Dokumentace k projektu z předmětů IFJ a IAL

Implementace překladače imperativního jazyka IFJ18

Tým 013, varianta I

Formánková Klára	`xforma14`	20 %
Láncoš Jan	`xlanco00`	30 %
Šebela Vít	`xsebel04`	15 %
Chaloupka Jan	`xchalo16`	35 %

Implementovaná rozšíření: **BOOLOP**, **BASE**

2018/2019

Obsah

Práce v týmu

1. Rozdělení práce mezi členy týmu

1.1. Klára Formánková (xforma14)

Implementace tabulky symbolů
Generace výsledného kódu (vyjma výrazů)

1.2. Jan Láncoš (xlanco00)

Vypracování tabulky LL(1) gramatiky
Návrh a implementace syntaktické analýzy kódu (vyjma výrazů)
Implementace sémantické analýzy kódu (vyjma výrazů)

1.3. Vít Šebela (xsebel04)

Testování kódu

1.4. Jan Chaloupka (xchalo16)

Tvorba a návrh lexikálního analyzátoru
Syntaktická analýza výrazů (precedenční analýzou)
Sémantická analýza výrazů
Generování kódu z výrazů

1.5. Společná práce

Tvorba gramatických pravidel
Dokumentace

2. Zdůvodnění odchylek od rovnoměrnosti bodů

Nesnadná komunikace mezi členy týmů způsobila nepoměr v odpracovaných částech na osobu.

Způsob řešení jednotlivých částí projektu

3. Lexikální analýza

Lexikální analýza je implementovaná, včetně dalších podpůrných funkcí a struktur, ve zdrojovém souboru `scanner.c`. Analyzátor definuje datovou strukturu `Token`, což je token vrácený konečným automatem. Tato datová struktura si uchovává

- *Typ tokenu* jako jednu položku z výčtu `tType`
- *Hodnotu tokenu* u některých typů (např. string, id, int…) – tato hodnota není nijak formátována, tedy i číslo je uloženo jako řetězec přesně tak, jak je reprezentovaný ve zdrojovém kódu.
- *Přesnou pozici*, na které se token v souboru nachází (číslo řádku a počet znaků od začátku řádku), pozice je využita při výpisu chyb – programátor tak ví na kterém místě chyba nastala
- Ukazatele na předchozí a další token, pokud má být výsledkem analýzy seznam tokenů

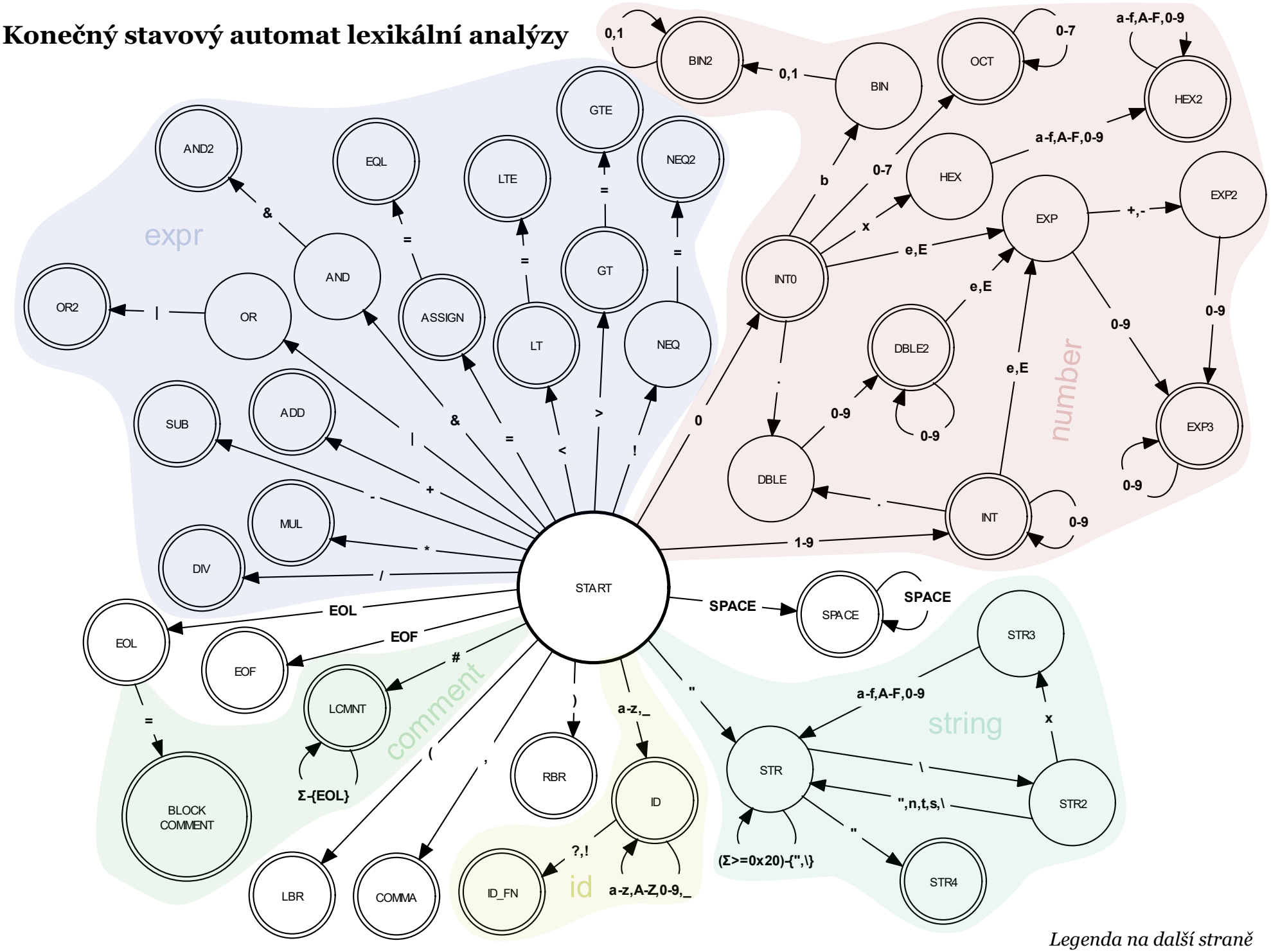
Jsou dva způsoby získání tokenů – voláním funkce `scannerGetToken` se ze souboru načítá token po tokenu. Po dosažení konce souboru vrací funkce pokaždé token typu `T_EOF`. Druhý způsob je zavolání funkce `scannerGetTokenList`, která vrátí seznam všech tokenů načtených ze souboru (jako obousměrný seznam).

Konečny automat je samostatná funkce `scannerFSM` a je navrhnutý podle diagramu – názvy stavů jsou uložené ve výčtu `sState` a jsou shodné s názvy stavů v diagramu. Pokud automat skončí, poslední načtený znak je zachován a při dalším spuštění automatu se začne od zachovaného znaku. V případě nalezení chybného zápisu vrátí analyzátor token `T_EOF`, vypíše chybu na standartní chybový výstup a funkce vrátí číslo 1.

př.: máme vstupní řetězec „3+2“ – automat se ukončí při načtení znaku „+“ a vyhodnotí, že se jedná o číslo. Tento znak si zachová a při dalším spuštění začne znovu od znaku „+“, dále načte znak „2“ a vyhodnotí, že se jedná o symbol plus. Znak „2“ si zachová a při dalším spuštění začne od znaku „2“…

Pokud automat načte token typu `T_ID`, proběhne kontrola, zda se nejedná o klíčové slovo (již mimo automat). V případě, že jde o klíčové slovo, token změní typ a odstraní se dodatečná data.

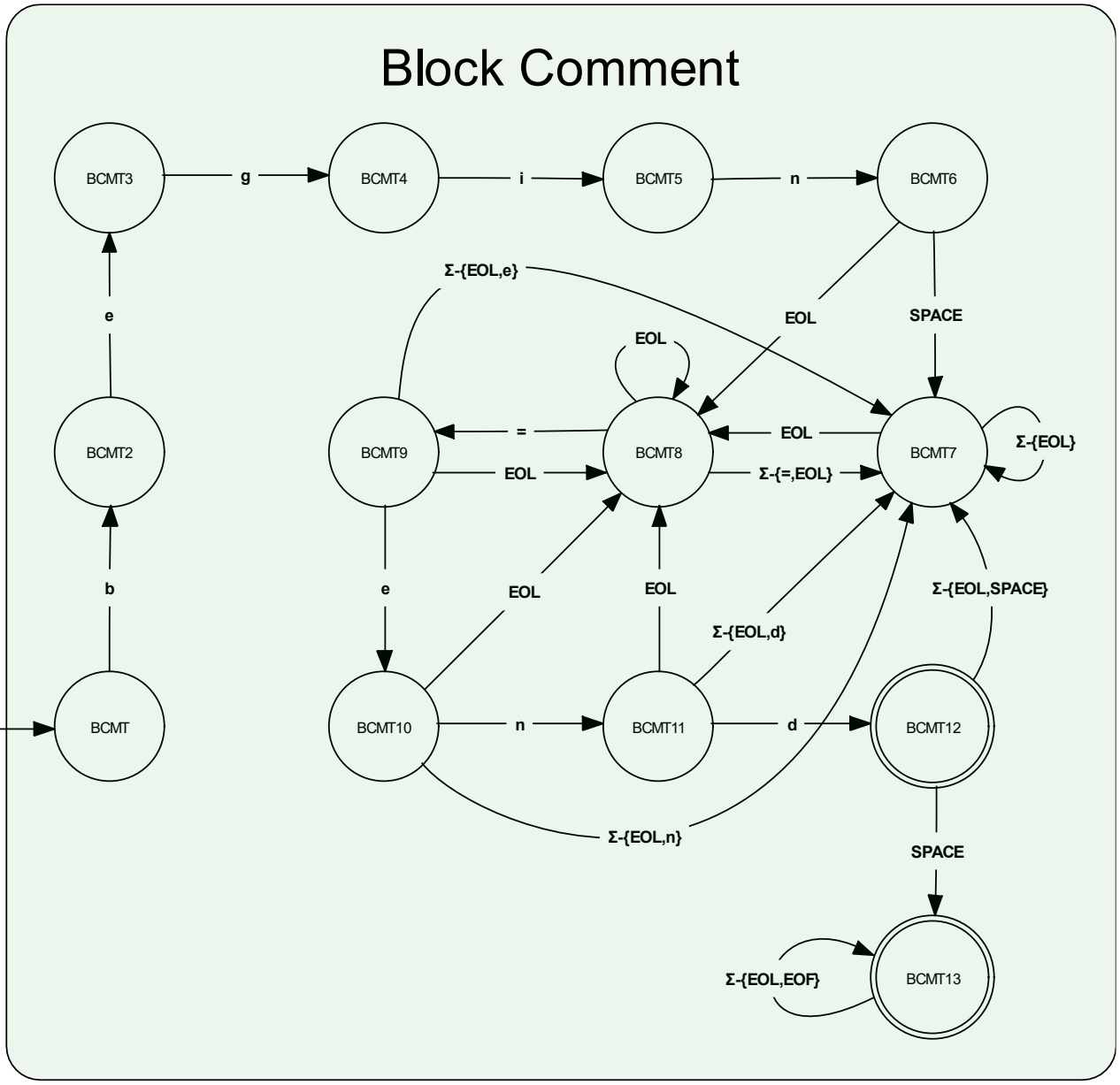
3.1. Konečný stavový automat lexikální analýzy



Legenda na další straně

3.2. Konečný stavový automat (pokračování)

Diagram konečného automatu blokového komentáře je složitější než v případě ostatních větví, proto je vypsán zvlášť (tento blok navazuje na přechod mezi EOL → BLOCK COMMENT v hlavním diagramu).



Legenda

SPACE – *whitespace znak (mimo EOL)*
EOL – *konec řádku*
EOF – *konec souboru*
a-b – *libovolný znak v rozmezí a, b*
a,b,... – *libovolný znak z výčtu*
Σ – *použitá abeceda (mimo EOF)*
Σ >= 0x20 – *jen tisknutelné znaky*
Σ-{...} – *mimo znaky ve složené závorce*

4. Syntaktická a sémantická analýza

4.1. Implementační detaily

Hlavní tělo syntaktické a sémantické analýzy tvoří funkce souboru `parser.c`, konkrétně samotná funkce `parser`.

Po inicializaci **zásobníku** (popsán níže), dvou tabulek symbolů, a implicitním naplnění tabulky určené k zápisu funkcí vestavěnými funkcemi probíhají dva samostatné běhy – **pre-run** a **hlavní běh**.

Při běhové fázi **pre-run** `parser` prochází kompletní seznam tokenů získaný od předešle spuštěné funkce `scannerGetTokenList`, a s pomocí funkce `parserSemanticPreRun` sémanticky kontroluje každý token typu `T_ID`, jehož předchůdcem je token typu `T_DEF`. Jedná-li se o redefinici funkce (`ID` bylo nalezeno v tabulce funkcí), je stav vyhodnocen jako chybový. Pokud tomu tak není, identifikátor funkce je vložen do tabulky funkcí, a s ním i v budoucnu kontrolovaný počet parametrů funkce při jejím volání. Funkce i ze syntaktického hlediska kontroluje, zda-li se v parametrech definované funkce nachází pouze proměnné.

`parser` dále při **hlavním běhu** využívá k syntaktickým kontrolám vstupního kódu principu rekurzivního sestupu za použití dynamicky rozšiřitelného **zásobníku**, datové struktury složené z pole hodnot `tType` a tří integerových hodnot udávající první volné místo zásobníku, pozici nejvyššího prvku na zásobníku a jeho současnou velikost.

Při **hlavním běhu** se využívá stejný obousměrně provázaný seznam tokenů jako ve fázi **pre-run**. Na zásobníku předpokládaných stavů je na počátku implicitně neterminální stav `n_prog`, jenž lze podle postupné aplikace pravidel LL(1) gramatiky postupně rozložit na řetězec terminálů. **Hlavní běh** sestává ze dvou částí, **expand** a **compare**. Tyto části se vzájemně vylučují, každý cyklus je vykonána pouze jedna z nich.

V případě, že se na vrcholu zásobníku nachází neterminál, uskuteční se funkce `parserSyntaxExpand` (část **expand**), a neterminál na vrcholu zásobníku se rozloží na základě terminálu ze vstupního kódu podle pravidel předem určené LL(1) gramatiky. V případě, že se jedná o výraz, volá se externí funkce `exprParse` ze souboru `expression.c`.

V případě, že se na vrchou zásobníku nachází terminál, uskuteční se funkce `parserSyntaxCompare` (část **compare**), a porovná se vstupní terminál s terminálem na vrcholu zásobníku. Pokud se liší, došlo k syntaktické chybě.

Pokud jsou terminály totožné, terminál na vrcholu zásobníku se eliminuje, a následně se provádějí další kontroly.

V případě, že vstupním tokenem je token typu `T_ID`, provádí se syntaktická kontrola znaků "?" a "!" na konci identifikátorů proměnných.

Dále se pomocí funkce `parserSemanticInFunc` vyhodnotí, nacházíme-li se momentálně při průchodu řetězcem tokenů ve funkci, konkrétně mezi tokeny `T_DEF` a příslušným `T_END` (k vyloučení tokenů `T_END`, které k `T_DEF` nepatří, byl implementován jednoduchý **semafor** formou počítadla), čehož poté využívá funkce `parserSemanticsCheck`, jenž provádí za předpokladu tokenu typu `T_ID` zbylé sémantické kontroly (nahlížením do předchozích a následujících tokenů), a s pomocí booleanovské proměnné `inFunc` vyhodnotí, má-li definované proměnné zapisovat do lokálního rámce funkce, nebo rámce hlavního programu.

`parser` poté v **hlavním běhu** volá funkci `codeFromToken` souboru `codegen.c`, generující na základě současného stavu zásobníku, vstupního tokenu a lokální tabulky symbolů výsledný strojový kód.

V případě jak **pre-runu** tak i **hlavního běhu** probíhá na konci každého cyklu kontrola případné interní či překladové chyby za použití funkce `parserError`. Každé zapisování do proměnných `error` a `internalError` je ošetřeno podmínkou zajišťující, že v nich zatím žádná chyba zapsaná není; případné další zachycené chyby tedy nepřepíší chybu původní.

Na konci cyklu se v případě chyby zastaví překlad, vypíše se příslušné chybové hlášení na `stderr` a `parser` vrátí odpovídající návratový kód.

4.2. Gramatická pravidla

1.	<prog>	→	<def_func> EOL <prog>
2.	<prog>	→	<body> <prog>
3.	<prog>	→	EOF
4.	<body>	→	**id** <body_id> EOL <body>
5.	<body>	→	<expr> EOL <body>
6.	<body>	→	ε
7.	<body>	→	<if> EOL <body>
8.	<body>	→	<while> EOL <body>
9.	<body>	→	EOL <body>
10.	<body_id>	→	<expr_o>
11.	<body_id>	→	= <defvar>
12.	<body_id>	→	<func>
13.	<body_id>	→	ε
14.	<type>	→	**nil**
15.	<type>	→	**integer**
16.	<type>	→	**float**
17.	<type>	→	**string**
18.	<type>	→	true
19.	<type>	→	false
20.	<type_id>	→	<type>
21.	<type_id>	→	**id**
22.	<def_func>	→	def **id** (<params>) EOL <body> end
23.	<func>	→	(<params>)
24.	<func>	→	<params>
25.	<params>	→	ε
26.	<params>	→	<type> <params_n>
27.	<params>	→	**id** <params_n>
28.	<params_n>	→	ε
29.	<params_n>	→	, <type_id> <params_n>
30.	<if>	→	if <expr> then EOL <body> else EOL <body> end
31.	<while>	→	while <expr> do EOL <body> end
32.	<def_var>	→	<expr>
33.	<def_var>	→	**id** <def_var_id>
34.	<def_var_id>	→	<expr_o>
35.	<def_var_id>	→	<func>

Pozn.:
<expr> značí začátek výrazu
<expr_o> značí první operátor ve výrazu

4.3. LL(1) tabulka

	<prog>	<body>	<body_id>	<type>	<type_id>	<def_func>	<func>	<params>	<params_n>	<if>	<while>	<def_var>	<def_var_id>
id	2	4	12		21		24	27				33	35
nil	2	5	12	14	20		24	26				32	35
integer	2	5	12	15	20		24	26				32	35
float	2	5	12	16	20		24	26				32	35
string	2	5	12	17	20		24	26				32	35
true	2	5	12	18	20		24	26				32	35
false	2	5		19	20		24	26				32	35
not	2	5										32	
EOF	3												
EOL	2	9	13					25	28				
+	2	5	10									32	34
-	2	5	10									32	34
*****			10										34
/			10										34
==			10										34
!=			10										34
<			10										34
>			10										34
<=			10										34
>=			10										34
(	2	5	12				23					32	35
)								25	28				
=			11										
,									29				
def	1					22							
end		6											
if	2	7								30			
then													
else		6											
while	2	8									31		
do													
$		6						25	28				

4.4. Zpracování výrazů

Výrazy se zpracovávají odděleně od zpracování syntaxe, a to precedenční analýzou. Veškerá práce s výrazy, od syntaktické kontroly až po generaci kódu, probíhá v souboru `expressions.c`.

Hlavní funkce `exprParse` je zavolána hlavním parserem v případě, že parser narazí na výraz, a předá funkci `exprParse` ukazatel na první token z výrazu. Po skončení funkce vrátí stavový kód a ukazatel bude směrovat na token **za výrazem**.

Nezpracované Termy, Neterminály a otevírající závorka (<) se ukládají do zásobníku `eStack`, který je dynamicky se zvětšující – dokud je volná paměť, nehrozí přetečení zásobníku. Protože na zásobníku je najednou více typů hodnot, pole je tvořeno strukturou `eItem`, která obsahuje informace o typu a hodnotu.

Hlavní funkce výrazů inicializuje zásobník a v cyklu načítá tokeny, které ukládá podle **precedenční tabulky** na zásobník. Pokud precedenční tabulka vrátí zavírací znak „>“, spustí se funkce `exprParseStack`, která ověří, jestli mezi „<“, „>“ je gramaticky validní zápis výrazu. Po syntaktickém ověření se provede sémantická kontrola, zda odpovídají typy podle dané operace a případná konverze čísel.

Pokud je ve výrazu **proměnná**, není možné určit její typ, proto v tomto případě proběhne typová kontrola až za běhu programu. Pro kontrolu za běhu je v souboru `common.c` vypsaná základní kostra vygenerovaného programu, která obsahuje všechny vestavěné funkce a mimo to také funkce pro typovou kontrolu za běhu. Nakonec se z výrazu vygeneruje kód a nahradí se na zásobníku za **jeden neterminál**.

Protože náš překladač podporuje rozšíření BOOLOP – v případě, že je výraz součástí podmínky, proběhne typová kontrola, zda je výsledek výrazu typu **bool**, jinak se jedná o běhovou chybu 4. Zároveň také gramatika podporuje zápis záporných čísel **přímo ve výrazu** (*př. 2 + (-3)*) – dovoluje tak vypustit levý operand u operátorů + a -.

Analýza výrazů probíhá, dokud na zásobníku nezůstane samotný terminál $ (mimo neterminály) a na vstupu je $. Poté se funkce ukončí a pokračuje v analýze znovu hlavní parser.

4.5. Precedenční tabulka výrazů

	+	-	*	/	<	>	<=	>=	==	!=	not*	and*	or*	(	)	val	$
+	>	>	<	<	>	>	>	>	>	>		>	>	<	>	<	>
-	>	>	<	<	>	>	>	>	>	>		>	>	<	>	<	>
*****	>	>	>	>	>	>	>	>	>	>		>	>	<	>	<	>
/	>	>	>	>	>	>	>	>	>	>		>	>	<	>	<	>
<	<	<	<	<					>	>	<	>	>	<	>	<	>
>	<	<	<	<					>	>	<	>	>	<	>	<	>
<=	<	<	<	<					>	>	<	>	>	<	>	<	>
>=	<	<	<	<					>	>	<	>	>	<	>	<	>
==	<	<	<	<	<	<	<	<			<	>	>	<	>	<	>
!=	<	<	<	<	<	<	<	<			<	>	>	<	>	<	>
not*	<	<	<	<	<	<	<	<	<	<	<	>	>	<	>	<	>
and*	<	<	<	<	<	<	<	<	<	<	<	<	<	<	>	<	>
or*	<	<	<	<	<	<	<	<	<	<	<	<	<	<	>	<	>
(	<	<	<	<	<	<	<	<	<	<	<	<	<	<	=	<	
)	>	>	>	>	>	>	>	>	>	>		>	>		>		>
val	>	>	>	>	>	>	>	>	>	>		>	>		>		>
$	<	<	<	<	<	<	<	<	<	<	<	<	<	<		<	

** Součást rozšíření BOOLOP*

4.6. Zjednodušená precedenční tabulka výrazů

	+ -	* /	= < > <	!= ==	not*	and or*	(	)	val	$
+ -	>	<	>	>		>	<	>	<	>
*** /**	>	>	>	>		>	<	>	<	>
< > <= >=	<	<		>	<	>	<	>	<	>
!= ==	<	<	<		<	>	<	>	<	>
not*	<	<	<	<	<	>	<	>	<	>
and or*	<	<	<	<	<	<	<	>	<	>
(	<	<	<	<	<	<	<	=	<	
)	>	>	>	>		>		>		>
val	>	>	>	>		>		>		>
$	<	<	<	<	<	<	<		<	

** Součást rozšíření BOOLOP*

Při zpracování výrazů se používá tato zjednodušená precedenční tabulka, kde jsou sjednoceny terminály, které mají společná pravidla. Tímto zjednodušením se zachová funkčnost, ale výrazně se zkrátí zdrojový kód parseru výrazů (V hlavičkovém souboru `expressions.h` jsou tyto termy uložené v `enum eRelTerm` a o vyhledávání v tabulce se stará funkce `exprGetRelation`).

5. Tabulka symbolů

Tabulky symbolů jsou implementovány jako binární vyhledávací stromy. Každý uzel stromu obsahuje kromě identifikátoru a ukazatelů na jeho dva podstromy také data. V nich je uložen typ identifikátoru, informace, jestli byl identifikátor již definován, ukazatel na lokální tabulku symbolů funkce a počet parametrů funkce.

V jednotlivých binárních stromech pak vyhledáváme za pomocí klíče, kterým je pro nás identifikátor.

Funkce pro práci s tabulkou symbolů jsou implementovány v souboru `symtable.c`.

V tomto souboru najdeme mimo jiné funkci `symTabDefvarPre`, která vypíše všechny proměnné v binárním stromě a definuje je.

6. Generace kódu

Pro generování výsledného kódu je implementována funkce `codeFromToken` souboru `codegen.c`, která je volána v hlavním běhu `parseru`. Jednotlivé instrukce jsou generovány podle současného stavu zásobníku, vstupního tokenu a lokální tabulky symbolů.

Funkce po zavolání přechází do jednoho ze stavů podle aktuálního typu tokenu na vrcholu zásobníku. V každém stavu je buď přímo generována instrukce nebo pouze ukládána data pro další generování kódu.

Data vstupního tokenu jsou použita při generování kódu pro definice proměnných a funkcí a také volání funkcí. Konkrétně se využívá ukazatel na předchozí token a proměnná `data`, kde je uložen identifikátor funkce nebo proměnné.

Instrukce pro definici proměnné je generována na začátku těla programu/funkce za použití funkce `symTabDefvarPre`, která vypíše definuci a inicializaci všech proměnný uložených v lokální tabulce symbolů.

Všechny instrukce jsou postupně vypisovány na standardní výstup za běhu `parseru`.

7. Vlastní tělo překladače

Samotný překladač je ve výsledku relativně krátký kód, v němž proběhne postupné volání funkcí `scannerGetTokenList`, `generateBaseCode` a `parser`.

Do proměnné `retval` se ukládá výstupní hodnota funkcí `scannerGetTokenList` a `parser`.

Dojde-li k chybě v rámci lexikální analýzy, volání funkcí `generateBaseCode` a `parser` se již nevykoná.

V obou případech však před ukončením programu proběhne ještě volání funkce `scannerFreeTokenList` a korektně se uvolní alokovaná paměť.

Překladač poté vrací získanou návratovou hodnotu v proměnné `retval`.