

Projektová dokumentace
Implementace překladače imperativního jazyka IFJ22
Tým xmoise01, varianta BVS

6. prosince 2022

Nikita Moiseev	(xmoise01)	25 %
Maksim Kalutski	(xkalut00)	25 %
Elena Marochkina	(xmaroc00)	25 %
Nikita Pasynkov	(xpasyn00)	25 %

Obsah

1 Úvod	1
2 Návrh a implementace	1
2.1 Lexikální analýza	1
2.2 Syntaktická analýza	1
2.3 Sémantická analýza	3
2.4 Optimalizace kódu	3
2.5 Generování cílového kódu	4
3 Práce v týmu	4
3.1 Způsob práce v týmu	4
3.1.1 Vývoj	4
3.1.2 Komunikace	4
3.2 Rozdělení práce mezi členy týmu	5
4 Závěr	5
A Diagram konečného automatu specifikující lexikální analyzátor	6
B LL – gramatika	7
C LL – tabulka	8
D Precedenční tabulka	9

1 Úvod

Cílem projektu bylo vytvořit program v jazyce C, který načte zdrojový kód zapsaný ve zdrojovém jazyce IFJ22, jenž je zjednodušenou podmnožinou jazyka PHP a přeloží jej do cílového jazyka IFJcode22 (mezikód).

Program funguje jako konzolová aplikace, které načítá zdrojový program ze standardního vstupu a generuje výsledný mezikód na standardní výstup nebo v případě chyby vrací odpovídající chybový kód.

2 Návrh a implementace

Projekt jsme sestavili z několika námi implementovaných dílčích částí, které jsou představeny v této kapitole. Je zde také uvedeno, jakým způsobem spolu jednotlivé dílčí části spolupracují.

2.1 Lexikální analýza

Při tvorbě překladače jsme začali implementací lexikální analýzy. Hlavní funkce této analýzy je `get_next_token`, pomocí níž se čte znak po znaku ze zdrojového souboru a převádí na strukturu `token`, která se skládá z typu a hodnoty. Typy tokenů jsou `EOF`, speciální znaky, speciální závorky PHP, identifikátory, klíčová slova, datové typy a také aritmetické, relační a logické operátory a operátor přiřazení a ostatní znaky, které mohou být použity v jazyce IFJ2022. Hodnota atributu je `value`. Pokud je typ tokenu identifikátor, pak bude atribut daný identifikátor, když by byl typ tokenu klíčové slovo, přiřadí atributu dané klíčové slovo, pokud číslo, atribut bude ono číslo. S takto vytvořeným tokenem poté pracují další analýzy.

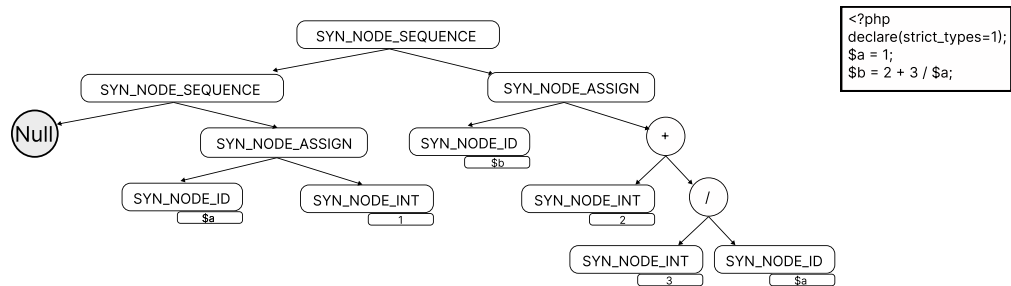
Celý lexikální analyzátor je implementován jako deterministický konečný automat podle předem vytvořeného diagramu 6. Konečný automat je v jazyce C jako jeden nekonečně opakující se `switch`, kde každý případ `case` je ekvivalentní k jednomu stavu automatu. Pokud načtený znak nesouhlasí s žádným znakem, který jazyk povoluje, program je ukončen a vrací chybu `1 LEXICAL ERROR CODE 1`. Jinak se přechází do dalších stavů a načítají se další znaky, dokud nemáme hotový jeden token, který potom vrátíme a ukončíme tuto funkci.

2.2 Syntaktická analýza

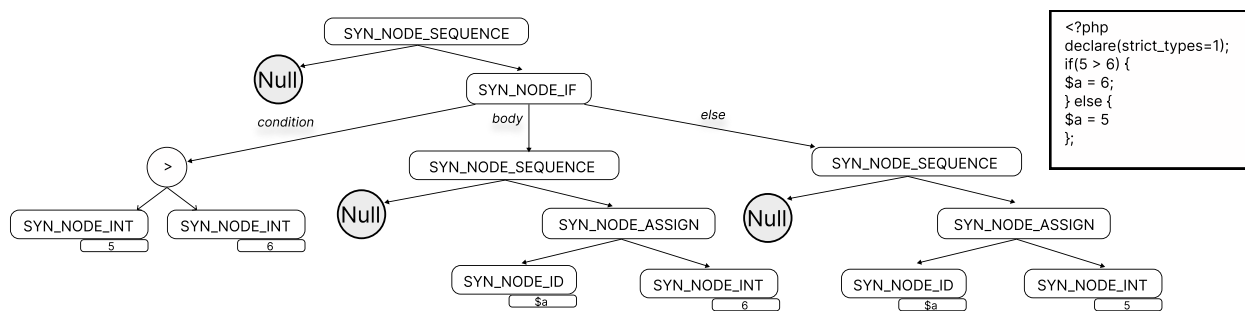
Nejdůležitější částí celého programu je syntaktická analýza. Syntaktická analýza je implementována v souboru `syntax_analyzer.c`, a její rozhraní pro implementaci v `syntax_analyzer.h`.

Syntaktická analýza je implementována pomocí rekurzivního sestupu na základě LL gramatiky 2. Syntaktický analyzátor zpracovává všechny části LL gramatiky na výrazy podle pravidel v LL - tabulce 3. Úkolem syntetického analyzátoru je sestavit abstraktní syntaktický strom na základě seznamu tokenů. Strom je postaven tak, že je nastavena priorita operací a ve výsledku můžeme načtením tohoto stromu spustit kód v požadovaném pořadí. Pokud je nalezena chyba, překladač dokončí kontrolu, vymaže alokovanou paměť a zobrazí chybovou hlášku na standardní chybový vstup. Syntaktický analyzátor nijak neupravuje tokeny a všechny chyby jsou detekovány až v průběhu sémantické analýzy. Po vytvoření abstraktního syntaktického stromu syntaktický analyzátor přejde do další fáze sémantické analýzy.

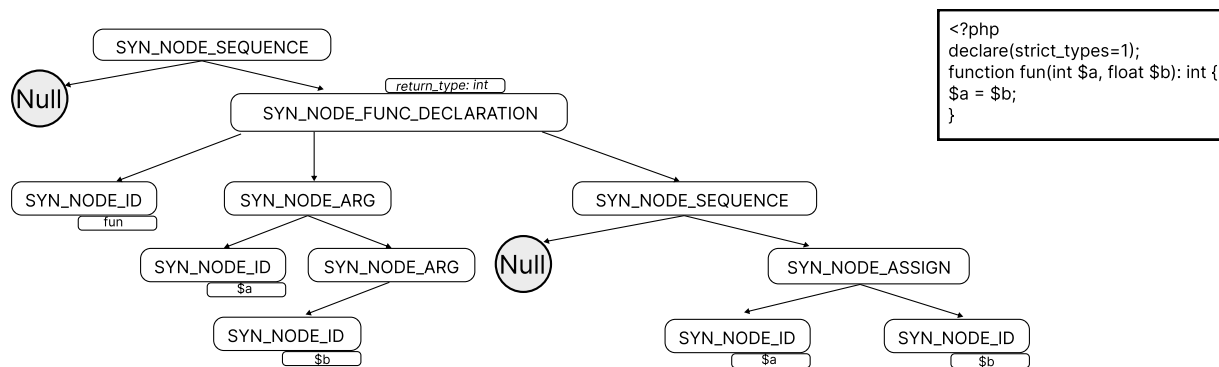
Příklad stromu pro syntaktickou analýzu je uveden na obrázcích 1- 4.



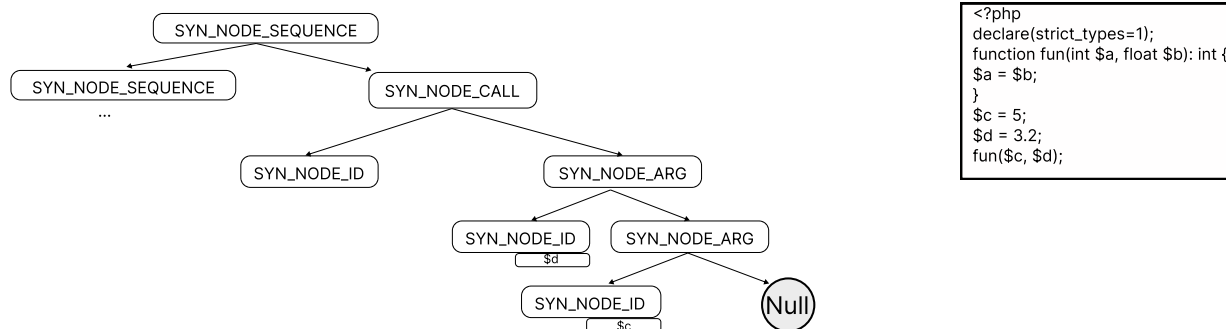
Obrázek 1: Abstraktní syntaktický strom přiřazení proměnných



Obrázek 2: Abstraktní syntaktický strom cyklů



Obrázek 3: Abstraktní syntaktický strom deklarací funkcí



Obrázek 4: Abstraktní syntaktický strom volání funkcí

2.3 Sémantická analýza

Sémantická analýza je implementována v souboru `semantic_analyzer.c` a její rozhraní pro implementaci v `semantic_analyzer.h`.

Sémantický analyzátor pracuje s abstraktním syntaktickým stromem, který vytvořil syntaktický analyzátor. Úkolem sémantického analyzátoru je kontrola sémantické správnosti zdrojového programu: kontrola deklarací, datových typů, seznamů parametrů apod. a to pomocí kontroly datových typů.

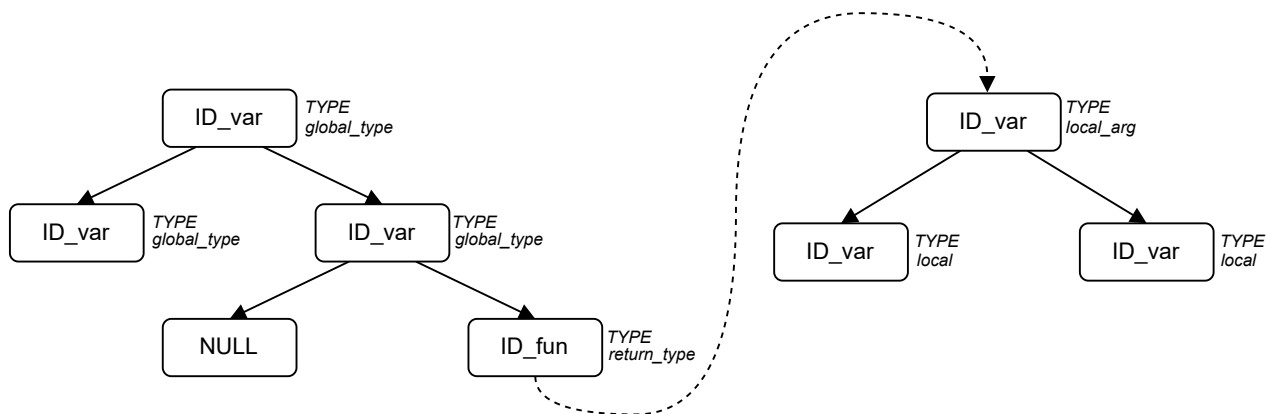
K tomu využívá datové struktury a funkce pro sémantické kontroly. Pokud při kontrole je nalezena chyba, překladač dokončí kontrolu, vymaže alokovanou paměť a zobrazí chybovou hlášku na standardní chybový vstup.

V souborech `symtable.c` a `symtable.h` je uložená tabulka symbolů.

Tabulka symbolů je binární vyhledávací strom, který obsahuje informace o všech identifikátorech kódu. Tabulka symbolů se skládá jak z globálního stromů (pro všechny globální proměnné), tak z lokálních stromů (pro argumenty a lokální proměnné funkcí).

Tabulka symbolů je uložena a okamžitě použita v sémantické analýze a slouží ke kontrole, zda daný identifikátor existuje a zda souhlasí jeho datový typ, případně návratová hodnota.

Příklad stromu pro tabulku symbolů je uveden na obrázku 5.



Obrázek 5: Tabulka symbolů

2.4 Optimalizace kódu

Optimalizace kódu je implementována v souboru `optimiser.c` a jeho rozhraní pro implementaci v `optimiser.h`.

Optimalizace kódu pracuje s abstraktním syntaktickým stromem a mění to.

Optimalizátor kódu se spustí ihned po sémantické analýze a před vygenerováním kódu.

Optimalizátor kódu provádí následující hlavní akce:

1. Optimalizuje matematické výrazy (například `$a = 5 + 9/3;` \rightarrow `$a = 8;`;

2. Přiřazování a změna hodnoty proměnné během programu
3. Optimalizuje nedosažitelné smyčky while a if
4. Odstraňuje nepoužívané proměnné

2.5 Generování cílového kódu

Generování cílového kódu je implementované v souboru `code_generator.c` a jeho rozhraní pro implementaci v `code_generator.h`. Generování kódu pracuje s abstraktním syntaktickým stromem po optimalizaci kódu.

Generování cílového kódu znamená generování mezikódu IFJcode22. Kód je generován na standardní výstup po dokončení všech analýz.

Na začátku generování jsou inicializovány potřebné datové struktury (které jsou na závěr uvolněny), vygenerována hlavička mezikódu, která zahrnuje potřebné náležitosti pro korektní interpretaci mezikódu a skok do hlavního těla programu. Poté jsou vygenerovány vestavěné funkce, které jsou zapsány přímo v jazyce IFJcode22.

Každá funkce mezikódu IFJcode22 je tvořena návěštím ve tvaru `$generate_funkce`.

Pak se spouští parser syntaktického stromu, dokud není nalezen jeden z uzlů: `SYN_NODE_ASSIGN`, `SYN_NODE_KEYWORD_IF`, `SYN_NODE_KEYWORD_WHILE`, `SYN_NODE_FUNCTION_DECLARATION`, `SYN_NODE_CALL`. Pro každý z uzlů jsou spuštěny různé funkce generování kódu.

Generování výrazů

Jednoduché výrazy jsou zpracovávány optimalizátorem a generovány na jednom řádku. U složitých výrazů, které optimalizátor nerozpozná, lze k výrazům přidat další proměnné pro provádění výpočtů.

Generování cyklů

Pokud je proměnná deklarována ve cyklu, bylo nutné ji před začátkem cyklu předdefinovat.

Generování funkcí

Pro funkce a jejich proměnné byl vytvořen lokální rámec. Pokud nejsou v kódu volány vestavěné funkce, generátor kódu je nevygeneruje.

3 Práce v týmu

3.1 Způsob práce v týmu

Na projektu jsme začali pracovat na začátku října. Práci jsme si dělili postupně, tj. neměli jsme od začátku stanovený kompletní plán rozdělení práce. Na dílčích částech projektu pracovali většinou dvojice členů týmu.

3.1.1 Vývoj

Veškerá naše práce byla rozdělena do sprintů, každý trval týden. Během sprintu musel každý jednotlivý člen týmu splnit určitý objem úkolu.

Pro správu souborů projektu jsme používali verzovací systém Git. Jako vzdálený repositář jsme používali *repositář na GitHubu*.

Git nám umožnil pracovat na více úkolech na projektu současně v tzv. větvích. Většinu úkolů jsme nejdříve připravili do větve a až po otestování a schválení úprav ostatními členy týmu jsme tyto úpravy začlenili do hlavní vývojové větve.

Pro testování jsme použili knihovnu pro testování jednotek *googletest*.

3.1.2 Komunikace

Komunikace mezi členy týmů probíhala převážně osobně nebo prostřednictvím aplikace Telegram.

V průběhu řešení projektu jsme měli i osobní setkání každý týden, kde jsme probírali a řešili problémy týkající se různých částí projektu. Pro plánování úkolů jsme použili webovou aplikaci *Notion*.

3.2 Rozdělení práce mezi členy týmu

Práci na projektu jsme si rozdělili rovnoměrně s ohledem na její složitost a časovou náročnost. Každý tedy dostal procentuální hodnocení 25 %. Tabulka 1 shrnuje rozdělení práce v týmu mezi jednotlivými členy.

Člen týmu	Přidělená práce
Nikita Moiseev	vedení týmu, organizace práce, dohlížení na provádění práce, konzultace, kontrola, lexikální analýza, syntaktická analýza, optimalizace kódu, testování, dokumentace
Maksim Kalutski	generování cílového kódu, testování, dokumentace
Elena Marochkina	implementace tabulky symbolů, syntaktická analýza, sémantická analýza, testování, dokumentace
Nikita Pasynkov	lexikální analýza, syntaktická analýza, sémantická analýza, testování, dokumentace

Tabulka 1: Rozdělení práce v týmu mezi jednotlivými členy

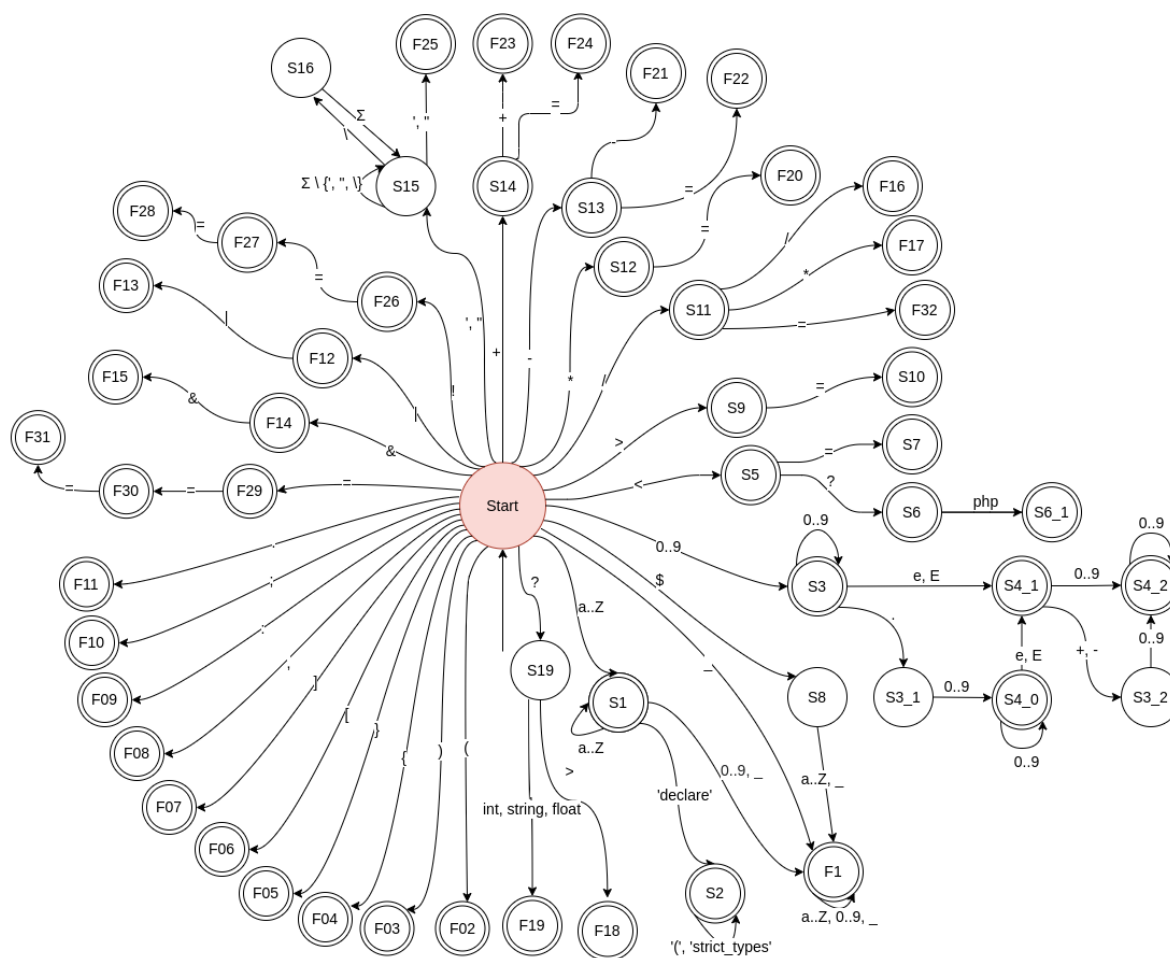
4 Závěr

Náš tým jsme měli sestaven brzy a pracovalo se nám společně velmi dobře.

V průběhu vývoje jsme se potýkali s menšími problémy týkajícími se nejasností v zadání, ale tyto jsme vyřešili díky fóru k projektu. Správnost řešení jsme si ověřili pomocí Google testy a pokusnému odevzdání, díky čemuž jsme byli schopni projekt ještě více odladit.

Tento projekt nám celkově přinesl spoustu znalostí ohledně fungování překladačů, prakticky nám objasnil probíranou látku v předmětech IFJ a IAL a přinesl nám zkušenosti s projekty tohoto rozsahu.

A Diagram konečného automatu specifikující lexikální analyzátor



S1: keyword and ID state
 S2: declare state
 S3: integer state
 S3_1, S3_2: float middle state
 S4_0, S4_1, S4_2: float state
 S5: less
 S6: open php bracket
 S6_1: open php bracket
 S7: less or equal
 S8: identifier state
 S9: greater
 S10: greater or equal
 S11: divide
 S12: multiply
 S13: minus
 S14: plus
 S15: string state
 S16: string escape state
 S17: not
 S18: not equal
 S19: optional question mark

F01: identifier
 F02: left parenthesis
 F03: right parenthesis
 F04: left curly brackets
 F05: right curly brackets
 F06: left square brackets
 F07: right square brackets
 F08: comma
 F09: colon
 F10: semicolon
 F11: concatenation
 F12: bitwise or
 F13: logical or
 F14: bitwise and
 F15: logical and
 F16: comment

F17: multiline comment
 F18: close php bracket
 F19: optional data type
 F20: multiply assign
 F21: decrement
 F22: minus assign
 F23: increment
 F24: plus assign
 F25: string
 F26: not
 F27: not equal
 F28: typed not equal
 F29: assign
 F30: equal
 F31: typed equal
 F32: divide assign

Obrázek 6: Diagram konečného automatu specifikující lexikální analyzátor

B LL – gramatika

1. `<prog> -> <?php <declare> <f-dec-stats> <stat-list> ?>`
2. `<prog> -> <? <declare> <f-dec-stats> <stat-list> ?>`
3. `<f-dec-stats> -> <f-dec-stat>`
4. `<f-dec-stats> -> <f-dec-stat> <f-dec-stats>`
5. `<f-dec-stat> -> function ID (<f-args>) : <f-type> { <stat-list> }`
6. `<f-dec-stat> -> function ID (<f-args>) { <stat-list> }`
7. `<f-type> -> int`
8. `<f-type> -> float`
9. `<f-type> -> string`
10. `<f-type> -> void`
11. `<f-type> -> ϵ`
12. `<f-args> -> <f-arg>`
13. `<f-args> -> <f-arg>, <f-args>`
14. `<f-args> -> ϵ`
15. `<f-arg> -> <arg-type> ID`
16. `<arg-type> -> int`
17. `<arg-type> -> float`
18. `<arg-type> -> string`
19. `<arg-type> -> ϵ`
20. `<stat> -> ID = <expr> ;`
21. `<stat> -> ID = ID (<args>) ;`
22. `<stat> -> ID (<args>) ;`
23. `<stat> -> return <expr> ;`
24. `<stat> -> if (<expr>)`
25. `<stat> -> if (<expr>) <stat> else <stat>`
26. `<stat> -> while (<expr>) <stat>`
27. `<stat> -> { <stat-list> }`
28. `<args> -> <arg>`
29. `<args> -> <arg>, <args>`
30. `<arg> -> <term>`
31. `<arg> -> ϵ`
32. `<term> -> int`
33. `<term> -> float`
34. `<term> -> string`
35. `<term> -> NULL`
36. `<term> -> ID`
37. `<stat-list> -> <stat> <stat-list>`
38. `<stat-list> -> ϵ`
39. `<expr> -> EXPR <expr>`
40. `<expr> -> ID (<args>)`
41. `<expr> -> ϵ`
42. `<declare> -> declare(strict-types = 0) ;`
43. `<declare> -> declare(strict-types = 1) ;`

Tabulka 2: LL – gramatika řídící syntaktickou analýzu

C LL – tabulka

	<?php	<?	declare	function ID	ID	int	float	string	void	null	=	:	,	;	()	if	else	while	{	}	?>	return
<prog>	1	1																					
<declare>			2																				
<f-dec-stats>				4																			
<f-dec-stat>				5																			
<f-args>						6	6	6								7							
<f-arg>						8	8	8					9										
<f-type>						10	10	10	10														
<arg-type>						11	11	11															
<stat>					12												12	13	12	12			12
<stat-list>					14												14		14	14	15		14
<args>					16	16	16	16		16						17							
<arg>					18	18	18	18		18			19										
<term>					20	20	20	20		20													
<expr>					21									22		23							

Tabulka 3: LL – tabulka použitá při syntaktické analýze

D Precedenční tabulka

	+ -	* /	<>	()	id	\$
+ -	>	<	>	<	>	<	>
* /	>	>	>	<	>	<	>
<>	<	<	0	<	>	<	>
(<	<	<	<	=	<	0
)	>	>	>	0	>	0	>
id	>	>	>	0	>	0	>
\$	<	<	<	<	0	<	End

+ - – aritmetické operátory + a -

* / – aritmetické operátory * a /

<> – relační operátory ==, !=, <, >, <=, >=

(– levá závorka

) – pravá závorka

id – identifikátor

\$ – konec vstupu

Tabulka 4: Precedenční tabulka použitá při syntaktické analýze