Vysoké učení technické v Brně
Fakulta Informačních technologií

Dokumentace překladače

Projekt do předmětů: IFJ, IAL

Tým **xkotou08, Varianta – vv-BVS**

Řešitelé

(25%) Lukáš Kotoun (xkotou08) – **vedoucí týmu**
(25%) Petr Novák (xnovak3l)
(25%) Jan Pánek (xpanek11)
(25%) Tibor Šimlaštík (xsimla00)

Rozšíření

2. prosince 2023

Obsah

1 Úvod

Dokumentace obsahuje popis jednotlivých částí překladače imperativního jazyka IFJ2023, který tvoří podmnožinu jazyka Swift. V jednotlivých kapitolách jsou podrobněji popsány jednotlivé podčásti překladače a formální modely, na kterých jsou založeny.

2 Lexikální analýza

Logika lexikálního analyzátoru je implementována v souboru `lexilcal_analyzer.c`. Strukturu pro reprezentaci lexikálního analyzátoru `Scanner` obsahuje soubor `lexilcal_analyzer.h`. Strukturu pro reprzetanci tokenů společně s definovanými typy tokenů a operace nad tokeny obsahují soubory `tokens.h`, `tokens.c`.

Lexikální analyzátor je implementován na základě navrženého deterministického konečného automatu 1, kdy jednotlivé podčásti automatu jsou implementovány pomocí samostatných funkcí. Funkce realizující části automatu jsou volány z hlavní funkce `scan_token`, která sekvenci znaků načtenou ze standardního vstupu reprezentuje pomoci sturktury `Token`, kterou vrací. Pro propagaci chyby byly vyhrazeny typy tokenů `TOKEN_LA_ERROR` a `TOKEN_MEMMORY_ERROR`.

Pro umoznění nahlednutí vpřed `Scanner` obsahuje buffer tří následujících nezpracovaných znaků. Náhlédnutí vpřed realizují funkce `peek`, `peek_next` a `forward_lookup`, která navíc umožňuje porovnat obsah bufferu se specifikovaným formátem. Velikost bufferu je dostatečná pro určení části automatu, do které má být přechod uskutečněn bez nutnosti vracení znaků. Na základě obsahu bufferu je z funkce `scan_token` volána funkce implementující patřičnou část automatu.

Bíle znaky jsou zpracovány přimo ve funkci `scan_token`. Zpracování jednořádkových komentářů implementuje funkce `consume_single_line_comment` a víceřádkových komentářů funkce `consume_multi_line_comment`. Jelikož vnořené víceřádkové komentáře nelze modelovat pomocí samotného konečného automatu, bylo použito počítadlo vnoření.

Operátory, závorky a `_`, `:`, `,`, `->` jsou zpracovány funkcí `scan_operator`. Číslicové literály jsou zpracovány ve funkci `scan_number_literal`.

Čás automatu pro identifikátory a klíčová slova je implementována funkcí `scan_identifier`. Funkce `is_kw` následně určí, zda se jméno načteného identifikátoru shoduje se jménem klíčového slova a případně vrací token pro dané klíčové slovo. Jinak vrací token pro daný identifkátor. U klíčových slov `Double`, `String`, `Int` je ještě provedena kontrola, zda není následováno znakem `?`.

Částem automatu pro jednořádkové a víceřádkové řetězce korespondují funkce `scan_single_line_string`, `scan_multi_line_string`. Řetězce jsou zpracovány dvěma průchody. Při prvním průchodu jsou načteny do bufferu a je určeno, zda jsou korektně zakončeny. V druhém průchodu jsou zpracovány obsažené escape sekvence a u víceřídkových řetězců ještě odsazení.

Pro účely syntaktické analýzy je u každého lexému, který nasleduje `\n` nebo {, nastaven příznak `follows_separator`.

3 Tabulka symbolů

Podle vybrané verze řešení projektu je námi implementována tabulka založena na výškové balancovaném stromu, konkrétně samo-vyvažovací strom typu AVL. Každý element stromu ukládá informaci o svojí výšce, ukazatel na své dva potomky a informace o symbolu které jsou o něm uložené v typu `symData`. Typ `symData` ukládá jméno symbolu, typ symbolu (proměnná nebo funkce), datový typ proměnný nebo návratový typ funkce, zda je daný symbol konstanta, deklarovaný a jestli byl inicializován. Jestli symbol reprezentuje funkci, jsou v `symData` uložený též informace o jeho parametrech (jejich počet a datové typy).

Ze vnějšku se pracuje s typem symtable a wrapper funkcemi začínající s prefixem `symtable-`. Typ symtable obsahuje kořen stromu typu `symtTreeElementPtr`. S tímhle typem a se stromem samotným pracují rekurzivní funkce s prefixem `symtTree-`. Propagace změny struktury, tj. přidání/odstranění elementu ze stromu se vykonává pomocí návratové hodnoty rekurzivních funkcí, kterou je ukazatel na pozměněný podstrom. Před návratem ukazatele je na něj volaná funkce `symtTreeRebalance`, která aktualizuje výšku a v případě potřeby vykoná rotaci podstromu a vrací finální kořen daného podstromu.

4 Syntaktická a sémantická analýza

Syntaktická analýza řídí celý překlad, proto pracuje skoro ze všemi moduly celého překladače. Využívá lexikální analýzu pro postupné získávání tokenů, volá precedenční analýzu pro zpracování výrazů, používá hierarchickou tabulku symbolů pro ukládání informací o proměnných a funkcích a zajišťuje generování výsledného kódu.

4.1 Zpracování příkazů

Syntaktická analýza je implementována jednoprůchodově pomocí metody rekurzivního sestupu a řídí se pomocí LL gramatiky a LL tabulky. Každé pravidlo z LL gramatiky je implementováno jako samostatná funkce s prefixem `rule_`. Mezi jednotlivými funkcemi nejsou předávány parametry veškerá sdílená data jsou uložena v globálních proměnných.

Sémantická analýza je poté vepsána přímo do jednotlivých syntaktických funkcí a využívá další pomocné funkce pro kontrolu mnoha pravidel jako například: kontrola redefinice u definice proměnné nebo funkce, kontrola kompatibility datových typů, nebo kontrola parametrů u volání funkce.

Pro jednoduší kontrolu parametrů u volání a definice funkce byla využita datová struktura `param_vector` pro ukládání typů parametrů a jejich názvů. Vzhledem k implementaci jednoprůchodové analýzy, byla pro kontrolu volání funkce před její definicí implementována pomocná funkce na odvození datových typů parametrů. Odvození probíhá na základě datových typů použitích u jednotlivých volání. Při definici funkce je poté zkontrolována kompatibilita definovaných datových typů s odvozenými datovými typy použitými při volání.

V případě, kdy je odhalena syntaktická nebo sémantická chyba je volána funkce `errorHandle`, která ukončí provádění programu s chybovým kódem pomocí instrukce exit. Instrukce byla použita z důvodu příliš komplikovaného propagování chybového kódu.

Implementace těchto částí syntaktické analýzy se nachází v souborech `syntactic_analysis.c`, `syntactic_analysis.h`.

4.2 Zpracování výrazů

Precedenční analýza je implementována na základě tabulky 1. Její programová podoba se nachází v souboru `precedence_analysis.h`. Logika pro zpracování výrazů je implementována v souboru `precedence_analysis.c`. Při zpracování výrazů se využívá datové struktury `ExpressionStack`, `ExpressionStackItem` definované v souboru `expression_stack.h`. Zásobníková struktura je implementována jako nafukovací pole. Operace nad `ExpressionStack` a `ExpressionStackItem` jsou implementovány v souboru `expression_stack.c`.

Klíčovou funkcí pro zpracování výrazů je `parse_expression`, která na základě precedenční tabulky a aktuálního tokenu určí, jaká operace má být provedena nad `ExpressionStack` (přidání prvku, započetí podvýrazu s vyšší prioritou, pokus o redukci pravida), nebo zda má být generován error.

V rámci precedenční analýzy rozlišujeme několik typů prvku zásobníku. Typy `LITERAL`, `IDENTIFIER`, `TERMINAL` reprezentují terminály. Prvek s typem `EXPRESSION` potom vzniká při aplikaci redukčních pravidel. Zásobník udržuje přehled o nejvrchnější přidaném terminálu pomocí indexu do pole realizujícího zásobník `top_most_expr_start` a zároveň každý terminál obsahuje příznak `start_of_expr`, který

indukuje, že terminál nad ním je již součástí podvýrazu s vyšší prioritou. Sekvence prvků na indexech vyšších než `top_most_expr_start` potom tvoří vrchol zásobníku.

Při aplikaci redukčního pravidla je na základě vrcholu zásobníku odvozeno číslo pravidla, které by se mělo aplikovat, pomocí funkce `get_rule_number`.

Zpracování výrazů je realizováno pomocí 2 průchodů. Při prvním průchodu jsou tokeny načítání do vektoru a je řešena syntaxe výrazu. Zároveň je při prvním průchodu kontrolováno, zda se ve výrazu nevyskytuje operand s typem `Double`, `Double?`. Pokud ano, je odvozen typ výsledného výrazu `Double`.

V druhém průchodu jsou následně prováděny sémantické akce a generování kódu. Ukazatele na funkce realizující redukční pravidla jsou udržované v poli na indexu korespondujícím číslu pravidla. Jednotlivé funkce implementují sémantické akce a generování kódu.

5 Generování kódu

Generování kódu je řešeno funkcemi v souboru `codegen.c`, `codegen.h`, které jsou volány v syntaktické analýze. Příkazy pro interpret jsou psány rovnou na standartní výstup. Pouze když se generuje kód, který je součástí cyklu while, tak se příkazy ukládají do dynamického stringu, který je vypsaný po ukončení cyklu. Definování proměnných se vypíše před tento string, protože se nemůže definovat proměnná se stejným názvem vícekrát.

Na začátku vykonávání SA se vypíše na výstup string, obsahující vestavěné funkce a globální proměnné potřebné pro správný chod programu.

Na začátku volání funkce se nejdříve vytvoří nový dočasný rámec, do kterého se uloží hodnoty parametrů pojmenovaných číslem, o kolikátý parametr se jedná, ve formátu „_číslo“. Na začátku definování funkce se dočasný rámec uloží do zásobníku lokálních rámců, funkce potom pracuje s lokálními či globálními proměnnými. Poté se přeuloží hodnoty parametrů do proměnných se správným názvem parametru, aby se s nimi lépe pracovalo. Na konci funkce se návratová hodnota uloží na datový rámec a aktuální lokální rámec se vyjme ze zásobníků rámců.

Každá proměnná vytvořená uživatelem má k sobě připojené číslo scopu, aby se docílilo jedinečnosti názvu proměnných. Swift umožňuje překrývání proměnných v různých scopech.

Podobně se přidává číslo při vytváření labelů pro if a while, kde while i if mají své unikátní počítadlo. Při jejich ukončování se využívá zásobníku, který umožňuje jejich nekonečné zanořování.

Dále se generování kódu využívá v precedenční analýze, kde se veškeré výrazy počítají na datovém zásobníku pomocí zásobníkových operací. V některých případech, kdy neexistují zásobníkové operace, se využívá globálních proměnných, které jsou definovány na začátku programu.

6 Pomocné datové struktury

Kromě již zmíněných datový struktur, jejichž popis byl pro větší přehlednost zařazen do konkrétních kapitol, byly dále použity následující pomocné datové struktury.

Pro účely ukládání řetězců literálů a identifikátorů slouží pomocná datová struktura `LiteralVector`. Tato struktura umožňuje bufferování znaků tvořícího se literálu pomocí `LV_add` a jeho následné uložení pomocí `LV_submit`. Implementaci obsahují soubory `literal_vector.c`, `literal_vector.h`.

Pro ukládaní tabulek symbolu pro jednotlivé scopy byl vytvořen zásobníkový typ `symStack`, který je naimplementovány na základě jednosměrného lineárního seznamu. Kromě základních funkcí zásobníku umožňuje symstack i přechod pomocí ukazatele activeTable a funkcí `ActiveToTop` a `Next`. Element symstacku ukládá ukazatel na dynamicky alokovaný symtable a jedinečný identifikátor daného scopu. Tabulka se Scope ID 0 je tabulka Globálního scopu. Implementaci obsahují soubory `symstack.c`, `symstack.h`.

Pro kontrolu parametrů funkcí byla v sémantické analýze použita datová struktura `ParamVector`. Pro bufferování tokenů při prvním průchodu v precedenční analýze byla použita struktura `TokenVector`.

Implementace jsou popořadě obsaženy v souborech `token_vector.c`, `token_vector.h`, `param_vector.c`, `param_vector.h`.

7 Rozdělení práce a plnění projektu

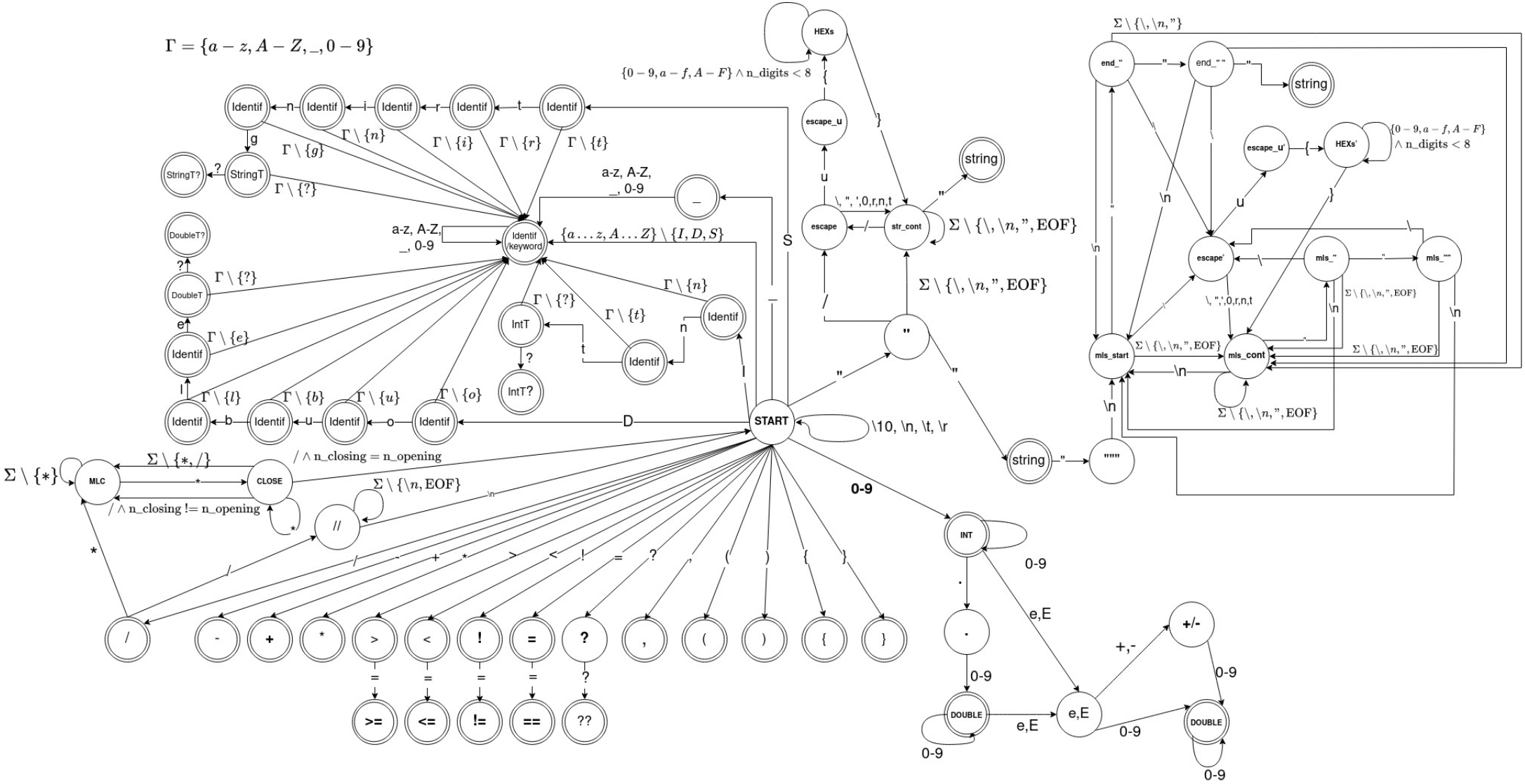
Po rozdělení práce si všichni nastudovali potřebnou teorii a ve spolupráci s ostatními včas dokončili požadovanou část. Komunikace probíhala prostřednictvím discordu a osobních setkání, která se konala jednou týdně. Pro verzování jsme použili git hostovaný na GitHubu.

Rozdělení práce:

Lukáš Kotoun (xkotou08): Syntaktická a sémantická analýza, gramatika, testování, dokumentace, LL-gramatika
Petr Novák (xnovak3l): Generování kódu, testování, dokumentace, LL-gramatika
Jan Pánek (xpanek11): Lexikální analýza, precedenční a sémantická analýza, dokumentace
Tibor Šimlaštík (xsimla00): Tabulka symbolů, generování kódu, dokumentace

8 Přílohy

8.1 Diagram konečného automatu



Obrázek 1: Diagram deterministického konečného automatu

8.2 Gramatika

1. ⟨prog⟩ → ⟨prog-body⟩ EOF
2. ⟨prog-body⟩ → ⟨func-list⟩ ⟨statement-list⟩ ⟨prog-body⟩
3. ⟨prog-body⟩ → ε
4. ⟨statement-list⟩ → ⟨statement⟩ ⟨statment-list⟩
5. ⟨statement-list⟩ → ε
6. ⟨statement-func-list⟩ → ⟨statement-func⟩ ⟨statment-func-list⟩
7. ⟨statement-func-list⟩ → ε
8. ⟨func-list⟩ → ⟨func-decl⟩ ⟨func-list⟩
9. ⟨func-list⟩ → ε
10. ⟨func-decl⟩ → FUNC ID (⟨param-first⟩) ⟨return-type⟩ ⟨func-body⟩
    - 10.1. ⟨param-first⟩ → ⟨param⟩ ⟨param-n⟩
    - 10.2. ⟨param-first⟩ → ε
    - 10.3. ⟨param-n⟩ → , ⟨param⟩ ⟨param-n⟩
    - 10.4. ⟨param-n⟩ → ε
    - 10.5. ⟨param⟩ → ⟨param-name⟩ ⟨param-res⟩
      - 10.10.1. ⟨param-name⟩ → _
      - 10.10.2. ⟨param-name⟩ → ID
      - 10.10.3. ⟨param-res⟩ → _ : ⟨type⟩
      - 10.10.4. ⟨param-res⟩ → ID : ⟨type⟩
    - 10.6. ⟨return-type⟩ → -> ⟨type⟩
    - 10.7. ⟨return-type⟩ → ε
11. ⟨type⟩ → INT
12. ⟨type⟩ → INT-NIL
13. ⟨type⟩ → DOUBLE
14. ⟨type⟩ → DOUBLE-NIL
15. ⟨type⟩ → STRING
16. ⟨type⟩ → STRING-NIL
17. ⟨func-body⟩ → { ⟨statement-func-list⟩ }
18. ⟨body⟩ → { ⟨statement-list⟩ }
19. ⟨statement-func⟩ → RETURN ⟨return-value⟩
    - 19.1. ⟨return-value⟩ → ⟨expr⟩
    - 19.2. ⟨return-value⟩ → ε

20. ⟨statement-func⟩ → IF ⟨if-cond⟩ ⟨func-body⟩ ELSE ⟨func-body⟩
    20.1. ⟨if-cond⟩ → LET ID
    20.2. ⟨if-cond⟩ → ⟨expr⟩
21. ⟨statement-func⟩ → WHILE ⟨expr⟩ ⟨func-body⟩
22. ⟨statement-func⟩ → ⟨id-decl⟩ ⟨decl-opt⟩
    22.1. ⟨id-decl⟩ → VAR ID
    22.2. ⟨id-decl⟩ → LET ID
    22.3. ⟨decl-opt⟩ → : ⟨type⟩ ⟨assign⟩
      22.22.1. ⟨assign⟩ → ε
      22.22.2. ⟨assign⟩ → = ⟨statement-value⟩
    22.4. ⟨decl-opt⟩ → = ⟨statement-value⟩
23. ⟨statement-func⟩ → ID ⟨statement-action⟩
24. ⟨statement⟩ → IF ⟨if-cond⟩ ⟨body⟩ ELSE ⟨body⟩
25. ⟨statement⟩ → WHILE ⟨expr⟩ ⟨body⟩
26. ⟨statement⟩ → ⟨id-decl⟩ ⟨decl-opt⟩
27. ⟨statement⟩ → ID ⟨statement-action⟩
28. ⟨statement-action⟩ → = ⟨statement-value⟩
29. ⟨statement-action⟩ → (⟨first-arg⟩)
    29.1. ⟨first-arg⟩ → ⟨arg⟩ ⟨arg-n⟩
      29.29.1. ⟨arg⟩ → ⟨literal⟩
      29.29.2. ⟨arg⟩ → ID ⟨arg-opt⟩
        29.29.29.1. ⟨arg-opt⟩ → : ⟨term⟩
        29.29.29.2. ⟨arg-opt⟩ → ε
    29.2. ⟨first-arg⟩ → ε
    29.3. ⟨arg-n⟩ → , ⟨arg⟩ ⟨arg-n⟩
    29.4. ⟨arg-n⟩ → ε
30. ⟨statement-value⟩ → ID ⟨arg-expr⟩
    30.1. ⟨arg-expr⟩ → (⟨first-arg⟩)
    30.2. ⟨arg-expr⟩ → ⟨expr⟩
31. ⟨statement-value⟩ → ⟨expr⟩
32. ⟨term⟩ → ID
33. ⟨term⟩ → ⟨literal⟩
    33.1. ⟨literal⟩ → INT-LITERAL
    33.2. ⟨literal⟩ → DOUBLE-LITERAL
    33.3. ⟨literal⟩ → STRING-LITERAL
    33.4. ⟨literal⟩ → NIL-LITERAL

8.3 LL-tabulka

	EOF	ID	(	)	{	}	LET	VAR	IF	ELSE	WHILE	FUNC	RETURN	:	,	_	->	=	!	-	+	*	/	??
<prog>	1	1					1	1	1		1	1												
<prog-body>	3	2					2	2	2		2	2												
<statement-list>	5	4				5	4	4	4		4	5												
<statement>		27					26	26	24		25													
<statement-func-list>		6				7	6	6	6		6		6											
<statement-func>		23					22	22	20		21		19											
<func-list>	9	9					9	9	9		9	8												
<func-decl>												10												
<param-first>		10.1		10.2												10.1								
<param-n>				10.4											10.3									
<param>		10.5														10.5								
<param-rest>		10.5.4														10.5.3								
<param-name>		10.5.2														10.5.1								
<return-type>					10.7												10.6							
<type>																								
<func-body>					17																			
<body>					18																			
<return-value>		19.1	19.1			19.2	19.2	19.2	19.2		19.2		19.2											
<if-cond>		20.2	20.2				20.1																	
<id-decl>							22.2	22.1																
<decl-opt>														22.3				22.4						
<assign>	22.3.1	22.3.1				22.3.1	22.3.1	22.3.1	22.3.1		22.3.1	22.3.1	22.3.1					22.3.2						
<statement-action>			29															28						
<statement-value>		30	31																					
<first-arg>		29.1		29.2																				
<arg-n>				29.4											29.3									
<arg>		29.1.2																						
<arg-opt>				29.1.2.2										29.1.2.1	29.1.2.2									
<arg-expr>			30.1																30.2	30.2	30.2	30.2	30.2	30.2
<term>		32																						
<literal>																								

Obrázek 2: LL tabulka

	INT	INT-NIL	DOUBLE	DOUBLE-NIL	STRING	STRING-NIL	INT-LITERAL	DOUBLE-LITERAL	STRING-LITERAL	NIL-LITERAL
<prog>										
<prog-body>										
<statement-list>										
<statement>										
<statement-func-list>										
<statement-func>										
<func-list>										
<func-decl>										
<param-first>										
<param-n>										
<param>										
<param-rest>										
<param-name>										
<return-type>										
<type>	11	12	13	14	15	16				
<func-body>										
<body>										
<return-value>							19.1	19.1	19.1	19.1
<if-cond>							20.2	20.2	20.2	20.2
<id-decl>										
<decl-opt>										
<assign>										
<statement-action>										
<statement-value>							31	31	31	31
<first-arg>							29.1	29.1	29.1	29.1
<arg-n>										
<arg>							29.1.1	29.1.1	29.1.1	29.1.1
<arg-opt>										
<arg-expr>										
<term>							33	33	33	33
<literal>							33.1	33.2	33.3	33.4

Obrázek 3: LL tabulka – pokračování

8.4 Precedenční tabulka

	SEP	TERM	(	)	+	-	*	/	==	!=	<=	>=	>	<	!	??
SEP	x	<	<	>	<	<	<	<	<	<	<	<	<	<	<	<
TERM	>	x	x	>	>	>	>	>	>	>	>	>	>	>	>	>
(	>	<	<	=	<	<	<	<	<	<	<	<	<	<	<	<
)	>	x	x	>	>	>	>	>	>	>	>	>	>	>	>	>
+	>	<	<	>	>	>	<	<	>	>	>	>	>	>	<	>
-	>	<	<	>	>	>	<	<	>	>	>	>	>	>	<	>
*	>	<	<	>	>	>	>	>	>	>	>	>	>	>	<	>
/	>	<	<	>	>	>	>	>	>	>	>	>	>	>	<	>
==	>	<	<	>	<	<	<	<	x	x	x	x	x	x	<	<
!=	>	<	<	>	<	<	<	<	x	x	x	x	x	x	<	<
<=	>	<	<	>	<	<	<	<	x	x	x	x	x	x	<	<
>=	>	<	<	>	<	<	<	<	x	x	x	x	x	x	<	<
>	>	<	<	>	<	<	<	<	x	x	x	x	x	x	<	<
<	>	<	<	>	<	<	<	<	x	x	x	x	x	x	<	<
!	>	x	x	>	>	>	>	>	>	>	>	>	>	>	x	>
??	>	<	<	>	<	<	<	<	<	<	<	<	<	<	<	<

Tabulka 1: Precedenční tabulka