VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Dokumentace k projektu

Implementace překladače imperativního jazyka IFJ22

Tým xfedor14, varianta TRP

Tatiana Fedorova	**xfedor14**	**25%**
Vadim Goncearenco	xgonce00	25%
Vladyslav Kovalets	xkoval21	25%
Aleksandr Shevchenko	xshevc01	25%

6. prosince 2022

Obsah

1 Úvod

Hlavním cílem projektu byla implementace překladače imperativního jazyka IFJ22, který je zjednodušenou podmnožinou jazyka PHP. Byl vytvořen program v jazyce C, který načítá zdrojový kód zapsaný ve zdrojovém jazyce IFJ22 a překládá jej do cílového jazyka IFJcode22. Schéma projektu: 1.

2 Návrh a implementace

2.1 Lexikální analýza

Překladač jsme začali vytvářet pomocí lexikálního analyzátoru, ale před jeho implementací bylo zapotřebí navrhnout konečný stavový automat (obrázek 2).

Hlavní funkcí této části je `scanner_get_next_token()`, která postupně načítá znak po znaku ze vstupního souboru a přepíná se mezi jednotlivými stavy pomoci diagramu. Každý koncový stav reprezentuje, že byl načten jeden token, který se ukládá do struktury `Token` s odpovídajícím typem a případně atributem. Pokud funkce skončí v nekoncovém stavu, tak vstupní program je chybný. Lexikální analýza se skládá ze souborů `scanner.(c/h)`, `string_t.(c/h)`.

2.2 Syntaktická analýza

Syntaktická analýza v našem řešení vychází přímo z LL-gramatiky a intenzivně využívá speciální pomocná makra (`macros.h`) pro stručný zápis stále se opakující funkcionality jako načtení následujícího tokenu, ověření jeho typu a vracení kódu syntaktické chyby v případě že typ nesedí s gramatickými pravidly.

Také pro zvýšení přehledností kódu jsou k dispozici makra `RULE_OPEN`, `RULE_CLOSE`, která zajišťují přítomnost definice proměnné obsahující návratový kód a vracení kódu `SUCCESS` v případě úspěchu.

Jedním z problémů v rámci syntaktické analýzy, na který jsme se narazili během implementace parseru (`parser.h`), byla nutnost ověřit nejen aktuální token, ale i následující. V podstatě bylo to nutné jenom v jediném případě (při rozhodovaní mezi pravidly 6. a 10.). I když to způsobilo určité potíže, přivedlo nás k nápadu udělat si pomocnou datovou strukturu ADT vector(3.1), která se pak velmi hodila pro implementaci funkcionality *„function pass“ 2.3.1* a *„condition pass“ 2.3.2*, zásobníku symbolů v `expression.c` a zpracování neomezeného počtu parametrů vestavěné funkce `write`.

2.3 Sémantická analýza

Při implementaci sémantické analýzy jsme vycházeli z popisu jednotlivých chyb uvedených v zadání projektu. Chyby označené za běhové jsme se snažili detekovat až za běhu, pomocí generovaní příslušného kódu, dynamicky ověřujícího určitou podmínku (např. použití nedefinované proměnné). Avšak sémantické chyby, které se dost jednoduše daly odchytit při kompilaci (např. špatný počet/typ parametrů funkce), jsme se nesnažili takhle dynamicky kontrolovat.

Pomocná datová struktura `vec_t`, použitá pro ukládaní přijatých tokenů, hodně usnadnila realizaci pokročilejších kontrol, které jsou popsaný v následujících podsekcích.

2.3.1 Function pass

Jedním z netriviálních problémů, na který jsme se narazili v souvislosti se sémantickou analýzou, byla možnost volaní funkce před její definicí. Pro realizaci kontroly v tomto případě jsme se rozhodli udělat dodatečný průchod (*„function pass“*) na začátku kompilace. Během dodatečného průchodu jsou čteny všechny tokeny zdrojového kódu, ale zpracovávány jenom signatury funkce a všechna data o každé funkci jsou uložena do globální tabulky symbolů.

Během tohoto průchodu jsou všechny přijaté tokeny uloženy do fronty (ADT vector), ze které jsou pak čteny během „normálního“ průchodu(*„main pass“*). Toto řešení zajišťuje, že v moment, kdy se narazíme na volaní funkce, jejíž definice existuje někde ve zdrojovém kódu, budeme mít všechnu potřebnou informaci pro sémantické kontroly spojené s jejím voláním.

2.3.2 Condition pass

Sémantické kontroly tykající se definice proměnné se provádí dynamicky, proto jsme z technických důvodu museli přidat jeden průchod parseru spouštěný v moment, kdy narazíme na klíčové slovo „if“ nebo „while“. Tento průchod (*„conditional pass“*) zpracovává celý obsah „if“/„while“ až po poslední uzavírací závorku `}` a generuje instrukci `DEFVAR` (bez následujícího přiřazení hodnoty) pro každou nalezenou proměnnou. Přítomnost instrukce `DEFVAR` odpovídající každé proměnné je nezbytná pro dynamickou kontrolu definice proměnné, jinak když interpret (`ic22int`) se narazí na její identifikátor, vrací chybu jako kdyby nalezl neznámý symbol.

2.4 Analýza výrazů

V případě, že se při parsingu dat program narazí na výraz, potom se hned zavolají funkce pro zpracování výrazů. Tyto funkce se nacházejí v souborech `expression.c` a `expression.h`.

Celá analýza je postavena na precedenční tabulce 3. Operátory, závorky, identifikátory, symbol dolaru jsou umístěny na osách X (udává aktuálně zpracovaný symbol) a Y (udává symbol, který je teď na vrcholu zásobníku). Na začátku parsingu výrazů se na vrchol zásobníku ukládá symbol dolaru. Podle toho, který symbol aktuálně zpracováváme a pomocí funkce `index_info` se získá informace o typu operace, kterou je nutné provést. Buď se jedná o operaci `SHIFT` (v tab. 3 „$<$“), při níž se vkládá znak `REDUCE` před top terminál na zásobníku a aktuálně zpracovaný symbol se uloží na vrchol, nebo se jedná o operaci `REDUCE` (v tab. 3 „$>$“), při níž se uskuteční redukce na zásobníku podle existujícího pravidla. Pokud se jedná o operaci `EQUAL` (v tab. 3 „$=$“), potom se aktuálně zpracovaný symbol umísti na vrchol zásobníku. V případě, že se jedná o operaci `NONE` (v tab. 3 prázdná buňka), kde na obou osách jsou symboly dolaru, potom je analýza výrazu ukončena.

Během redukce se také určuje typ výrazu a kontroluje se zda mezi operandy nebyla použita nedefinovaná proměnná, pak se zavolá funkce generátoru kódu, která na základě typu operace vygeneruje kód obsahující volaní interní funkci, která až za běhu realizuje funkcionalitu příslušné operace a všechny sémantické kontroly/konverze spojené s ní (více o tom v kapitole 2.5).

2.5 Generování cílového kódu

Generátor cílového kódu je implementován v souborech `codegen.c` a `codegen.h`. Generování provádíme průběžně, a pokud za běhu programu nenastává žádná chyba, vypisujeme kód na standardní výstup.

Na začátku generovaného kódu vždy máme úvodní řádek pro IFJcode22, definici pomocných proměnných, návěští pro případy chybného návratu, skok do hlavního těla programu. Pro jednoduchost

jsme se rozhodli všechny vestavěné funkce napsat v jazyce IFJcode22 a tyto kódy také přidat na začátek výsledného kódu. Stejným způsobem jsme realizovali všechny logické a aritmetické operátory a spojené s nimi implicitní typové konverze. Všechny tyto „přípravné“ funkce jsou implementovány jako makra v souborech `builtins.h` a `internals.h`.

Pro volání funkcí generátoru kódu z jiných částí překladače jsme vytvořili makro `CODEGEN`, které jako argumenty přijímá jméno funkce, kterou voláme, a její parametry. Jádrem generátoru kódu je interní paměť, což je dynamický řetězec (viz sekce 3.2), do kterého uchováváme průběžný kód pomocí makra `EMIT` a jemu podobných maker pro výpis kódu s přechodem na nový řádek (`EMIT_NL`) a pro výpis celého čísla (`EMIT_INT`).

Jedním z největších problémů během vypracování generátoru cílového kódu byla generace unikátních návěští pro konstrukce „if“ a „while“. Vyřešili jsme to tak, že k tímto návěštím přidali 2 pomocné indexy. První značí vnořenost, čili hloubku aktuální konstrukce, druhý uvádí celkové pořadí této konstrukce v celém programu.

3 Pomocné datové struktury

3.1 Vektor

Jako pomocnou datovou strukturu jsme využili ADT **vektor** (implementace v `vector_t.h`), implementovaný podobně dvojsměrně vázanému seznamu, což umožňuje použití takových operací jako `push_front`, `push_back` resp. `pop_front`, `pop_back`.

Tento ADT může byt definován pro libovolný datový typ díky speciálním makrům `GENERATE_VECTOR_DECLARATION`, `GENERATE_VECTOR_DEFINITION`. Tata makra automaticky generují patřičnou definici s uvedeným prefixem pro každou funkci rozhraní, což umožňuje vyhnout se kolizím v názvech.

V projektu typ **vektor** je využít jednak v parseru (`parser.h`) pro ukládaní a následné čtení přijatých tokenů, a v precedenční analýze (`expression.h`) jako zásobník symbolů.

3.2 Dynamický řetězec

Při práci nad projektem jsme skoro všude potřebovali řetězce se stále měnící se délkou. Proto jsme vytvořili ADT **dynamický řetězec**, jehož implementace je v souborech `string_t.c` a `string_t.h`.

Struktura `str_t` zahrnuje ukazatel na samotný řetězec (posloupnost charů), jeho aktuální délku a velikost alokované paměti. Na začátku jsme nastavili velikost řetězce na 16 znaků, alokační krok na 2, což znamená, že při překročení řetězcem alokovaného prostoru dojde ke dvojnásobnému zvětšení alokované paměti.

Nad **dynamickým řetězcem** můžeme provádět následující operace: inicializaci, mazání, konkatenaci, kopírovaní, porovnání řetězců a přidání znaku na konec řetězce.

3.3 Tabulka s rozptýlenými položkami

Součástí naší varianty projektu byla implementace tabulky symbolů pomocí **tabulky s rozptýlenými položkami (TRP)**. Jelikož teoretický počet položek může být nekonečný, použili jsme variantu s explicitním zřetězením synonym, s jejich propojením pomocí jednosměrně vázaného seznamu.

TRP je popsána v souborech `symtable.c` a `symtable.h`. Základní strukturou je `TSymtable`, která obsahuje pole ukazatelů na `TItem` o velikosti `MAX_SYMTABLE_SIZE`. Položkami tabulky

je `TItem`, což je struktura obsahující ukazatel na následující položku, identifikátor funkce anebo proměnné a samotná data ve struktuře `TData`.

Hašovací funkci jsme převzali z 2. projektu předmětu IJC[1]. Ona se pro naše účely projevila jako zcela vhodná. Danou funkci jsme upravili tak, aby operací modulo vracela index do mapovacího pole. Pro větší efektivitu jeho velikost musí být prvočíslem, takže jsme zvolili číslo 10007.

Nad TRP jsme implementovali všechny základní operace, např. inicializaci, vyhledávání, vkládání a mazání prvků. Navíc máme funkci `symtable_add_param`, která na základě typů parametrů funkce přidává písmena 'f', 'i', 's' anebo 'u' (což znamená float, int, string a undefined) do položky `param_str` (typ `str_t`, viz 3.2) struktury `TData` (obsahuje atributy nutné pro provádění sémantických kontrol).

4 Testování

Pro průběžné testování jsme používali testy vytvořené jinými studenty v jazyce Python, a svoje pomocné funkce, implementované v souborech `debug.c/h`. Také jsme použili pomocné makro obalující `assert` a vypisující informaci o souboru, funkci a řádku, kde `assert` selhal.

5 Výčet souborů

Ve výsledku máme implementovány následující soubory:

Soubory	Význam
`builtins.h`	Vestavěné funkce
`codegen.(c|h)`	Generátor IFJcode22
`debug.(c|h)`	Testování
`errors.(c|h)`	Výpis chyb
`internals.h`	Makra pro výrazy
`macros.h`	Pomocná makra
`main.c`	Start programu

Soubory	Význam
`parser.(c|h)`	Parser
`expression.(c|h)`	Prec. analýza
`scanner.(c|h)`	Scanner
`string_t.(c|h)`	Dyn. řetězec
`sybtable.(c|h)`	TRP
`vector_th.`	Vektor

Tabulka 1: Stručný popis členění řešení

6 Týmová práce

6.1 Použité aplikace a zdroje

Pro komunikaci mezi sebou jsme především používali sociální síť Telegram. Také jsme vytvořili vlastní kanál na Discordu pro videohovory a uložení již hotových tabulek a schémat.

Samotný kód jsme psali ve Visual Studio Code a CLion. Pro správu verzí kódu jsme používali repositář na GitHubu. Dobrou pomůckou pro kreslení diagramu konečného automatu byl Diagrams.net. Tato dokumentace je napsána pomocí Overleaf.

[1]Odkaz, uvedený v zadání projektu IJC: `http://www.cse.yorku.ca/~oz/hash.html`

6.2 Rozdělení práce

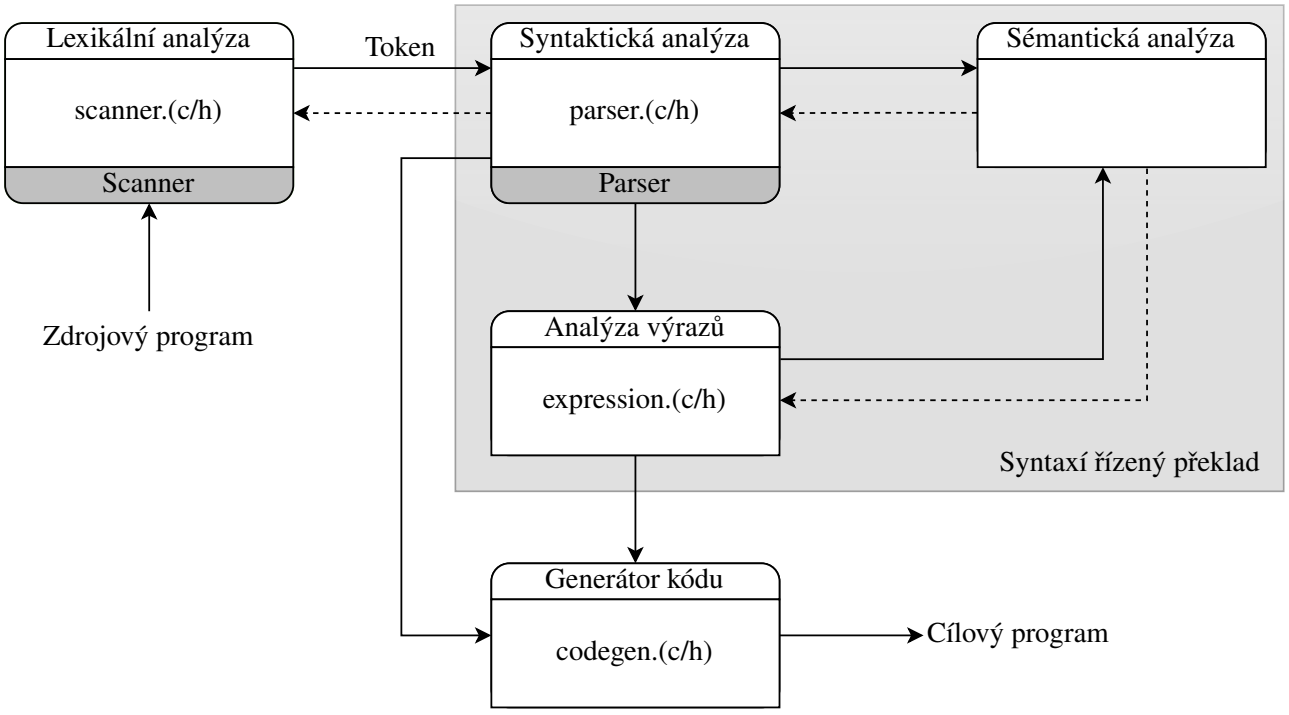
Náš tým byl vytvořen již v létě, a začali jsme pracovat nad projektem hned po zveřejnění zadání. Jednotlivým částem projektu byli přiřazeni 1 anebo 2 členi týmu, v závislosti na složitosti. Pak jsme společně ověřovali funkčnost a shodu částí se zadáním. Tabulka 2 ukazuje rozdělení práce mezi členy týmu:

Člen týmu	**Části projektu**
Tatiana Fedorova	Vedení týmu, zpracování výrazů, dokumentace
Vadim Goncearenco	Syntaktická analýza, sémantická analýza, testování, dokumentace
Vladyslav Kovalets	Lexikální analýza, testování, dokumentace
Aleksandr Shevchenko	Tabulka symbolů, generátor kódu, dokumentace

Tabulka 2: Rozdělení práce mezi členy týmu

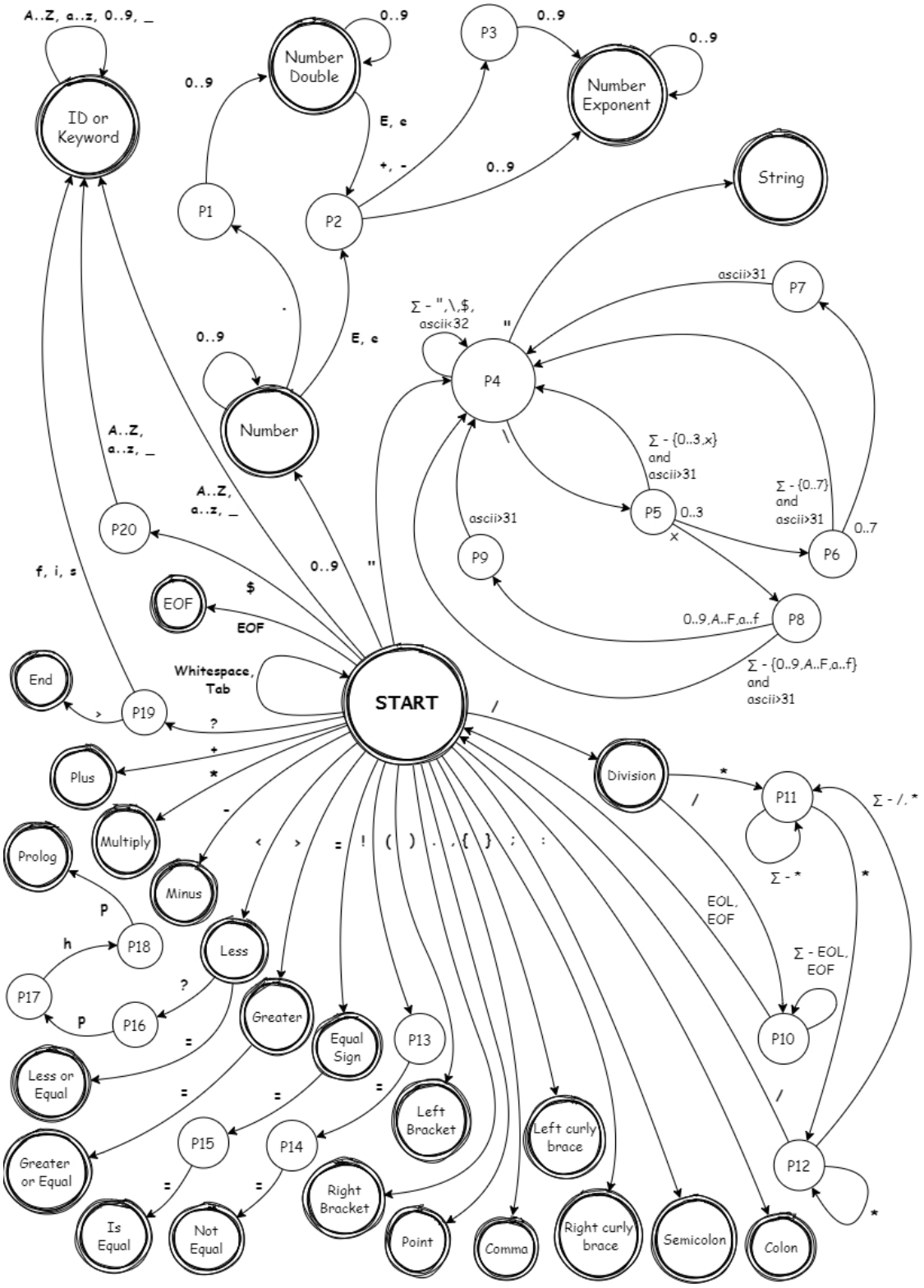
7 Pomocné tabulky a grafy

7.1 Schéma projektu



Obrázek 1: Schéma projektu

7.2 Diagram konečného automatu



Obrázek 2: Konečný automat pro lexikální analýzu

LEGENDA:
P1 STATE_NUMBER
P2 STATE_NUMBER_EXPONENT_START
P3 STATE_NUMBER_EXPONENT_SIGN
P4 STATE_STRING_START
P5 STATE_STRING_BACKSLASH
P6 STATE_STRING_BACKSLASH_ONE_TWO_THREE
P7 STATE_STRING_BACKSLASH_ZERO_TO_SEVEN
P8 STATE_STRING_BACKSLASH_HEX
P9 STATE_STRING_BACKSLASH_HEX_FIRST
P10 STATE_COMMENT
P11 STATE_COMMENT_BLOCK_START
P12 STATE_COMMENT_BLOCK_END
P13 STATE_NOT_EQUAL_START
P14 STATE_NOT_EQUAL_END
P15 STATE_IS_EQUAL
P16 STATE_PROLOG_ONE
P17 STATE_PROLOG_TWO
P18 STATE_PROLOG_THREE
P19 STATE_QUESTION_MARK
P20 STATE_DOLLAR

7.3 Precedenční tabulka

	+ - .	* /	=== !==	r	(	)	id	$
+ - .	>	<	>	>	<	>	<	>
*** /**	>	>	>	>	<	>	<	>
=== !==	<	<	>	<	<	>	<	>
r	<	<	>	>	<	>	<	>
(	<	<	<	<	<	=	<	
)	>	>	>	>		>		>
id	>	>	>	>		>		>
$	<	<	<	<	<		<	

r - reláční operátor (<, >, <=, >=)

Tabulka 3: Precedenční tabulka pro zpracování výrazů

7.4 LL-tabulka

	<?php	function	$ID	if	while	return	ID	INT_VAL	FLOAT_VAL	STR_VAL	null	?>	EOF	int	float	string	.	void	$
begin	1																		
program		2	3	3	3	3	3	3	3	3	3	3	3						
end												4	5						
statement			6,10	7	8	9	10	10	10	10	10								10,11
rvalue			13				12	13	13	13	13								13
params														14	14	14			15
param_n																	16		17
func_type														19	19	19		18	
args			20					20	20	20	20								21
arg_n																	22		23
term			28					24	25	26	27								
type														29	30	31			

Tabulka 4: LL-tabulka pro syntaktickou analýzu

7.5 LL-gramatika

1. <begin> -> <?php declare (strict_types = 1) ; <program> <end>
2. <program> -> function ID (<params>) : <func_type> { <statement> } <program>
3. <program> -> <statement>
4. <end> -> ?>
5. <end> -> EOF
6. <statement> -> $ID = <rvalue> ; <program>
7. <statement> -> if(<expression>){<statement>} else {<statement>}<program>
8. <statement> -> while (<expression>){<statement>}<program>
9. <statement> -> return <expression> ; <program>
10. <statement> -> <rvalue> ; <program>
11. <statement> -> ε
12. <rvalue> -> ID (<args>)
13. <rvalue> -> <expression>
14. <params> -> <type> $ID <param_n>
15. <params> -> ε
16. <param_n> -> , <type> $ID <param_n>
17. <param_n> -> ε
18. <func_type> -> void
19. <func_type> -> <type>
20. <args> -> <term> <arg_n>
21. <args> -> ε
22. <arg_n> -> , <term> <arg_n>
23. <arg_n> -> ε
24. <term> -> INT_VAL
25. <term> -> FLOAT_VAL
26. <term> -> STR_VAL
27. <term> -> null
28. <term> -> $ID
29. <term> -> int
30. <term> -> float
31. <term> -> string