Projektová dokumentace

Implementation of Imperative Language ”IFJ22”Compiler

Tým xalaka00, varianta BVS

Herman Isaichanka	(xisaic00)	5 %
Michal Diachenko	(xdiach00)	10 %
Kambulat Alakaev	(xalaka00)	45 %
Andrei Kulinkovich	(xkulin01)	40 %

7. prosince 2022

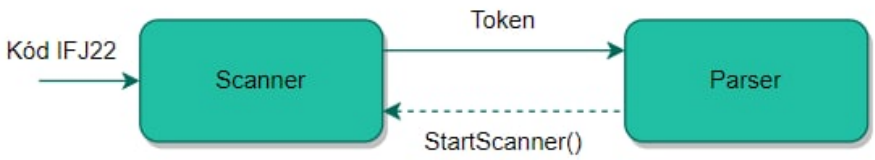
Obsah

1 Úvod

Cílem projektu bylo vytvořit překladač v jazyce C. Překladač načítá řídicí program v jazyce IFJ22(podmnožina jazyka PHP) ze standardního vstupu a musí genirovat výsledný mezikód na standardní výstup. V případě chyby, ukončí kompilaci s odpovídajícím navratovým kódem chyby.

2 Návrh a implementace

Projekt se skládá z několika implementovaných částí: skanner a parser. Jednotlivé části překladače a jejich chování budou detailně popsány v následujících kapitolách.



2.1 Lexikální analyzátor

Skanner na vstup přijímá kód IFJ22, postupně čte ho, a převádí posloupnost symbolu na datovou strukturu `token`, pak každý token postupně předává do parseru. Před čtením tokenů skanner jednou zkontroluje kód na existenci prologu a po čtení tokenů zkontroluje kód na vyskyt volitelné uzavírací značky. Pokud není prolog, pak vrátí lexikální chybu. Existuje několik typů tokenů: EOF, neznámý token, identifikátor, klíčové slovo, číslo typu float, číslo typu int, string, název funkce, operátory, složená otevírací závorka, složená uzavírací závorka, kulatá otevírací závorka, kulatá uzavírací závorka, středník, čárka, dvojtečka.

Před implementací skanneru byl vyvinut deterministický konečný automat, který se chová takto: Skanner probíhá vstupní kód symbol po symbolu a používá pravidla pro převod do různých stavů. Každý konečný stav znamená, že byl načten jeden token. Pokud přijde symbol, který nevede z jednoho stavu(neukončujícího) do dalšího stavu to znamená, že vstup je nesprávný, dokončujeme program s lexikální chybou.

2.2 Parser

2.2.1 Syntaktický analyzátor

Pro náš překlad jsme zvolili metodu rekurzivního sestupu. Syntaktický analyzátor(shora dolů) dostává token ze skanneru postupně a na základě typu a atributu použije pravidla pro přechod. Pokud nějaký token nevyhovuje žádnému z pravidel, pak analyzátor vrátí syntaktickou chybu. Pravidla pro přechody jsou v LL-gramatice a LL-tabulce. Když syntaktický analyzátor dosáhne místa, kde je nějaký výraz, pak předá řízení analyzátoru zdola nahoru.

Během provozu analyzátor převádí data z tokenů na datovou strukturu `tabulka symbolů`.

2.2.2 Sémantický analyzátor

Sémantický analyzátor kontroluje dva typy chyb. První typ chyby je sémantická chyba – nedefinovaná funkce, pokus o redefinice funkce. Druhý typ chyby je sémantická chyba – špatný počet parametrů u volání funkce. Zpracování běžných chyb se provádí na základě práce s datovou strukturou `tabulka symbolů`.

3 Datové struktury

3.1 Tabulka symbolů

Tabulka symbolů je datová struktura, která implementována na základě binárního vyhledávacího stromu, každá položka kterého je dynamické alokovaná. Položka obsahuje název funkce(klíč), počet její parametrů, a ukazatele na levý a pravý podstrom.

3.2 Zásobník

Používáme datovou strukturu zásobníku v precedenčním syntaktickém analyzátoru při zpracování výrazu pro vyhodnocování operací.

Položka zásobníku obsahuje symbol, který jsme získali během precedenční analýzy z tokenu. Náš zásobník je implementován na základě dynamického alokovaní paměti, což umožňuje realokaci a následné zvětšení kapacity v případě potřeby.

4 Členění implementačního řešení

Lexikální analyzátor	`scanner.{c,h}`
Syntaktický analyzátor	`parser.{c,h}`, `expressions.{c,h}`
Semantický analyzator	`parser.{c,h}`, `expressions.{c,h}`
Tabulka symbolů	`symtable.{c,h}`
Zásobník symbolů	`symstack.{c,h}`
Funkce main	`main.c`

5 Práce v týmu

5.1 Komunikace

Komunikace mezi členy týmu probíhala hlavně na soukromém serveru Discord vyhrazeném pro projekt, kde jsme vytvořili kanály pro konkrétní témata a snažili se být v kontaktu. Měli jsme také osobní schůzky, jejichž hlavním cílem bylo rozhodnout, co je třeba udělat, kdo a jak to má udělat.

5.2 Verzovací systém

Pro paralelní vývoj jsme použili verzovací systém Git a vzdálený repozitář byl vytvořen na serveru Github. Pro efektivní práci by si každý mohl vytvořit vlastní pobočku pro část projektu, na které právě pracují.

5.3 Rozdělení práce mezi členy týmu

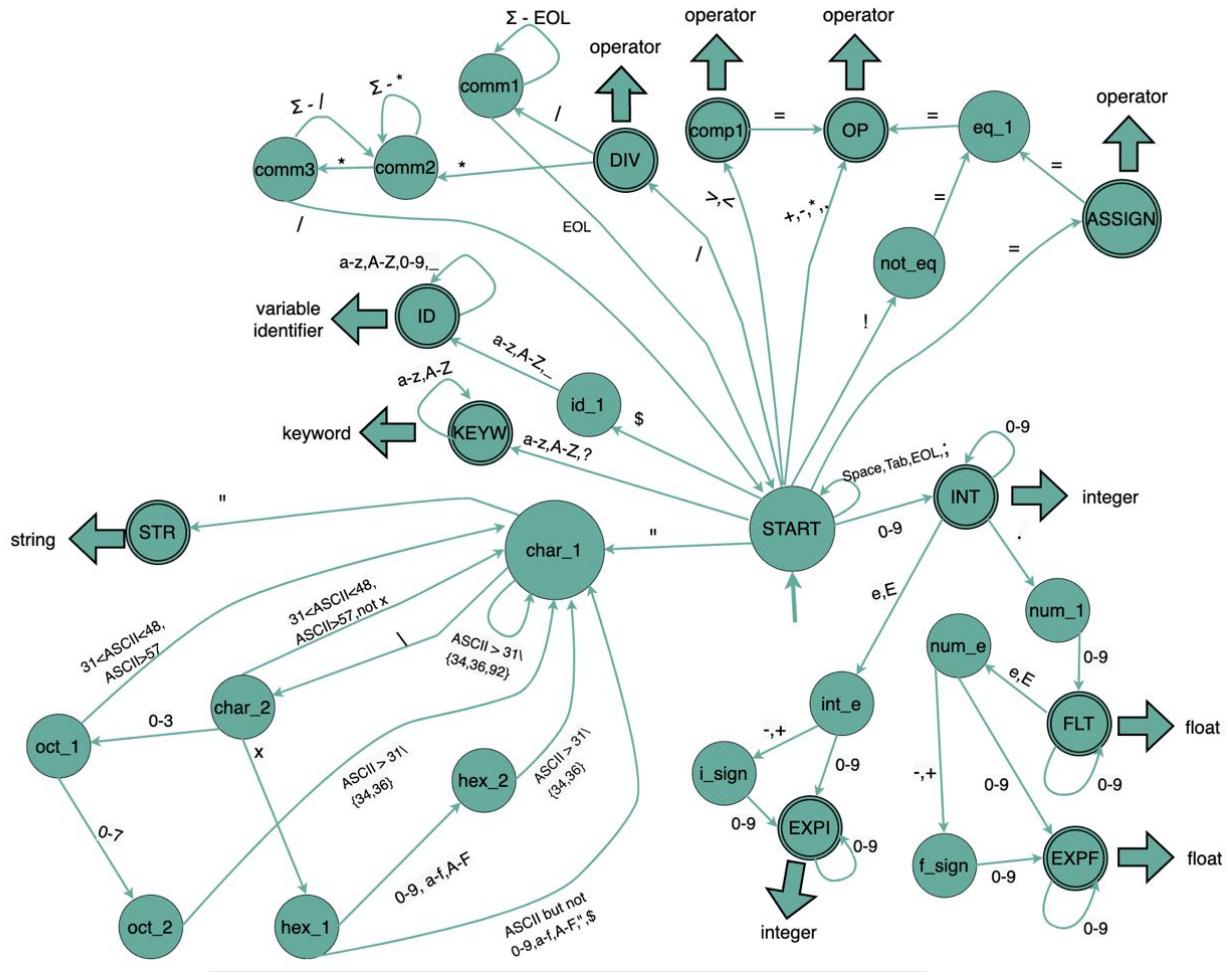
Práci na projektu jsme si rozdělili rovnoměrně na základě jeho složitosti a časové náročnosti.

Člen týmu	Přidělená práce
Herman Isasichanka	Scanner
Michal Diachenko	Scanner, git správa projektu/kontrola verzí
Kambulat Alakaev	Scanner, parser,vedení týmu, vytvoření LL-Gramatiky, vytvoření DKA, návrh prec. tabulky
Andrei Kulinkovich	Scanner, lehký parser, vytvoření LL-Gramatiky, vytvoření DKA, dokumentace, prezentace k projektu

6 Token

```
typedef struct Token
{
    tokenType type; // typ tokenu
    char *attribute; // atribut tokenu je reprezentovan jako    etzec
} token_t;
```

7 Deterministický KA



8 LL-Gramatika

```
(1)  prog → prolog body ending
(2)  prolog →'<?php' 'declare(strict_types=1)' ';'
(3)  body → ε
(4)  body → code_block body
(5)  ending → ε
(6)  ending → '?>'
(7)  code_block → statement
(8)  code_block → def_function
(9)  def_function → 'function' func_name '(' param_dec_list_1 ')' ':' type '{' statements '}'
(10) param_dec_list_1 → ε
(11) param_dec_list_1 → type id param_dec_list_2
(12) param_dec_list_2 → ε
(13) param_dec_list_2 → ',' type id param_dec_list_2
(14) type → '?int'
(15) type → 'int'
(16) type → '?string'
(17) type → 'string'
(18) type → 'float'
(19) type → '?float'
(20) type → 'void'
(21) statement → expression ';'
(22) statement → id '=' expression ';'
(23) statement → 'return' opt_expr ';'
(24) statement → 'if' '(' expression ')' '{' statements '}' 'else' '{' statements '}'
(25) statement → 'while' '(' expression ')' '{' statements '}'
(26) statement → function_call ';'
(27) statements → ε
(28) statements → statement statements
(29) opt_expr → ε
(30) opt_expr → expression
(31) function_call → func_name '(' param_list ')'
(32) param_list → ε
(33) param_list → arg arg_list
(34) arg → expression
(35) arg_list → ε
(36) arg_list → ',' arg arg_list
```

9 LL-Tabulka

	<?php declare(strict_types=1);	;	?>	function	func_name	(	)	:	{	}	id	,	?int	Int	?string	string	float	?float	void	=	return	If	while	expression	EOF
prog	1																								
prolog	2																								
body			3	4	4						4										4	4	4	4	3
ending			6																						5
code_block				8	7						7										7	7	7	7	
statement					26						22										23	24	25	21	
def_function				9																					
param_dec_list_1							10						11	11	11	11	11	11	11						
param_dec_list_2							12					13													
type													14	15	16	17	18	19	20						
opt_expr		29																						30	
statements					28					27	28										28	28	28	28	
function_call					31																				
param_list							32																	33	
arg																								34	
arg_list							35					36													

10 Tabulka symbolů

```
typedef struct sym_bst_node
{
    char *key;                  // name for function
    int num_of_params;          // count of parameters in a function
    struct sym_bst_node *left;  // left son
    struct sym_bst_node *right; // right son
} SymTabNode_t;
```

11 Zásobník

```
typedef struct stack_i
{
    PrecTabSymbEnum symbol; /// Symbol of stack item.
    struct stack_i *next; /// Pointer to next stack item.
} SymbStackItem;

typedef struct
{
    SymbStackItem *top; /// Pointer to stack item on top of stack.
} SymbStack;
```

12 Precedenční tabulka

	+	-	*	/	.	>	<	>=	<=	===	!==	(	)	i	$
+	>	>	<	<	>	>	>	>	>	>	>	<	>	<	>
-	>	>	<	<	>	>	>	>	>	>	>	<	>	<	>
*	>	>	>	>	>	>	>	>	>	>	>	<	>	<	>
/	>	>	>	>	>	>	>	>	>	>	>	<	>	<	>
.	>	>	<	<	>	>	>	>	>	>	>	<	>	<	>
>	<	<	<	<	<							<	>	<	>
<	<	<	<	<	<							<	>	<	>
>=	<	<	<	<	<							<	>	<	>
<=	<	<	<	<	<							<	>	<	>
===	<	<	<	<	<							<	>	<	>
!==	<	<	<	<	<							<	>	<	>
(	<	<	<	<	<	<	<	<	<	<	<	<	=	<	
)	>	>	>	>	>	>	>	>	>	>	>		>		>
i	>	>	>	>	>	>	>	>	>	>	>		>		>
$	<	<	<	<	<	<	<	<	<	<	<	<		<	