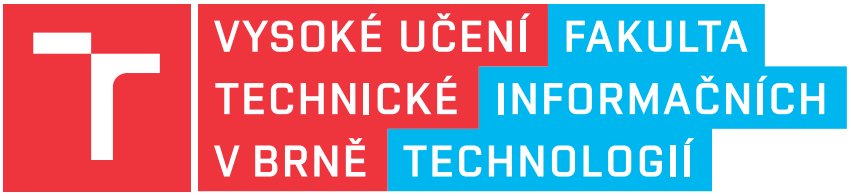


Dokumentace k projektu do IFJ / IAL
Implementace překladače imperativního jazyka IFJ23

Tým xhobza03 varianta TRP-izp

6.12.2023

Autoři:

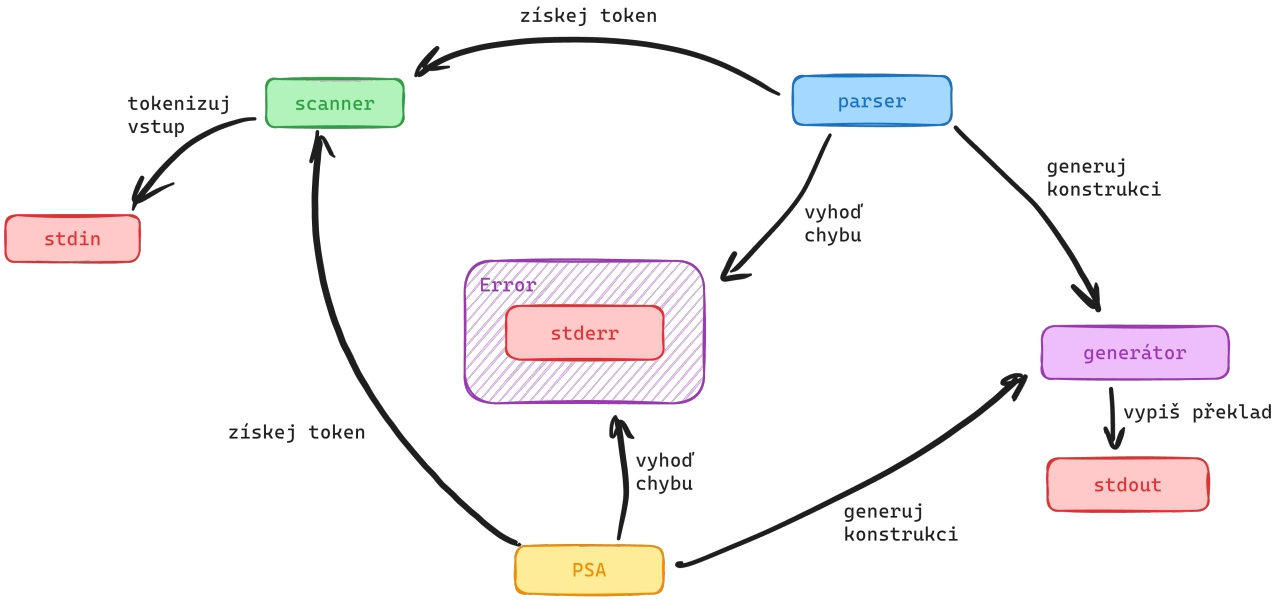
Anastasia Butok - 25%	xbutok00@stud.fit.vutbr.cz
Simona Valkovská - 25%	xvalko12@stud.fit.vutbr.cz
Tomáš Hobza - **vedoucí** - 25%	xhobza03@stud.fit.vutbr.cz
Jakub Všetečka - 25%	xvsete00@stud.fit.vutbr.cz

Obsah

1 Úvod

Tato dokumentace popisuje návrh a implementaci překladače imperativního jazyka IFJ23, který je zjednodušenou podmnožinou jazyka Swift, který byl zamýšlen jako alternativa k Objective-C . Naše varianta TRP, měla za úkol implementovat tabulku symbolů, která se nachází v souboru `symtable.c` a `symtable.h` pomocí tabulky s rozptýlenými položkami (hashovací tabulky).

2 Implementace



Obrázek 1: zjednodušené schéma celého překladače

2.1 Části překladače

- Lexikální analyzátor
- Rekurzivní syntaktický analyzátor (*celková syntaxe a mimo-výrazová sémantika*)
- Precedenční syntaktický analyzátor (*syntaxe a sémantika výrazů*)
- Generátor kódu

2.2 Lexikální analyzátor

Při implementaci Lexikálního analyzátoru využíváme principu deterministického stavového automatu, který je převeden do kódové podoby a je obohacen o speciální operace při práci s řetězci. Analyzátor je implementován v souborech `scanner.c` a `scanner.h`.

Analyzátoru přijde od ‘parseru‘ požadavek na token. Následně se analyzátor rozhodne, zda vybrat token ze zásobníku vracených tokenů, nebo jestli je potřeba načíst nový token.

Při čtení nového tokenu čte analyzátor vstupní soubor znak po znaku a na základě přečtených znaků analyzátor přechází do dalších stavů, do té doby než se dostane ke konci lexikálního celku - tokenu. Následně se do něj nahrajou všechny potřebné informace, konkrétně typ tokenu dle vytvořeného výčtového typu (enumu), hodnota jež analyzátor přečetl, booleovský příznak zda tokenu předcházel

znak nového řádku (příznak je potřeba pro správné vyhodnocení výrazů, které mohou být zapsané na více řádcích). Token je také obohacen o číslo řádku, na němž se nachází (nápomocné např. při debugovaní).

Až analyzátor narazí na konec souboru, do struktury token se nahrajou hodnoty a funkce vrátí číselný kód chyby, která mohla nastat.

2.3 Rekurzivní syntaktický analyzátor (*celková syntaxe a mimo-výrazová sémantika*)

2.3.1 Syntaktická analýza

Syntaktická analýza byla implementována pomocí rekurzivního sestupu s využitím LL-gramatiky(1) a LL-tabulky(5). Implementace se nachází v souborech `parser.c`, `parser.h` a `parser_utils.c`.

Rozhraní mezi syntaktickou a lexikální analýzou představuje funkce `main_scanner`, která při zavolání vrací parametrem nově načtený token.

Komunikace se sémantickou analýzou a generátorem probíhá převážně ve funkci `cmp_type`. Ta kromě kontroly shody mezi tokenem a očekávaným terminálním symbolem také volá funkci `run_control` (viz 2.3.2), kterému parametrem nastaví odpovídající stav typu `Contorl_state`. Stav pro `run_control` určuje syntaktická analýza.

V rámci rekurzivního sestupu se kromě tokenu propaguje i struktura `sym_items` obsahující datové struktury pro uchování potřebných informací o symbolu potřebných pro sémantickou analýzu.

Anomálií od ostatních funkcí zajišťujících rekurzivní sestup je funkce `EXP` reprezentující v jazyce IFJ23 výraz. Standardně u rekurzivního sestupu každá funkce implementuje rozklady pomocí gramatických pravidel jednoho konkrétního neterminálního symbolu abecedy daného jazyka, avšak `exp` je v naší abecedě terminálním symbolem. Funkce `EXP` tedy slouží pro komunikaci s PSA. Aby bylo možné PSA předat očekávaný token, musí navíc funkce nejdřív aktuální token vrátit lexikální analýze.

2.3.2 Sémantická analýza

Sémantická analýza jakožto i samotný překlad je řízen syntaktickou analýzou. Za účelem udržení modularity překladače, jsme se rozhodli implementovat řídící stavy v `parser_control.c`, kde se nachází i samotná funkce `run_control`.

Jak již bylo zmiňováno, funkci volá syntaktická analýza s odpovídajícím stavem, tokenem a propagovanými datovými konstrukcemi. Jednotlivé stavy pak určují, které sémantické akce, implementované v souborech `semantic.c` a `semantic.h`, se mají vykonat. Kromě toho jsou zde volány i funkce zajištující generování kódu.

Sémantický analyzátor má na starost kromě sémantických kontrol i vytváření tabulek symbolů a jejich následné zaplňování symboly proměnných a funkcí.

Problém s voláním doposud nedefinované funkce řeší funkce `get_func_definition` ve `parser_control.c`. Ta při zavolání načítá tokeny dokud nenarazí na definici dané funkce a vytvoří pro ni symbol s potřebnými údaji, který poté vrací v podobě parametru.

Pokud se v těle funkce s návratovou hodnotou nachází `if` anebo `while`, může to značně zkomplikovat ověření toho, zda se vykoná příkaz `return`. Z toho důvodu jsme do struktury `symtable`, kromě samotné tabulky symbolů, přidali i Boolovské atributy: `found_return`, `found_else` a `all_children_return`, které tuhle kontrolu značně ulehčí.

2.4 Precedenční syntaktický analyzátor (*syntaxe a sémantika výrazů*)

Precedenční syntaktický analyzátor (*dále bude zmíněn i jen jako PSA*) jsme vytvořili s rozhraním dvou funkcí pro ostatní moduly.

Tyto funkce jsou:

- `parse_expression()` - všeobecná funkce pro výraz

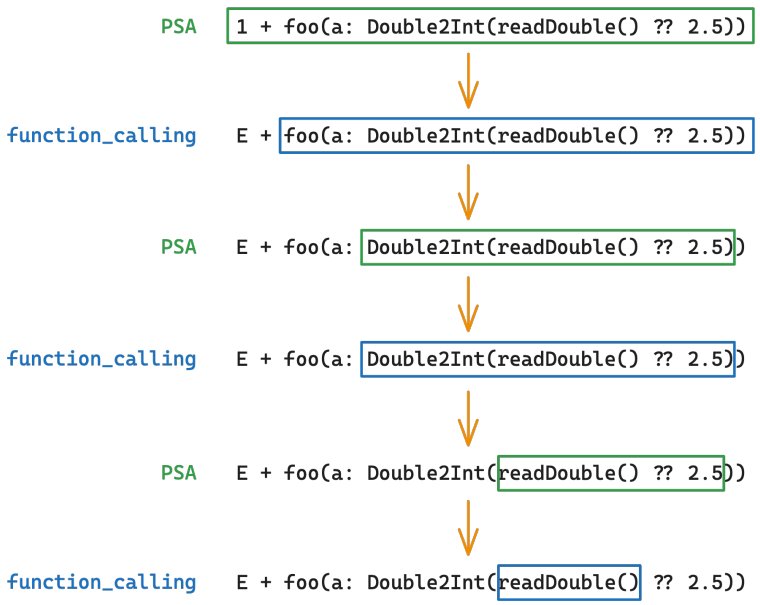
- `parse_param()` - funkce specificky pro výraz v argumentu volání funkce

Tyto dvě funkce se liší pouze tím, co považují za validní ukončení výrazu. Výrazu v argumentu volání funkce jsou totiž ukončený buď znakem čárky, či pravou kulatou závorkou bez páru.

Tokeny načítá PSA přímo od scanneru. V rámci PSA je implementovaná logika, která sama rozhoduje o konci výrazu. Funkce pro čtení dalšího tokenu (*v souboru* `io.c`) sama dle kontextu kontroluje validitu nově načteného tokenu ve výrazu. Nevalidním tokenem může být například operand za operandem, či token, který není obsažen v precedenční tabulce.

Implementace PSA je rozdělena na několik částí:

- `psa.c` - hlavní dvě funkce PSA - rozhraní
- `ptable.c` - precedenční tabulku a Look-Up-Table pro hledání v ní
- `semantic_control.c` - typové kontroly ve výrazech a operacích
- `rule_handle.c` - stará se o derivaci handle na výsledný derivovaný token
- `function_calling.c` - parsing volání funkcí (*a rekurzivní volání PSA*)
- `psa_utils.c` - podpůrné funkce pro PSA



Obrázek 2: diagram volání PSA a parsingu volání funkcí

2.5 Generátor kódu

Generátor je postaven na funkcích tvořící postupné abstrakce pro tvorbu výsledného kódu. Popsat tedy půjde nejlépe dle úrovní těchto abstrakcí jdoucí od nejnižší úrovně po tu nejvyšší.

1. Funkce pracující přímo s dočasným souborem pro výpis výsledného kódu.
   - `handle_0_operand_instructions()`
   - `handle_1_operand_instructions()`
   - `handle_2_operand_instructions()`

- `handle_3_operand_instructions()`
- `processInstruction()`

Tyto funkce slouží pro formalizaci výpisu a unifikaci formátu výsledného kódu. Zároveň je jejich záměrem vývojářský komfort a přehlednost našeho kódu.

2. Funkce pro formátování operandů.

   - `label()`
   - `type()`
   - `variable()`
   - `literal()`
   - `symbol()`

   Každá funkce odpovídá jednomu typu operandu ve výstupním kódu. Tyto funkce na základě daných struktur našeho překladače vytvoří daný operand ve správném formátu.

3. Funkce pro tvorbu jazykových kontrukcí výstupního kódu.

   - `generate_func_header()`
   - `generate_func_end()`
   - `generate_builtin_func_call()`
   - `generate_if_start()`
   - `generate_elseif_else()`
   - `generate_elseif_if()`
   - `generate_else()`
   - `generate_if_end()`
   - `generate_while_start()`
   - `generate_while_condition()`
   - `generate_while_end()`
   - `generate_implicit_init()`
   - `generate_temp_pop()`
   - `generate_temp_push()`
   - `generate_nil_coelacing()`
   - `generate_string_concat()`

   Funkce reprezentují jednotlivé konstrukce (nebo jejich části). Fungují jako jakési šablony pro výstupní kód.

Vzhledem k naší implementace překladače, která pracuje s voláním funkce jako s výrazem, jsme podobný přístup aplikovali i v generátoru. Vyrazy jsou vyhodnocovány v datovém zásobníku - je-li to možné. Konstrukce, jako například `if-then` předpokládají, že vyhodnocení jejich přidruženého výrazu je na vrcholu zásobníku.

Generátor zároveň obsahuje implementace vestavěných funkcí, které jsou generovány ”ad-hoc”způsobem vložení šablony této implementace na místo závolání této funkce.

3 Algoritmy a datové struktury

V našem projektu jsme používali některé známé postupy na zpracování jazyků, které doplňovaly hlavní struktury programu. S velkou částí z nich jsme se seznámili v předmětu IAL, z něhož jsme využili jak nabyté zkušenosti, tak i některé implementace.

3.1 Tabulka symbolů

Tabulku symbolů jsme implementovali pomocí tabulky s rozptýlenými položkami. Její variací se stala tabulka symbolů v souborech `symtable.c` a `symtable.h`. Tabulka symbolů byla použita při tvorbě zásobníku proměnných a funkcí, kdy se jednotlivé úrovně skládají z tabulek symbolů.

3.2 Zásobník

Implementace zásobníku je parametrizována makry, což umožnilo vytvořit zásobník pro různé typy dat. Prvky zásobníku jsou reprezentovány spojovým seznamem, kde každý prvek obsahuje data daného typu a ukazatel na následující prvek. Tím, že lze makra použít pro různé datové typy, je možné použít "jeden"zásobník ve více částech programu. Zásobník používáme pro tabulku symbolů, pro lexikální analyzátor (na vrácení tokenů), v PSA na ukládání informací o výrazech a v generátoru pro pomocnou indexaci kontrukcí.

4 Komunikace a práce v týmu

Celý tým se pravidelně potkával v internetových videokonferencích (*skrze službu Discord*), kde jsme navzájem sdílely naše pokroky v práci na projektu. Nejužitečnějšími se, zejména ze začátku, staly osobní setkání - bylo třeba řádně prodiskutovat teoretickou stránku problému, před samotným psaním kódu.

4.1 Rozdělení práce mezi členy týmu

- **Anastasia Butok** (xbutok00)
  - zpracovávání chyb (*error.c*)
  - syntaktická a sémantická analýzu volání funkcí (*function_calling.c*)
  - precedenční syntaktická analýza - hlavní funkce (*psa.c*)
  - precedenční tabulka (*ptable.c*)
  - derivace na základě syntaktických pravidel v PSA (*rule_handle.c*)
  - testovací skripty - syntaktické, sémantické i kompletní s interpretací přeloženého jazyka (*/tests/*)
- **Simona Valkovská** (xvalko00)
  - lexikální analyzátor (*scanner.c*)
  - chybové kódy (*error.h*)
  - tvorba LL-gramatiky
- **Tomáš Hobza** (xhobza03)
  - makra pro barevný výstup (*colorful_print.h*)
  - zpracovávání chyb (*error.c*)

  - generátor výsledného kódu (*generator.c*)
  - čtení a kontrola následujících tokenů a výpis v PSA (*io.c*)
  - hlavní funce programu (*main.c*)
  - precedenční syntaktická analýza - hlavní funkce (*psa.c*)
  - precedenční syntaktická analýza - sémantická kontrola (*psa.c*)
  - multifunkční zásobník a zásobníkové operace pomocí maker (*stack.c*)
  - podpůrné funkce pro celý program (*utils.c*)
  - tabulka symbolů (*symtable.c*)
- **Jakub Všetečka** (xvsete00)
  - makra pro debugování (*debug.h*)
  - stavový automat sémantických kontrol (*parser_control.c*)
  - podpůrné funkce rekurzivního syntaktického analyzátoru (*parser_utils.c*)
  - rekurzivní syntaktický analyzátor (*parser.c*)
  - sémantické kontroly (*semantic.c*)
  - tvorba LL-gramatiky
  - testovací skripty - syntaktické, sémantické i kompletní s interpretací přeloženého jazyka (*/tests/*)

5 Závěr

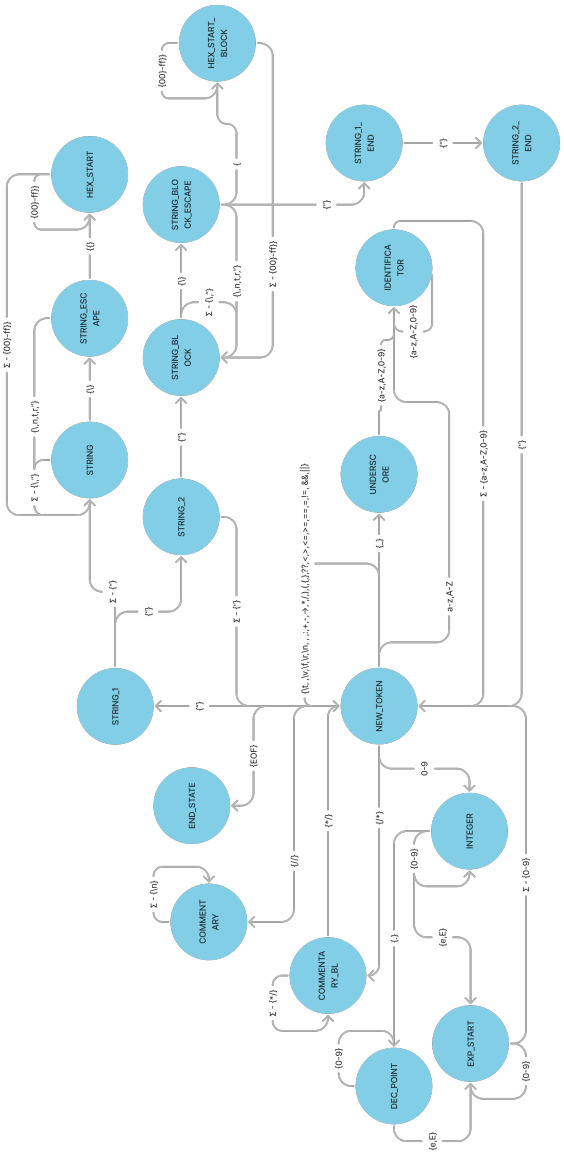
Tento projekt nám blíže ukázal skutečnou práci překladače a zároveň jsme si mohli vyzkoušet to, jak je občas složitá práce v týmu.

6 Zdroje

Informace a znalosti potřebné pro řešení tohoto projektu jsme získaly pouze na přednáškách a z materiálů na StudISu.

7 Přílohy

7.1 Diagram konečného automatu lexikálního analyzátoru



Obrázek 3: Diagram konečného automatu lexikálního analyzátoru

7.2 Precedenční tabulka

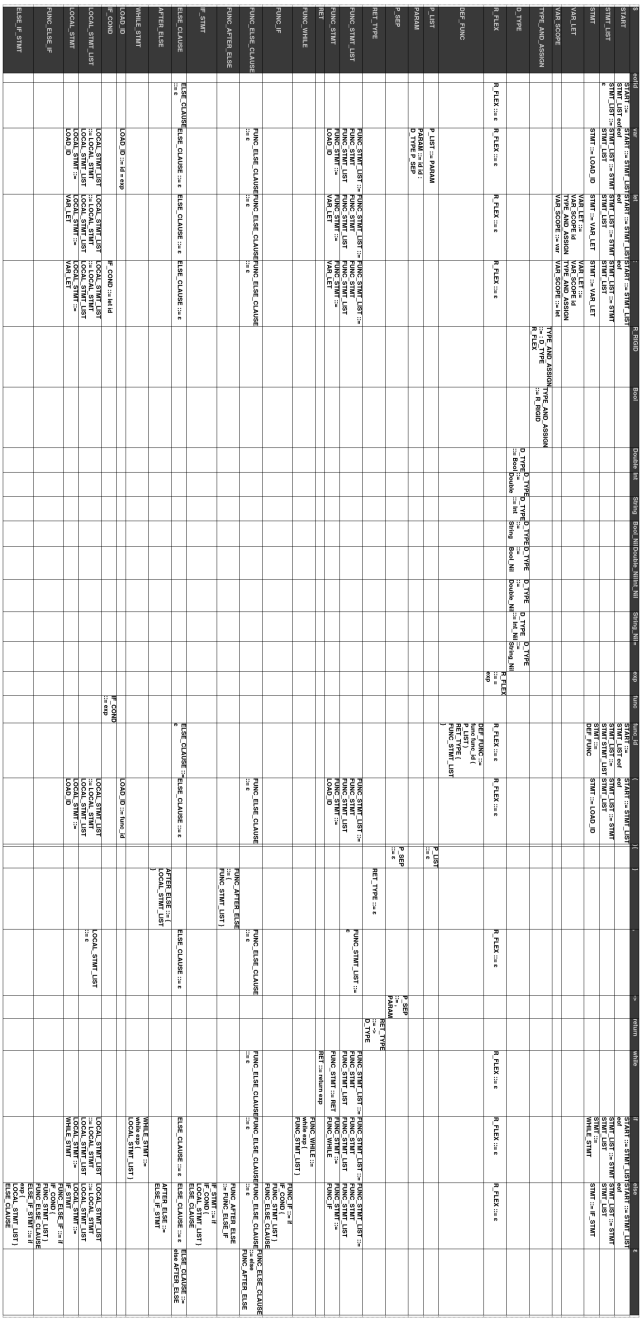
na vstupu

na zásobníku	!	*/	+-	REL	LOG	??	i	(	)	$
!	-	>	>	>	>	>	<	<	>	>
*/	<	>	>	>	>	>	<	<	>	>
+-	<	<	>	>	>	>	<	<	>	>
REL	<	<	<	>	>	>	<	<	>	>
LOG	<	<	<	<	>	>	<	<	>	>
??	<	<	<	<	<	>	<	<	>	>
i	>	>	>	>	>	>	-	-	>	>
(	<	<	<	<	<	<	<	<	=	-
)	>	>	>	>	>	>	-	-	>	>
$	<	<	<	<	<	<	<	<	-	-

REL: == != < > <= >= **LOG:** && ||

Obrázek 4: precedenční tabulka

7.3 Tabulka LL – gramatiky



Obrázek 5: Tabulka LL-gramatiky

Symbol	Produkce
START	→ STMT_LIST eof
STMT_LIST	→ ϵ
STMT_LIST	→ STMT STMT_LIST
STMT	→ VAR_LET
STMT	→ DEF_FUNC
STMT	→ IF_STMT
STMT	→ LOAD_ID
STMT	→ WHILE_STMT
VAR_LET	→ VAR_SCOPE id TYPE_AND_ASSIGN
VAR_SCOPE	→ var
VAR_SCOPE	→ let
TYPE_AND_ASSIGN	→ : D_TYPE R_FLEX
TYPE_AND_ASSIGN	→ R_RIGID
D_TYPE	→ Bool
D_TYPE	→ Double
D_TYPE	→ Int
D_TYPE	→ String
D_TYPE	→ Bool_Nil
D_TYPE	→ Double_Nil
D_TYPE	→ Int_Nil
D_TYPE	→ String_Nil
R_FLEX	→ ϵ
R_FLEX	→ = exp
DEF_FUNC	→ func func_id (P_LIST) RET_TYPE { FUNC_STMT_LIST }
P_LIST	→ PARAM
P_LIST	→ ϵ
PARAM	→ id : D_TYPE P_SEP
P_SEP	→ ϵ
P_SEP	→ , PARAM
RET_TYPE	→ ϵ
RET_TYPE	→ -> D_TYPE
FUNC_STMT_LIST	→ FUNC_STMT FUNC_STMT_LIST
FUNC_STMT_LIST	→ ϵ
FUNC_STMT	→ VAR_LET
FUNC_STMT	→ RET
FUNC_STMT	→ FUNC_WHILE
FUNC_STMT	→ LOAD_ID
FUNC_STMT	→ FUNC_IF
RET	→ return exp
FUNC_WHILE	→ while exp { FUNC_STMT_LIST }
FUNC_IF	→ if IF_COND { FUNC_STMT_LIST } FUNC_ELSE_CLAUSE
FUNC_ELSE_CLAUSE	→ ϵ
FUNC_ELSE_CLAUSE	→ else FUNC_AFTER_ELSE
FUNC_AFTER_ELSE	→ FUNC_ELSE_IF
FUNC_AFTER_ELSE	→ { FUNC_STMT_LIST }
IF_STMT	→ if IF_COND { STMT_LIST } ELSE_CLAUSE
ELSE_CLAUSE	→ else AFTER_ELSE
ELSE_CLAUSE	→ ϵ
AFTER_ELSE	→ ELSE_IF_STMT
AFTER_ELSE	→ { STMT_LIST }
WHILE_STMT	→ while exp { STMT_LIST }
LOAD_ID	→ id = exp
LOAD_ID	→ func_id
IF_COND	→ exp
IF_COND	→ let id
STMT_LIST	→ STMT STMT_LIST
STMT_LIST	→ ϵ
STMT	→ VAR_LET
STMT	→ LOAD_ID
STMT	→ IF_STMT
STMT	→ WHILE_STMT
FUNC_ELSE_IF	→ if IF_COND { FUNC_STMT_LIST } FUNC_ELSE_CLAUSE
ELSE_IF_STMT	→ if exp { STMT_LIST } ELSE_CLAUSE

Tabulka 1: Přehled LL gramatiky