontrolují se těla jednotlivých funkcí. Tato kontrola se řídí pravidly LL gramatiky [Obrázek 5].

3.2.2 Sémantická analýza- `parser.c`

Při procházení těly funkcí se do tabulky symbolů nahrávají informace o proměnných. Každé tělo představuje novou tabulku symbolů a postup vytváření tabulek je následující:

1. **Vstup do těla** (funkce, konstrukce `if`, cyklus)
2. **Vytvoří se nová tabulka symbolů** a vloží se do seznamu tabulek symbolů (vždy na první místo)
    - Hlavička cyklu je vždy samostatná nová tabulka, tudíž je zaručeno, že v ní lze použít proměnné definované dříve a zároveň, že proměnné definované v hlavičce budou dostupné v jejím těle
3. **Odchod z těla** znamená odstranění tabulky ze seznamu tabulek symbolů

Ověření, že nějaká proměnná již byla definována (a případné získání datového typu a podobně), pak vypadá tak, že se první podíváme do tabulky na začátku seznamu, a pokud nic nenajdeme, jdeme hlouběji, dokud ji nenajdeme, v opačném případě pak nastává chyba.

3.2.3 Precedenční analýza výrazů - `expression.c`

Hlavní funkcí analyzátoru výrazů je funkce `parseExp`. Ta obdrží seznam *Tokenů*, podle kterého následně vytvoří vlastní seznam, kde jsou prvky identifikovány podle symbolů precedenční tabulce, kterou je celá precedenční analýza řízena.
Ta je v kódu implementována jako dvourozměrné pole znaků:

	input								
opStack		+	-	*	/	(	)	**cmp**	**$**
	+	R	R	S	S	R	S	R	R
	-	R	R	S	S	R	S	R	R
	*	R	R	R	R	R	S	R	R
	/	R	R	R	R	R	S	R	R
	(	S	S	S	S	R	S	R	R
	)	S	S	S	S	S	S	S	E
	cmp	R	R	R	R	R	E	R	R
	$	S	S	S	S	S	S	E	A

Princip pro postup při precedenční analýze (a zároveň systém více seznamů) byl nastudován zde [2]. Žádný kód nebyl převzat. Analýza pracuje se třemi seznamy:

1. `input` reprezentující vstupní výraz
2. `idStack`, do kterého se vkládají výrazů
3. `opStack`, kam jsou vkládány operátory

Proměnná, ať už jde o nějaký identifikátor, číslo, desetinný literál nebo řetězec, je klasifikováno jako `E` (tedy jako výraz). Pokud při procházení seznamem narazíme na prvek označný jako `E`, rovnou jej vložíme do příslušného seznamu. Samotný proces precedenční analýzy pak probíhá následovně:

1. Symbol z precedenční tabulky získáme vždy zadáním "souřadnic":

   ```
   precTable[x][y]
   ```

   kde `x` reprezentuje symbol na konci seznamu `OpStack` a `y` reprezentuje symbol, na který se právě díváme ve vstupním seznamu.
2. Pokud dostaneme `R`, provedeme redukci
3. Pokud dostaneme `S`, provedeme přesunutí symbolu ze vstupu `input` na `opStack`
4. Pokud dostaneme `A`, kontrolujeme, konečné podmínky přijetí výrazu a pokud projdou v pořádku, výraz převádíme na *postfix*.

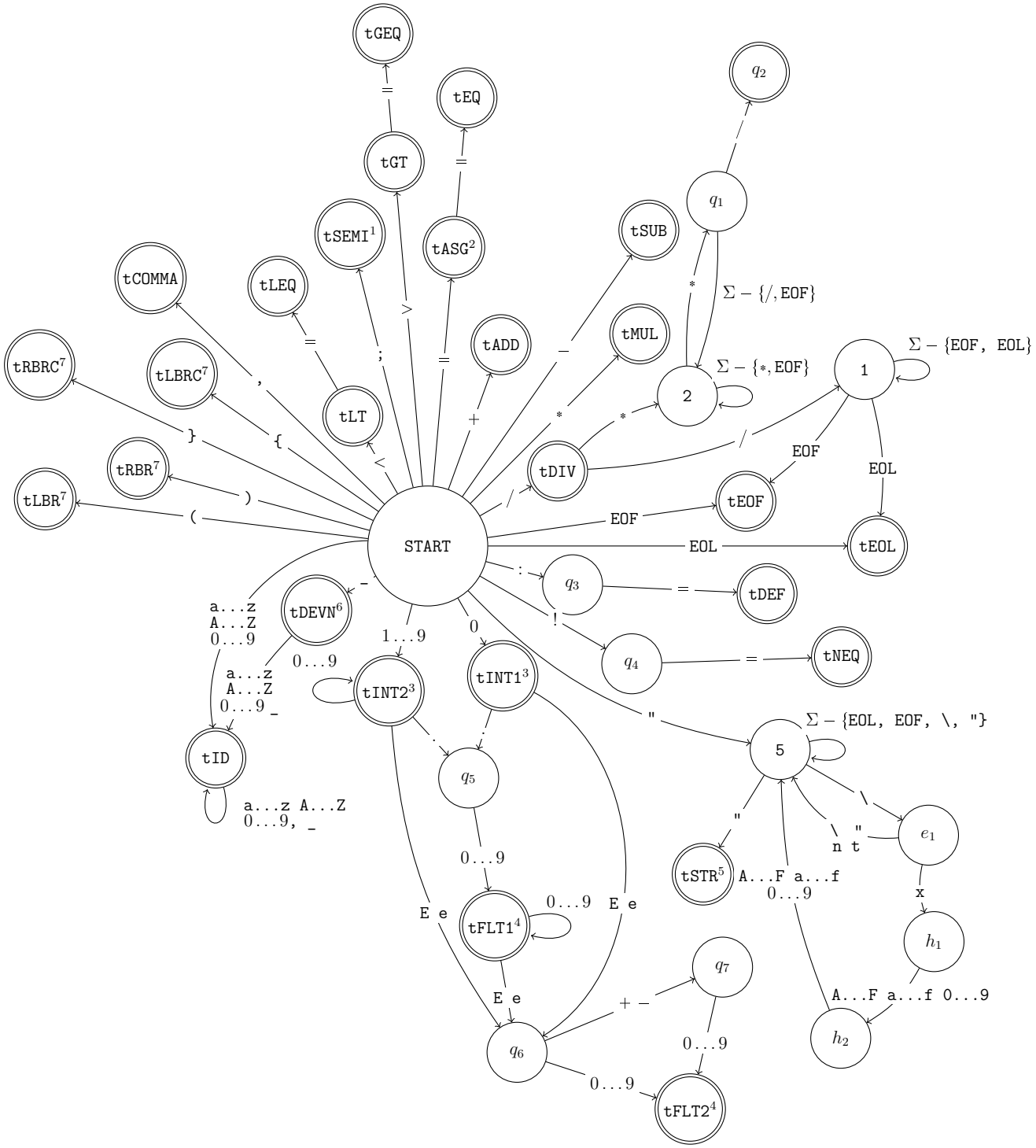
Princip převodu z notace *infix* na *postfix* byl nastudován zde [1]. Žádný kód nebyl převzat. Redukce je prováděna na základě následujících pravidel:

```
                (E)  →  E
E + E  →  E          E - E   →  E
E / E  →  E          E * E   →  E
E < E  →  E          E > E   →  E
E >= E →  E          E <= E  →  E
E == E →  E          E != E  →  E
```

3.3 Generování kódů - `generator.c`

První funkce generátoru, jejímž úkolem je vypsání vestavěných funkcí a funkce pomocné, je volána již na začátku parseru,a to hned poté co se ověří lexikální analýza. Průběžné vypisování kódu je uskutečněno voláním jednotlivých funkcí generátoru. Tyto funkce jsou zakomponovány do struktury překladače, což jim umožňuje využívat informace potřebné pro správnou funkci výsledného kódu. Speciálním případem je zde tisk výrazů, kdy se funkci předá odkaz na vrchol seznamu tokenů výrazu v postfixové notaci. ty jsou pak zpracovány strukturou funkce v generátoru. Díky postfixové notaci je pak značně zjednodušen tisk výrazů pomocí zásobníkových instrukcí.

4 Konečný automat



Obrázek 1: Deterministický konečný stavový automat[8]

[1] `tSEMI` změněno z `tSEMICOLON`
[2] `tASG` změněno z `tASSIGN`
[3] `tINT1` a `tINT2` obojí reprezentuje token `tINT`
[4] `tFLT1` a `tFLT2` obojí reprezentuje token `tFLOAT`
[5] `tSTR` změněno z `tSTRING`
[6] `tDEVN` změněno z `tDEVNULL`
[7] `tLBR` změněno z `tLBRACKET`, obdobně pro `tRBR`, `tLBRC` změněno z `tLBRACE`
[8] Σ reprezentuje ASCII hodnoty větší než 32 včetně a ASCII hodnoty 10, 13 a `EOF`

5 LL gramatika

```
<START>         ->  package main EOL <SCEL>
<SCEL>          ->  func <PROG> EOL <SCEL>
<SCEL>          ->  EOL <SCEL>
<SCEL>          ->  EOF
<PROG>          ->  id ( <PARAMS_DEF> ) <RET> { <BODY> }
<BODY>          ->  eps
<BODY>          ->  return <VALUE_EXTRA>
<BODY>          ->  EOL <BODY>
<BODY>          ->  if <IF> EOL <BODY>
<BODY>          ->  for <FOR> EOL <BODY>
<BODY>          ->  id <ID> EOL <BODY>
<ID>            ->  ( <PARAMS> )
<ID>            ->  := <EXPR_TYPE>
<ID>            ->  <ID_MORE> = <ID_ASSIGN>
<ID_MORE>       ->  eps
<ID_MORE>       ->  , id <ID_MORE>
<EXPR_TYPE>     ->  <TYPE>
<EXPR_TYPE>     ->  <EXPR>
<EXPR_TYPE_M>   ->  eps
<EXPR_TYPE_M>   ->  , <EXPR_TYPE> <EXPR_TYPE_M>
<TYPE>          ->  id
<TYPE>          ->  INTEGER_VAL
<TYPE>          ->  FLOAT_VAL
<TYPE>          ->  STRING_VAL
<VALUE_EXTRA>   ->  <EXPR_TYPE> <EXPR_TYPE_M>
<ID_ASSIGN>     ->  <VALUE_EXTRA>
<ID_ASSIGN>     ->  id ( <PARAMS> )
<DATA_TYPE>     ->  int
<DATA_TYPE>     ->  string
<DATA_TYPE>     ->  float64
<PARAMS>        ->  eps
<PARAMS>        ->  <TYPE> <TYPE_MORE>
<TYPE_MORE>     ->  eps
<TYPE_MORE>     ->  , <TYPE> <TYPE_MORE>
<PARAMS_DEF>    ->  eps
<PARAMS_DEF>    ->  id <DATA_TYPE> <PARAMS_DEF_M>
<PARAMS_DEF_M>  ->  eps
<PARAMS_DEF_M>  ->  , id <DATA_TYPE> <PARAMS_DEF_M>
<RET>           ->  eps
<RET>           ->  ( <RET_BODY> )
<RET_BODY>      ->  eps
<RET_BODY>      ->  <DATA_TYPE> <RET_BODY_M>
<RET_BODY_M>    ->  eps
<RET_BODY_M>    ->  , <DATA_TYPE> <RET_BODY_M>
<IF>            ->  <EXPR> { <BODY> } else { <BODY> }
<FOR>           ->  <EXPR_TYPE> ; <EXPR> ; <EXPR> { <BODY> }
```

Obrázek 2: LL gramatika

6 LL tabulka

	package main	EOL	EOF	func	id	(	)	{	}	return	if	for	:=	=	,	INTEGER_VAL	FLOAT_VAL	STRING_VAL	int	string	float64	;	$
<START>	1																						
<SCEL>		3	4	2																			
<PROG>					5																		
<BODY>		8			11				6	7	9	10											6
<ID>						12							13		14								
<ID_MORE>														15	16								15
<EXPR_TYPE>					17											17	17	17					
<EXPR_TYPE_M>		19							19						20								19
<TYPE>					21											22	23	24					
<VALUE_EXTRA>					25											25	25	25					
<ID_ASSIGN>					27											26	26	26					
<DATA_TYPE>																			28	29	30		
<PARAMS>					32		31									32	32	32					31
<TYPE_MORE>							33								34								33
<PARAMS_DEF>					36		35																35
<PARAMS_DEF_M>							37								38								37
<RET>						40		39															39
<RET_BODY>							41												42	42	42		41
<RET_BODY_M>							43								44								43
<IF>																							
<FOR>					46											46	46	46					

Obrázek 3: LL tabulka

7 Zdroje

[1] academy, R.: Infix, Prefix and Postfix expressions. [online]. Dostupné z: https://runestone.academy/runestone/books/published/pythonds/BasicDS/InfixPrefixandPostfixExpressions.html

[2] Niemann, T.: Operator precedence parsing. [online]. Dostupné z: https://epaperpress.com/oper/download/OperatorPrecedenceParsing.pdf