Implementace předkladače imperativního jazyka IFJ17

Tým 034, varianta II

Daniel Bílý (xbilyd01)	**25 %**
Jakub Gajdošík (xgajdo24)	25 %
Petr Marek (xmarek69)	25 %
Jan Fridrich (xfridr07)	25 %

Popis návrhu částí překladače a předávání informací mezi nimi

Lexikální analyzátor (scanner)

V tomto projektu jsme začali implementací syntaktického analyzátoru. Ten pomocí své hlavní funkce *get_token(token_t *token)* vždy ukládá další token ze vstupního (zdrojového) souboru do svého parametru. Pomocí této funkce scanner komunikuje se sémantickým analyzátorem, který má svou vlastní (globální) proměnnou, do níž scanner ukládá načtený token. Token se skládá z typu a atributu.

Do typu tokenu ukládáme číselnou konstantu, přičemž rozlišujeme mezi těmito typy tokenů: číslo (celé, nebo desetinné), identifikátor, klíčové slovo, řetězec, aritmetický nebo relační operátor, závorka, operátor přiřazení, čárka, znak konce řádku a znak konce souboru (EOF). Do atributu tokenu se uloží hodnota pouze v případě, že je typ tokenu něco z následujících: číslo (celé, nebo desetinné), identifikátor, klíčové slovo, řetězec. Ostatní typy tokenů jsou definovány pouze svým typem.

Náš lexikální analyzátor pracuje na základě přiloženého diagramu konečného automatu. Vždy se po zavolání funkce *get_token()* nastaví počáteční stav a program vstoupí do nekonečného cyklu while. V tomto cyklu se přepíná mezi jednotlivými stavy pomocí switch, kterému jako podmínku předáváme nastavený stav (proměnnou, která uchovává číselnou konstantu představující tento stav). Postupně se načítají znaky ze vstupního souboru, dokud není načten celý lexém. Mezi stavy se přepíná podle načítaných znaků. V této funkci jsou však dvě zvláštnosti, a to tehdy, když načítáme řádkový nebo blokový komentář.

V případě, že scanner narazí na řádkový komentář, celý jej přeskočí (pakliže následují další řádkové komentáře, přeskakuje i je) a vrátí následující token. Přeskočení komentáře se děje tak, že automat cyklí ve stavu určeném pro řádkový komentář a vždy načítá další znak tak dlouho, dokud nenarazí na znak nového řádku. Tehdy nastaví proměnnou reprezentující stav opět na počáteční. Z tohoto stavu snajde a vrátí další token (ležící pod komentářem).

Pokud se v kódu vyskytne blokový komentář, lexikální analyzátor jej taktéž přeskočí. Chtěli jsme zde však použít jednoduchý a kreativní způsob, a tak jsme zvolili následující řešení:

Při zavolání funkce *get_token()* se ve skutečnosti ještě před switch načte znak ze vstupního souboru. Pokud se jedná o odřádkování (tak může následovat blokový komentář) nebo o úplně první znak vstupního souboru, pak zkontroluje, jestli tento znak spolu s následujícími neoznačuje začátek blokového komentáře („=begin “). Pokud ano, tak se

přeskakují znaky, dokud lexikální analyzátor nenarazí na označení konce blokového komentáře („=end“). Teprve až po těchto kontrolách se nachází dříve zmiňovaný switch, rozlišující a přepínající mezi jednotlivými stavy automatu. Použití tohoto mechanismu přeskakování blokového komentáře je důvodem, proč jej nemáme zaznačený jako stav ani v diagramu konečného automatu.

Syntaktický analyzátor (parser)

Druhým blokem překladače, který jsme museli implementovat je syntaktický analyzátor. Při jeho konstrukci jsme zvolili metodu rekurzivního sestupu. Syntaktický analyzátor obsahuje hlavní funkci program(), která je volána z hlavní funkce překladače main() pro první token vstupního souboru a poté funkce program() rekurzivně volá sama sebe až dojde k znaku END OF FILE a potom se vrátí zpět do main()

Zde jsme řešili problém, Z funkce program() totiž jde rekurzivně přejít do definice funkce, která musí být na začátku libovolného řádku, ale musíme zajistit, abychom nedefinovali funkci v případě, že již v definici funkce jsme (definice funkce v těle definice funkce je chyba). Ke kontrole, zda se tedy nenacházíme v definici funkce používáme pomocnou globální proměnnou „uvnitr_funkce“, díky níž rozlišujeme, zda můžeme definovat novou funkci nebo překladač má uvolnit paměť , vrátit chybu a ukončit se.

Ve funkci program() načítáme další tokeny ze vstupního souboru a voláme některou z funkcí na základě načteného typu tokenu. Zde rozlišujeme mezi tím, jestli je na řádku vstupního souboru výraz, identifikátor, definice funkce, cyklus, volání vestavěné funkce, nebo podmínka. V těchto jednotlivých funkcích voláme další funkce, které simulují konstrukci derivačního stromu. Pokud se ve zdrojovém kódu nachází nějaký token, který nemůže následovat, je vyhodnocena chyba. Výrazy zpracováváme jinak než zbytek kódu – precedenční syntaktickou analýzou. Tato analýza využívá zásobníky implementované v souboru *stack.c*. Podle daných pravidel ukládá znaky na zásobník, případně redukuje výraz na zásobníku a v případě redukce, při provádění operace se volá generátor pro vygenerování příslušné operace v cílovém souboru.

Sémantiku řešíme z většiny při syntaktické analýze až na typovou kontrolu, která je řešena v generátoru.

Generátor

Generátor je implementován jako samostatný soubor (generator.c), který obsahuje funkce pro generaci jednotlivých instrukcí, popřípadě bloků instrukcí.

Po zavolání generátoru parserem se generuje určená instrukce kódu IFJcode18. Generovaný kód jsme se rozhodli nejprve vkládat do dvou řetězců a až po ukončení překladu tyto dva řetězce posílat na výstup. Tímto řešíme problém toho, že zdrojový kód IFJ18 nemá explicitně označené hlavní tělo programu. Do jednoho řetězce ukládáme definice uživatelských funkcí a do druhého vše ostatní (hlavní tělo programu).

Podobný princip jsme také použili při generování *while*, kdy se tělo while generuje do jiného stringu, který se při jeho konci spojí se stringem, do kterého jsme předtím generovali. Všechny definice proměných se provedou předsamotným while, tím že se průběžně budou zapisovat do stringu těla programu nebo stringu zbytku.

Do řetězců konkatenujeme jednotlivé instrukce pomocí maker, které zjednodušují práci s kódem a jeho přehlednost. Nejprve generujeme hlavičku souboru, která obsahuje úvodní řádek (IFJcode18), definici několika pomocných globálních proměnných a skok do hlavního těla programu.

Dále se generují vestavěné funkce, které později mohou být zavolány.Poté se generuje podle toho jak je generátor volán parserem.

Výpočet výrazů se provádí na zásobníku, výsledná hodnota může být uložena do globální proměnné. Hodnotu z globální proměnné poté můžeme dále používat pro vyhodnocování podmínek, její vypsání apod.

Rozdělení práce mezi členy týmu

Daniel Bílý – vedení týmu, implementace lexikálního analyzátoru a generátoru, diagram konečného automatu, implementace tokenů a práce s nimi, tvorba dokumentace

Jakub Gajdošík – debugging, implementace hash tabulky, správné ukončení programu

Petr Marek – implementace syntaktického a sémantického analyzátoru, implementace zásobníků, tvorba dokumentace

Jan Fridrich – implementace syntaktického a sémantického analyzátoru, implementace zásobníku, úprava hash tabulky

Způsob práce v týmu

Na začátku projektu jsme si vybrali verzovací systém Git, založili repozitář na Githubu a přistupovali do něj pomocí softwaru GitKraken. Komunikace probíhala většinou přes Facebook nebo osobně. S využitím systému větvení, který Git nabízí, jsme si práci rozdělili do různých větví. Obvykle jsme na jednotlivých problémech pracovali jednotlivě nebo ve dvojicích. Ve chvíli, kdy jsme se zastavili u větších problémů, všichni jsme se sešli a řešili

ho společně. Tato situace nastávala především u řešení komunikace mezi parserem a generátorem.

Datové struktury, algoritmy a vývojový cyklus

Struktura token_t – využívá pro uložení typu tokenu číslo (nějakou konstantu), jak již bylo uvedeno výše. V některých případech (podle typu tokenu) se ukládá také atribut tokenu a to do speciální struktury – union. V ní se ukládá atribut do jedné z proměnných: integer, decimal, string, keyword, nil. Když později (v generátoru) potřebujeme získat hodnotu tokenu, na základě typu tokenu rozhodneme, kterou proměnnou číst a následně přečteme její hodnotu.

Tabulka s rozptýlenými položkami – Používáme jako tabulku symbolů, kde ukládáme jednotlivé identifikátory a informace k nim. Je implementovaná jako struktura obsahující klíč (což je zahashovaný identifikátor), data (informace o identifikátorech) a odkaz na další položku.

Zpracování výrazů v syntaktickém analyzátoru – Řešíme precedenční syntaktickou analýzou. Čteme postupně výraz zleva doprava a na základě pravidel definovaných v precedenční tabulce provádíme tyto operace: posun (na zásobník vložíme znak posunu „<“ a dále na něj vložíme načtený operand/operátor), redukce (na vrcholu zásobníku provedeme redukci na základě definovaných pravidel a zavoláme generátor pro provedení příslušné operace).

Zásobníky – Při vyhodnocování výrazů používáme zásobník, který je implementovaný ve struktuře, která obsahuje pole celých čísel a číslo označující vrchol zásobníku. S tímto zásobníkem pracujeme při precedenční analýze, kdy si do něho ukládáme celá čísla (konstanty) reprezentující jednotlivé symboly z precedenční tabulky (příp. znak „<“). Dále využíváme zásobník stejného typu pro ukládání operátorů, abychom mohli později na základě operátoru rozhodnout, která operace se má v generátoru vygenerovat. V případě, že načítaný symbol výrazu je číslo/identifikátor/řetězec, do zásobníku uložíme konstantu reprezentující operand. V opačném případě uložíme konstantu příslušného operátoru. Kromě těchto dvou zásobníku používáme ještě jeden, ten je však jiného typu. Neobsahuje pole celých čísel, ale pole tokenů. Jinak se chová podobně jako předchozí dva zásobníky. Při vyhodnocování výrazů si do něj ukládáme tokeny (pouze operandy), a to za tím účelem, aby měl později generátor tyto operandy k dispozici, až bude provádět nějakou operaci s operandy.

Všechny tři zmíněné zásobníky si začínají ukládat hodnoty(resp. tokeny) od indexu 1, abychom při mazání zásobníku nedostávali záporedné hodnoty vrcholu zásobníku. Je to dáno způsobem naší implementace zásobníku.

Vývojový cyklus – V našem projektu jsme postupovali následovně: Nejprve jsme spolu založili a nastavili verzovací systém a dohodli se, jak bychom chtěli postupovat (rozdělení práce). Poté jsme začali implementací scanneru, na kterém jsme se částečně podíleli všichni, jelikož jsme ještě neměli úplnou představu, jak bude vypadat výsledný scanner, resp. jak bude komunikovat s ostatními částmi překladače a nevěděli jsme tedy, jak začít s ostatními částmi. Po dokončení scanneru jsme se pustili ve dvojicích do dalších částí – parseru a generátoru. Zde už jsme si byli jistější tím, jak budeme pracovat, co by měla každá část dělat apod., tudíž jsme mohli pracovat na projektu paralelně. Aby nebyl výsledný soubor (napsaný v jazyce C) příliš rozsáhlý a nepřehledný, vytvořili jsme samostatné soubory pro každý velký blok překladače – main, scanner, parser a generátor. Dále jsme také vytvořili soubory, v nichž jsme implementovali pomocné funkce a proměnné pro zmíněné hlavní soubory překladač (například samostatný soubor pro hash tabulku, zásobníky, pro práci s tokeny apod.). Nakonec, až jsme téměř dokončili parser i generátor, opět jsme se ve dvojicích zaměřili na správnou komunikaci mezi parserem a generátorem (a jejich korektní finální funkčnost), zatímco druhá dvojice začala s tvorbou dokumentace.

LL – gramatika

1.<program> -> <statement_list>
2.<statement_list> -> <statement> <end>
3.<end> ->EOL <statement_list>
4.<end> ->EOF

5.<statement> -> def FUNCTION_ID (<func_params>) EOL <body_list> end EOL<body>
6.<statement> -> <body_list>

7.<body_list> -> <body>EOL <body_list>
8.<body_list> -> <body_cycle>EOL <body_list>
9.<body_list> -> ε

10.<body_cycle> -> if EXPRESSION then EOL <body_list> else EOL <body_list> end
11.<body_cycle> -> while EXPRESSION do EOL <body_list> end
12.<body> -> ID = EXPRESSION
13.<body> -> EXPRESSION
14.<body> -> ID = <body>
15.<body> -> ε
16.<body> -> FUNCTION_ID <func_args>
17.<body> -> print <func_args>
18.<body> -> chr <func_args>
19.<body> -> ord <func_args>
20.<body> -> length <func_args>
21.<body> -> substr <func_args>
22.<body> -> inputs ()
23.<body> -> inputf ()
24.<body> -> inputi ()

25.<func_params> -> <params>
26.<func_args> -> <func_params>
27.<func_args> -> (<func_params>)

28.<params> -> ε
29.<params> -> <param> <params_list>
30.<params_list> -> <next_params><params_list>
31.<params_list> -> ε
32.<next_params> -> ,<param>

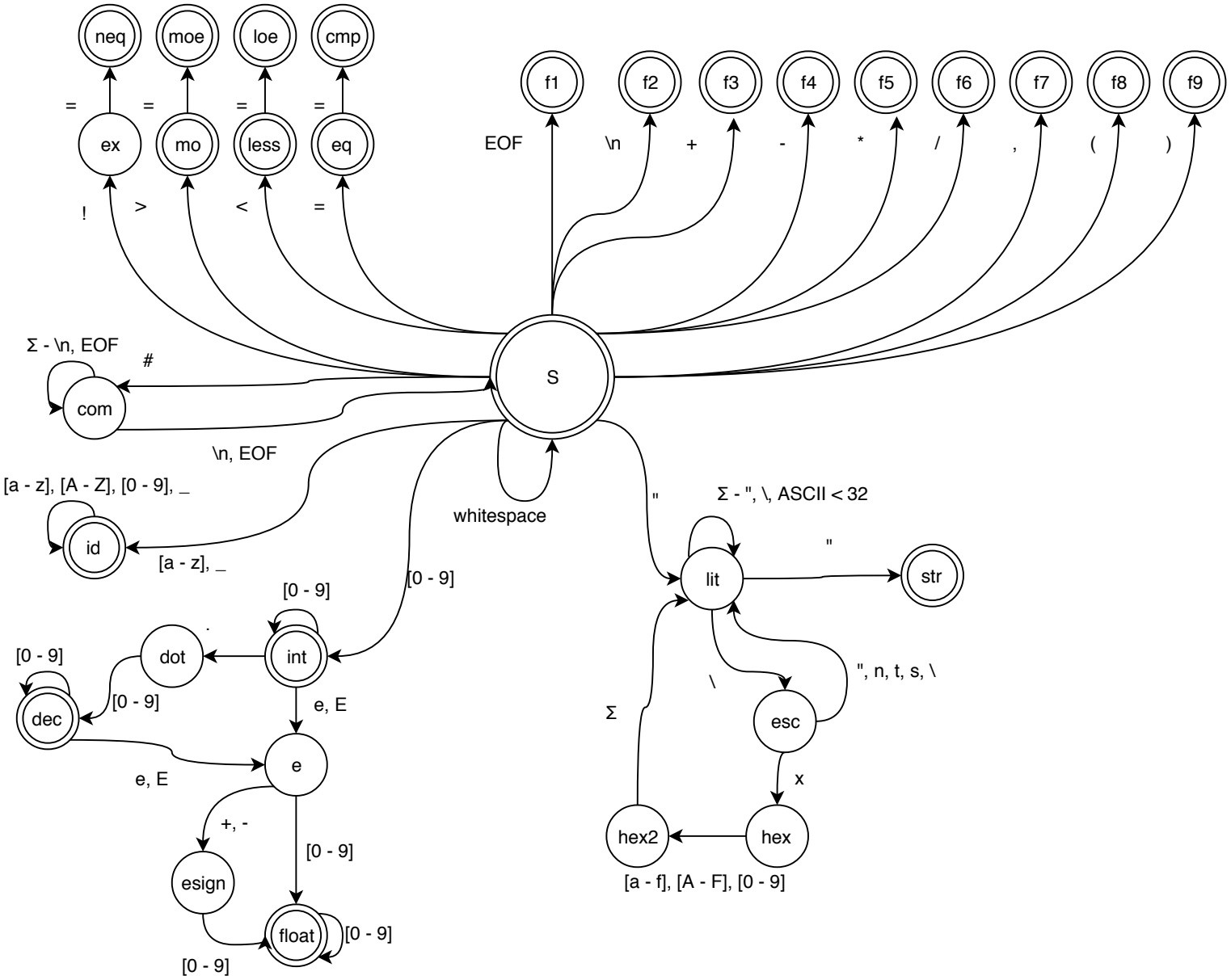
33.<param> -> ID
34.<param> -> EXPRESSION

	EOL	EOF	def	FUNCTION_ID	ε	if	EXPRESSION	while	ID	print	chr	ord	length	substr	inputs	inputf	inputi
<program>			1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
<statement_list>			2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
<end>	3	4															
<statement>				5	6	5	5	5	5	5	5	5	5	5	5	5	5
<body_list>				7	7	8,9	7	8	7	7	7	7	7	7	7	7	7
<body_cycle>						11		10									
<body>				16	15		13		12,14	17	18	19	20	21	22	23	24
<func_params>					25		25		25								
<func_args>					26,27		26,27		26,27								
<params>					28		29		29								
<params_list>					31		30		30								
<next params>							32		32								
<param>							34		34								

Precedenční tabulka

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

Diagram konečného automatu



S = STATE_START
f1 = EOF
f2 = EON
f3 = PLUS
f4 = MINUS
f5 = MUL
f6 = DIV
f7 = COMMA
f8 = LEFT_BRACKET
f9 = RIGHT_BRACKET
eq = STATE_EQUALS
cmp = COMPARE
less = STATE_LESSTHAN
loe = LOE
mo = STATE_MORETHAN
moe = MOE

ex = STATE_EXCLAMATION
neq = NOTEQUAL
com = STATE_COMMENT
id = STATE_ID_KW
int = STATE_NUMBER
dot = STATE_DECIMAL_DOT
dec = STATE_DECIMAL
e = STATE_DECIMAL_E
esign = STATE_DECIMAL_E_SIGN
float = STATE_DECIMAL_END
lit = STATE_STRING_LITERAL
str = STRING
esc = STATE_ESCAPE_SEQUENCE
hex = STATE_HEXADECIMAL
hex2 = STATE_HEXADECIMAL_SECOND