Projektová dokumentace

Implementace překladače jazyka IFJ19

Tým 109, varianta I

Členové týmu:
Tomáš Beránek (xberan46) 25%
Martin Haderka (xhader00) 25%
Richard Klem (xklemr00) 25%
Šimon Slobodník (xslobo06) 25%

Contents

1 Úvod

Cílem projektu bylo v jazyce C implementovat překladač jazyka IFJ19, který je zjednodušenou podmnožinou jazyka Python3. A následně jej přeložit do cílového jazyka IFJcode19 s využitím teroií automatů, rekurzivního sestupu a precedenční syntaktické analýzy.

2 Implementace překladače

Funkcionalita překladače je rozdělena na několik podčástí, kterými se zabývají jednotlivé podkapitoly. Při implemetaci byly využity zdrojové soubory z předmětu IAL (implementace BVS, zásobník, jednosměrně/dvousměrně vázaný seznam a převod infixové na postfixovou notaci).

Všechny analýzy a generování probíhá současně (kromě sémentické kontroly datových typů a dělení nulou, které probíhá při interpretaci).

2.1 Lexikální analyzátor

Úkolem lexikálního analyzátoru je projít celým vstupním souborem a zkontrolovat, zda neobsahuje lexikální chyby (např.: chybný formát identifikátorů, čísel, řetězců, atd.) a následně vygenerovat odpovídající tokeny, pro další zpracování. Nejdříve byl vytvořen návrh deterministického konečného automatu (DKA(viz. Obrázek 1). Implementace je v souboru IFJ_scanner.c s rozhraním v IFJ_scanner.h.

Hlavím výstupem lexikálního analyzátoru je struktura token_t, která obsahuje typ tokenu (definovaný výčtovým typem token_type) a jeho případnou hodnotu (ta je uchovávána pomocí struktury union, kvůli úspoře paměti). Token je možné získat pomocí funkce get_token. DKA je implementován pomocí konstrukce switch-case, která přepíná mezi jednotlivými stavy (ty jsou definovany výčtovým typem states).

Zajímavým problémem bylo načítání znaků (např. řetězce) ze standartního vstupu do dynamické paměti, když nebyla dopředu známa velikost načítaných dat. Aby nebylo nutné vytvářet abstraktní datový typ dynamicky alokovaného pole proměnné velikosti, tak se nejdříve přečtou všechny znaky, které bude potřeba uložit a zapamatuje se jejich počet. Poté se posune offset zpět o přislušný počet znaků. A do předem připraveného pole (už o známé velikosti), se znaky uloží.

2.2 Syntaktický analyzátor

Syntaktický analyzátor (SA) pomocí LL-gramatiky (LL-tabulky a LL-pravidel - viz. Obrázek 2 a Obrázek 3) kontroluje, zda je vstupní kód syntakticky správný (tzn. jestli jsou tokeny správně seřazeny).

SA je implementována rekurzivním sestupem a precedenční syntaktickou analýzou (PSA), která zpracovává výrazy. Následující token je uchováván v globální proměné next_token. PSA je implementována jako funkce expressionParse, které je předáno řízení vždy, když je rozpoznán výraz. expressionParse vrací rekurzivnímu sestupu další token, aby mohla SA pokračovat. V precedenční tabulce jsou kromě povinných pravidel i rozpracovaná pravidla pro zpracování přednosti funkce. K vypracování rozšíření FUNEXP nedošlo, proto se v tabulce vyskutuje nedefinovaná hodnota znázorněna jako XXXX.

Největší výzvou bylo vytvoření deterministické tabulky pravidel. Bylo třeba využít metody faktorizace pro odstranění nedeterminismu v pravidlech a ošetření veškerých možných situací.

2.3 Sémantický analyzátor

Část sémantického analyzátoru kontrolující definice/redefinice funkcí/proměnných funguje na principu zásobníku, který interně komunikuje s tabulkou symbolů (viz Obrázek 4). Tabulka symbolů je implementována v souboru *symtable.c* pomocí binárního vyhledávacího stromu, kde jako klíč slouží jméno daného identifikátoru. Tabulka symbolů je pouze jedna (nerozlišuje se lokální a globální), místo toho se používají informace ve struktuře *Record*, které určují zda je identifikátor globální/lokální, definovaný/nedefinovaný, atd.

Část kontrolující kompatibilitu datových typů a dělení nulou je implementována jako funkce *$do_operation* v jazyce IFJcode19 v *IFJ_builtin.c*. Funkce za běhu kontroluje kompatibilitu operandů všech operacích. Funkce je vždy vložena do hlavičky každého vygenerovaného souboru jazyka IFJcode19.

2.4 Generování kódu

Generování výsledného mezikódu probíhá v souborech *IFJ_builtin.c* (vygenerování hlavičky, vestavěných funkcí včetně funkce na kontrolu datových typů a generování

výrazů), *IFJ_stack_semantic.c* (definice proměnných) a *IFJ_parser.c* (řídící kontrukce a definice funkcí). Generování probíhá zároveň se syntaktickou analýzou (jedná se o syntaxí řízený překlad). Mezikód je generován na standardní výstup. Jelikož je generován průběžně, při chybě již vypsaná část programu zůstane na standardním výstupu, a je vypsána příslušná chybová hláška na chybový výstup.

Funkce *$do_operation* zpracovává veškeré operace ve výrazech pomocí zásobníku dostupného v IFJcode19, ze kterého čte operandy a operace v postfixové notaci, které byly předány jako pole tokenů od PSA. Funkce zároveň provádí kontrolu a potřebné přetypování operandů pro aritmetické operace. V případě chybných datových typů a dělení nulou vypíše příslušný kód chyby.

Jelikož bylo cíleno na jedno-průchodovou analýzu, bylo nutné vyřešit problém s možnou vícenásobnou definicí u cyklů nebo naopak s chybějící definicí u příkazu větvení. Problém je vyřešen pomocí globální proměnné a zásobníku instrukcí. Globální proměnná je nastavena na pravdivostní hodnotu pokud se SA nachází v cyklu nebo příkazu větvení a v takovém případě jsou všechny instrukce, které by se normálně tiskly na standartní výstup, ukládány na zásobník a jsou tisknuty pouze definice proměnných. Jakmile se vyskočí ven z nejvyšší vrstvy cyklu nebo příkazu větvení, jsou všechny instrukce vypsány ze zásobníku na standartní výstup.

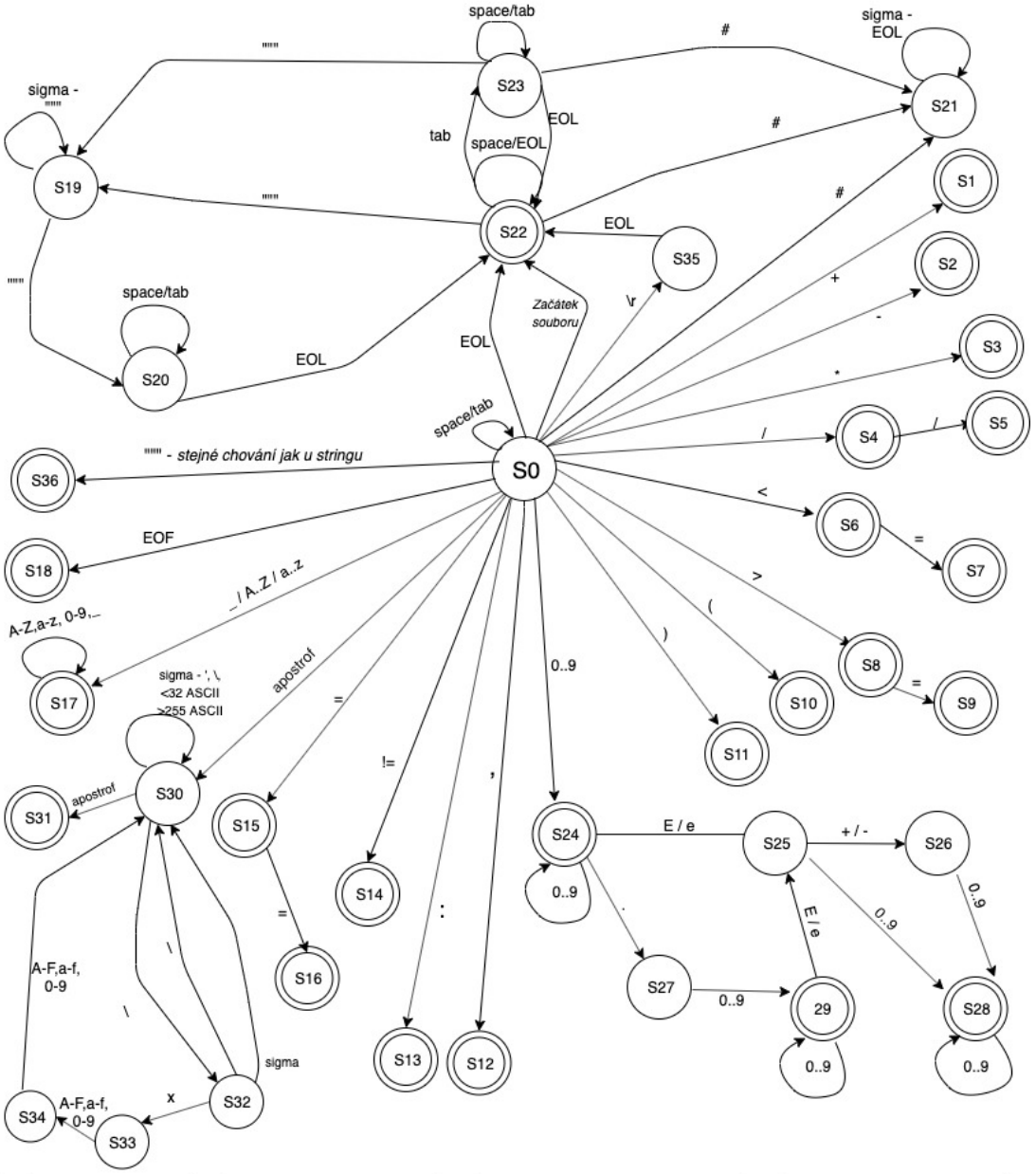
3 Metodika vývoje

Při vývoji byla využita agilní vývojová metodika scrum s pravidelnými sprinty o délce jednoho týdne doplněná o mimořádné schůzky.

4 Komunikační kanály a verzovací systémy

Ke komunikaci byl využit Slack v kombinaci s Facebook Messengerem a Skypem pro videohovory a sdílení obrazovky, přednostně se však využívalo pravidelných osobních setkání.

Jako verzovací systém byl využit git s hostingem na github.com. Propojili jsme github s aplikací Slack pro notifikaci commitů, aby bylo možné okamžitě reagovat a kontrolovat změny ostatních.



S0	STATE_START	**S10**	STATE_LEFT_BRACKET	**S20**	STATE_MULTI_LINE_COMM_AFTER	**S30**	STATE_STRING
S1	STATE_PLUS	**S11**	STATE_RIGHT_BRACKET	**S21**	STATE_SINGLE_LINE_COMM	**S31**	STATE_STRING_READ
S2	STATE_MINUS	**S12**	STATE_COMMA	**S22**	STATE_INDENT_DEDENT	**S32**	STATE_STRING_BACKSLASH
S3	STATE_MUL	**S13**	STATE_COLON	**S23**	STATE_EMPTY_LINE	**S33**	STATE_STRING_HEX_START
S4	STATE_DIV	**S14**	STATE_NOT_EQ	**S24**	STATE_INT	**S34**	STATE_STRING_HEX_END
S5	STATE_INT_DIV	**S15**	STATE_ASSIGNMENT	**S25**	STATE_DOUBLE_EXP	**S35**	STATE_CRLF
S6	STATE_LESS	**S16**	STATE_EQ	**S26**	STATE_DOUBLE_EXP_SIGN	**S36**	STATE_MULTI_LINE_LITERAL
S7	STATE_LESS_EQ	**S17**	STATE_ID	**S27**	STATE_DOUBLE_DECIMAL_POINT		
S8	STATE_GREATER	**S18**	STATE_EOF	**S28**	STATE_DOUBLE_WITH_EXP		
S9	STATE_GREATER_EQ	**S19**	STATE_MULTI_LINE_COMM	**S29**	STATE_DOUBLE_WITHOUT_EXP		

Obrázek 1: Konečný automat lexikálního analyzátoru

LL-pravidla

1. <prog> -> <eol-opt> <st-list> EOF
2. <st-list> -> <stat> <eol-opt> <st-list>
3. <stat> -> PASS
4. <st-list> -> ε
5. <stat> -> ID <expr-or-assign> EOL
6. <expr-or-assign> -> = <fun-or-expr>
7. <return> -> RETURN <expr> EOL
8. <return> -> ε
9. <eol-opt> -> EOL <eol-opt>
10. <eol-opt> -> ε
11. <stat> -> DEF ID (<param-list>) : EOL <eol-opt> INDENT <st-list> <return> DEDENT
12. <param-list> -> ID <param-next>
13. <param-list> -> ε
14. <param-next> -> ε
15. <param-next> -> , ID <param-next>
16. <stat> -> IF <expr> : EOL <eol-opt> INDENT <st-list> DEDENT ELSE : EOL <eol-opt> INDENT <st-list> DEDENT
17. <stat> -> WHILE <expr> : EOL <eol-opt> INDENT <st-list> DEDENT
18. <expr-or-assign> -> (+, -, *, /, //, <, <=, >, >=, !=, ==) <expr>
19. <stat> -> (STR, INT, DBL, (, NONE) <expr> EOL
20. <fun-or-expr> -> ID <fun-or-expr2>
21. <fun-or-expr-2> -> (<arg-list>)
22. <fun-or-expr> -> (STR, INT, DBL, (, NONE) <expr>
23. <fun-or-expr-2> -> (+, -, *, /, //, <, <=, >, >=, !=, ==) <expr>

Obrázek 2: LL-pravidla

LL-tabluka

	+	-	/	*	//	<	<=	>	>=	str	dbl	int	!=	==	none	id	(	)	eol	,	:	indent	dedent	=	def	else	while	pass	return	if	eof
<prog>										1	1	1			1	1	1		1						1		1	1		1	1
<st-list>										2	2	2			2	2	2						4		2		2	2	4	2	4
<stat>										19*	19*	19*			19*	5	19*								11*		17	3		16	
<eol-opt>										10	10	10			10	10	10		9			10	10		10		10	10	10	10	10
<return>																							8						7		
<param-list>																12		13													
<param-next>																		14		15											
<expr-or-assign>	18*	18*	18*	18*	18*	18*	18*	18*	18*				18*	18*			24		29					6							
<arg-list>										25*	25*	25*			25*	25*		26													
<arg-next>																		27		28*											
<fun-or-expr>										22*	22*	22*			22*	20	22*		29												
<fun-or-expr-2>	23*	23*	23*	23*	23*	23*	23*	23*	23*				23*	23*			21		30												

Obrázek 3: LL-tabulka

	+-	**// /***	**(**	**)**	**relační op.**	**funckce**	**proměnná**	**$**
+-	REDUCE	SHIFT	SHIFT	REDUCE	REDUCE	XXXX	SHIFT	REDUCE
// /*	REDUCE	REDUCE	SHIFT	REDUCE	REDUCE	XXXX	SHIFT	REDUCE
(	SHIFT	SHIFT	SHIFT	EQUAL	SHIFT	SHIFT	SHIFT	ERROR
)	REDUCE	REDUCE	ERROR	REDUCE	REDUCE	XXXX	ERROR	REDUCE
relační op.	SHIFT	SHIFT	SHIFT	REDUCE	ERROR	XXXX	SHIFT	REDUCE
funkce	ERROR	ERROR	EQUAL	ERROR	ERROR	ERROR	ERROR	REDUCE
proměnná	REDUCE	REDUCE	ERROR	REDUCE	REDUCE	ERROR	ERROR	REDUCE
$	SHIFT	SHIFT	SHIFT	ERROR	SHIFT	SHIFT	SHIFT	ERROR

Obrázek 4: PSA tabulka