



## Dokumentace k projektu překladače IFJ a IAL

6. prosince 2023

### Řešitelé

Jméno	Login	Rozdělení bodů
Ondřej Vala ( <i>vedoucí</i> )	xvalao01	25 %
Lukáš Prokeš	xproke14	25 %
Vít Slavíček	xslavi39	25 %
Petr Štanc1	xstanc12	25 %

**Identifikace varianty:** Tým xvalao01, varianta TRP-izp

### Rozšíření

FUNEXP  
OVERLOAD

# Obsah

<b>1</b>	<b>Úvod</b>	<b>3</b>
<b>2</b>	<b>Návrh</b>	<b>3</b>
2.1	Lexikální analýza . . . . .	3
2.2	Syntaktická analýza . . . . .	3
2.2.1	Rekurzivní sestup . . . . .	3
2.2.2	Precedenční syntaktická analýza . . . . .	4
2.3	Sémantická analýza . . . . .	4
2.4	Generování cílového kódu . . . . .	4
2.5	Datové struktury . . . . .	4
2.6	Členění implementačního řešení . . . . .	5
<b>3</b>	<b>Implementace</b>	<b>5</b>
3.1	Rozdělení práce . . . . .	6
<b>4</b>	<b>Přílohy</b>	<b>7</b>
4.1	Konečný automat . . . . .	7
4.2	Precedenční tabulka . . . . .	8
4.3	Redukční pravidla . . . . .	8
4.4	LL-gramatika . . . . .	9
4.5	LL-tabulka . . . . .	11

# Úvod

Tento dokument stručně popisuje návrh a proces implementace překladače jazyka, který je podmnožinou jazyka Swift, do jazyka IFJcode23.

## Návrh

Překladač je rozdělen do několika modulů, přičemž každý modul se specializuje na specifickou část překladu. Tato struktura umožňuje každému modulu zaměřit se na určitý aspekt překladu kódu, zatímco celkově spolupracují tak, aby pokryly všechny fáze překladu s maximální efektivitou a spolehlivostí.

## Lexikální analýza

Lexikální analýzu provádí **scanner** v průběhu překladu kódu, tak, jak si jiné části překladače žádají další tokeny (struktury). Celý **scanner** je založen na konečném automatu, který vidíte na obrázku 2, který jsme navrhli na začátku vývoje. V průběhu implementace **scanneru** jsme však zjistili, že náš návrh automatu není vždy úplně vhodný pro lexikální analýzu překládaného jazyka. **Scanner** tak provádí dodatečné operace, které nejsou obsaženy v automatu.

Jedním z problémů, kterému jsme v průběhu implementace čelili, bylo zpracování blokových komentářů. V našem návrhu automatu jsme totiž nepočítali s možností nekonečně vnořených blokových komentářů. Protože pomocí běžných konečných automatů nelze navrhnout něco, jako jsou nekonečně vnořené blokové komentáře, museli bychom například použít zásobníkový konečný automat. Abychom to vyřešili bez velkých zásahů do automatu, implementovali jsme mechanismus pomocí počítače, který sleduje počet začátečních a ukončovacích sekvencí znaků blokových komentářů.

Dalším aspektem, který jsme implementovali odlišně než pouze pomocí automatu, je rozlišení mezi identifikátory a klíčovými slovy. Konečný automat by mohl zaznamenávat rozpoznávání jednotlivých klíčových slov, ale to považujeme to za nevhodné řešení kvůli velkému počtu nových stavů a nepřehledným přechodům mezi těmi to stavy. Další nevýhodou by bylo, že při přidávání dalšího klíčového slova by bylo nutné upravit část automatu a s ním i část **scanneru**. Proto jsme se rozhodli, že rozlišování mezi identifikátory a klíčovými slovy bude prováděno pomocí seznamu klíčových slov. Toto řešení je sice trochu pomalejší, ale výrazně přehlednější a jednodušší na implementaci.

Ostatní části lexikální analýzy jsou implementovány v souladu s původním návrhem automatu. Pro každý stav automatu existuje odpovídající stav ve **scanneru**, a přechody jsou řešeny pomocí konstrukce switch-case. Pokud **scanner** narazí na sekvenci znaků, která není definována v konečném automatu, vrací lexikální chybu.

## Syntaktická analýza

Syntaktickou analýzu celého překládaného kódu zajišťuje **parser**. Výjimkou jsou výrazy, u kterých se o precedenční syntaktickou analýzu stará **expression parser**.

## Rekurzivní sestup

Při spuštění programu je aktivován **parser**, který postupně zpracovává tokeny poskytované scannerem a provádí syntaktickou analýzu. Implementace parseru je založena na jednoprůchodovém rekurzivním sestupu a pracuje s LL1 gramatikou. Jedinou výjimkou je čtení argumentů funkce, kde bylo obtížné jednoznačně určit, zda se jedná o argument či o jméno argumentu. **Parser** pracuje dokud nenačte operátor přiřazení, nebo nenarazí na jiné místo, kde je potřeba vyřešit výraz (argument funkce, návratová hodnota z funkce). V tom případě volá funkci z modulu **expression parser** a přes strukturu **programState** ji předá už načtené tokeny.

Po vyřešení výrazu `expression parser` vrátí přes `programState` adresu výsledku, typ výsledku a další parametry. Následně `parser` provede sémantické kontroly jestli má výsledek správný typ. V případě, že `parser` narazí na sekvenci tokenů, pro kterou nemá pravidlo z gramatiky vrátí syntaktickou chybu a ukončí překlad.

## Precedenční syntaktická analýza

`Expression parser` žádá od `scanneru` tokeny tak dlouho, dokud nenarazí na token, který není schopen zpracovat, nebo dokud nenarazí na specifickou sekvenci tokenů označující konec výrazu. Načtené tokeny jsou ukládány `expression parserem` do fronty. Kromě toho `expression parser` využívá zásobník pro průběžné zpracování výrazu.

Průběh analýzy začíná vyhodnocením precedence tokenu na vrcholu zásobníku a tokenu na začátku fronty, a to podle precedenční tabulky 1. Pokud je na vrcholu fronty token s vyšší prioritou, je přesunut na vrchol zásobníku, a program pokračuje ve zpracování dalších tokenů. Pokud je na vrcholu zásobníku token s nižší prioritou, použije se vhodné redukční pravidlo 4.3, které buď sníží počet tokenů na zásobníku, nebo je transformuje na jiný typ. Tento postup se opakuje, dokud na vrcholu zásobníku není pouze token s výsledkem a vstupní fronta je prázdná. Výsledný token je pak vrácen `parseru`, který jej dále zpracuje. Vyjimku tvoří funkce ve výrazech. V tomto případě `expression parser` volá funkci z `parseru`, která se stará o zpracování funkcí.

Syntaktická chyba nastává v případě, kdy se snažíme najít precedenci dvou tokenů, které nesmějí následovat za sebou, nebo pokud není nalezeno vhodné redukční pravidlo.

## Sémantická analýza

Sémantická analýza probíhá souběžně se syntaktickou analýzou během generování kódu. O kontrolu sémantiky se starají tři hlavní komponenty: `parser`, `expression parser` a `symtable` (tabulka symbolů). `Parser` zajišťuje kontrolu typů při použití operátorů přiřazení. `Expression parser` sleduje správné použití typů operandů ve výrazech. `Symtable` pak provádí sémantické kontroly související s funkcemi. Tato tabulka uchovává všechna volání funkcí v seznamu a na konci překladu provede sémantickou kontrolu všech těchto volání funkcí.

## Generování cílového kódu

O generování cílového kódu se stará `symtable`. Generovaný kód je během překladu ukládán do 4 seznamů, které obsahují:

1. Definice a těla funkcí
2. Hlavní funkce
3. Reference pro komunikaci mezi `parserem` a `expression parserem`
4. Temporary list - deklarace proměnných před smyčkami

Na závěr běhu programu projde `generátor` všechny seznamy a sestaví výsledný kód. Každá položka v seznamu představuje jeden řádek kódu, a je klíčové, aby seznamy následovaly ve správném pořadí, aby vznikl validní a funkční kód.

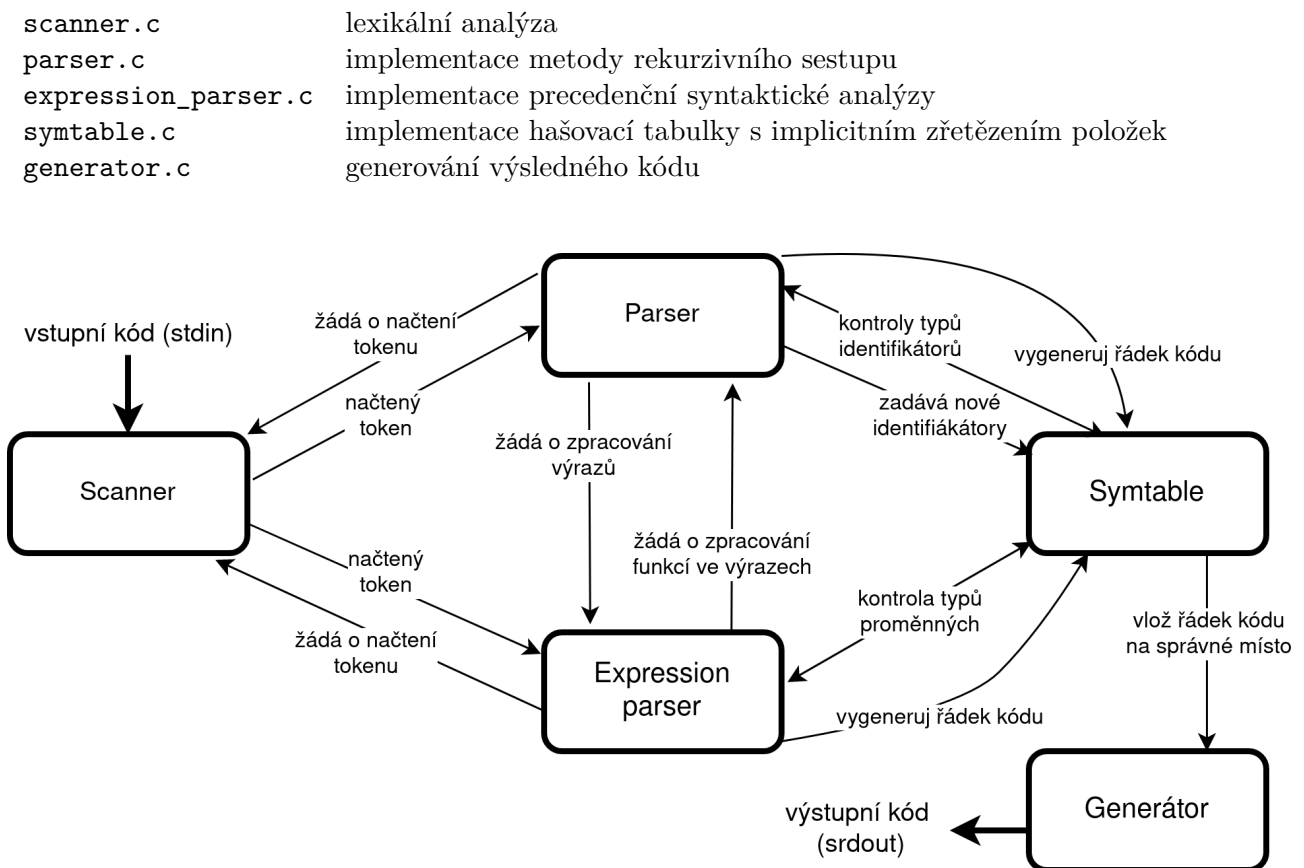
## Datové struktury

Při implementaci jsme využili několik klíčových datových struktur. Zde popíšeme nejdůležitější z nich:

- **struct token:** Tato struktura reprezentuje jednotlivé tokeny a slouží k komunikaci mezi `scannerem` a `parserem`. Také se pomocí tokenů provádí precedenční syntaktická analýza.

- **struct programState:** Tato struktura slouží k propojení informací mezi různými funkcemi programu. Výhodou této struktury je, že sdružuje všechny potřebné informace do jednoho objektu. Pokud bylo potřeba při implementaci přidat další informaci, stačilo přidat novou položku do této struktury, a nemusely se upravovat všechny funkce, které tuto strukturu využívají. Díky této struktruře by bylo snadné doimplementovat další možná rozšíření.
- **struct symtable:** Tato struktura představuje tabulku symbolů a je implementována jako seznam hašovacích tabulek. Každý prvek seznamu reprezentuje jedno zanoření. Pro hašování používáme funkci `Murmur0AAT()`<sup>1</sup>.
- **struct generator:** Tato struktura slouží k průběžnému ukládání generovaného cílového kódu. Obsahuje čtyři seznamy, kde každý řádek představuje jeden řádek v cílovém kódu. Dále obsahuje hašovací tabulku pro ukládání adres funkcí ve vygenerovaném kódu, což je implementováno pro podporu přetěžování funkcí.

## Členění implementačního řešení



Obrázek 1: Diagram komunikace jednotlivých částí překladače

## Implementace

Vývoj překladače probíhal během celého semestru, a snažili jsme se udržovat konstantní tempo práce, což se nám podařilo alespoň částečně. Pro správu verzí a sledování vývoje jsme využívali verzovací

<sup>1</sup><https://github.com/aappleby/smhasher/blob/master/src/Hashes.cpp>

systém Git se sdíleným repositářem na GitHubu. Hlavním komunikačním kanálem byl Discord, kde jsme také pořádali online porady.

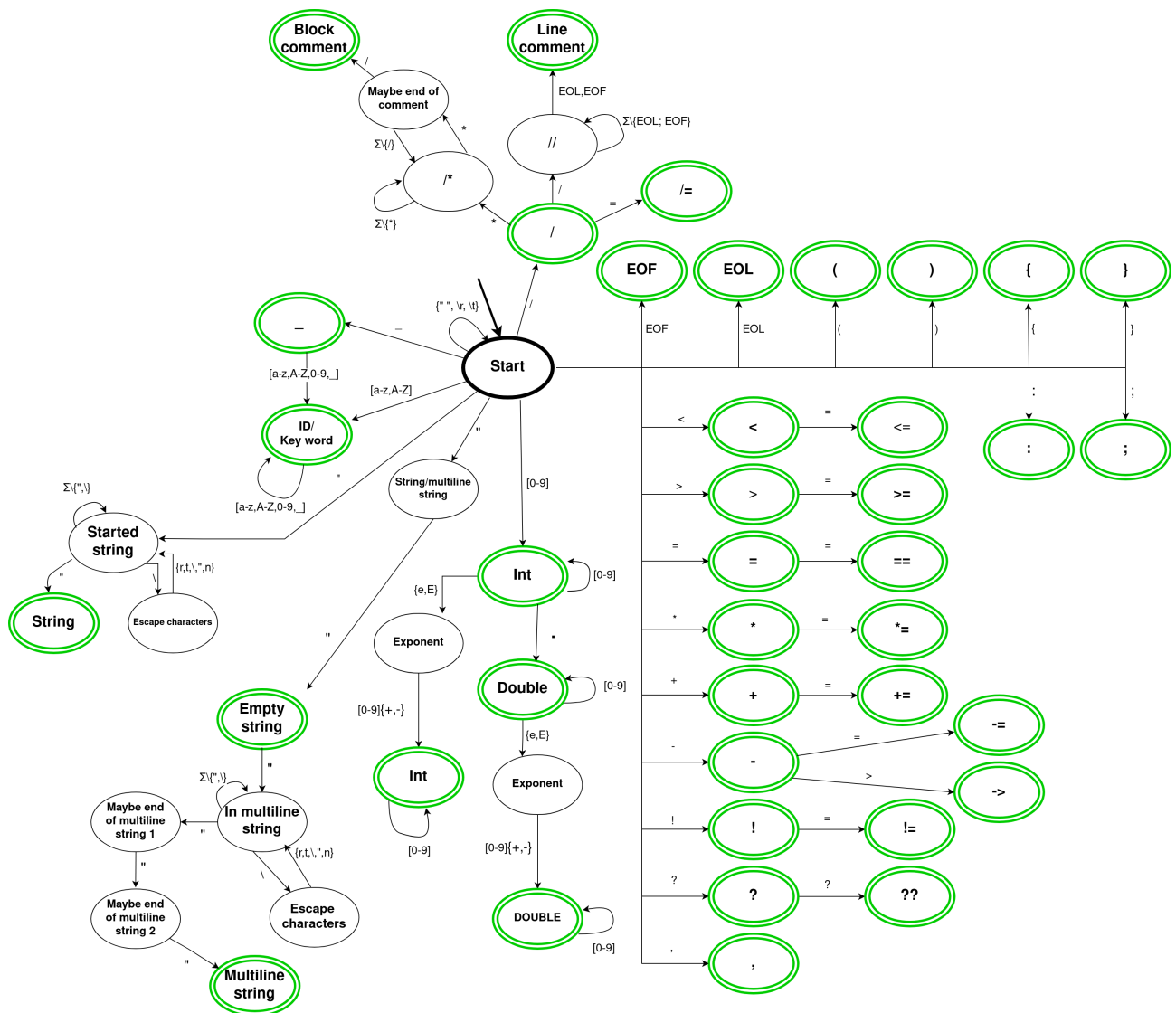
V počáteční fázi testování jsme využívali framework Google Test. Avšak v průběhu vývoje jsme zjistili, že pro naše účely je nevyhovující. Proto jsme vytvořili vlastní skript v jazyce bash, který spouštěl překladač a porovnával jeho výstupy s referenčními výsledky. Tímto způsobem jsme dosáhli efektivního a přizpůsobitelného testování na míru našim potřebám.

## Rozdělení práce

- **Ondřej Vala**
  - Vedoucí týmu
  - Návrh precedenční tabulky a precedenčních pravidel
  - Návrh lexikálního analyzátoru
  - Implementace expression parseru
- **Lukáš Prokeš**
  - Návrh lexikálního analyzátoru
  - Návrh LL gramatiky
  - Implementace parseru
  - Vedoucí testování
- **Vít Slaviček**
  - Implementace tabulky symbolů
  - Implementace generátoru
  - Generické datové struktury
- **Petr Štanc**
  - Implementace scanneru
  - Vytvoření pomocné knihovny na práci s řetězcí

## Přílohy

### Konečný automat



Obrázek 2: Konečný automat lexikálního analyzátoru

## Precedenční tabulka

	(	)	* /	+ -	< >	<= >=	== !=	??	!	i	\$
(	<	=	<	<	<	<	<	<	<	<	
)		>	>	>	>	>	>	>	>		>
* /	<	>	>	>	>	>	>	>	<	<	>
+ -	<	>	<	>	>	>	>	>	<	<	>
< <= > >=	<	>	<	<	>	>	>	>	<	<	>
== !=	<	>	<	<	>	>	>	>	<	<	>
??	<	>	<	<	>	>	<	<	<	<	>
!	>	>	>	>	>	>	>	>	>	>	>
i		>	>	>	>	>	>	>	>		>
\$	<		<	<	<	<	<	<	<	<	

Tabulka 1: Precedenční tabulka

## Redukční pravidla

1.  $E \rightarrow (E)$
2.  $E \rightarrow E * E$
3.  $E \rightarrow E / E$
4.  $E \rightarrow E + E$
5.  $E \rightarrow E - E$
6.  $E \rightarrow E > E$
7.  $E \rightarrow E >= E$
8.  $E \rightarrow E < E$
9.  $E \rightarrow E <= E$
10.  $E \rightarrow E == E$
11.  $E \rightarrow E != E$
12.  $E \rightarrow E ?? E$
13.  $E \rightarrow E !$
14.  $E \rightarrow i$

Redukční pravidla pro analýzu výrazu



## LL-gramatika

1. `<start> → <eol> <code>`
2. `<start> → EPS`
3. `<code> → <definition> EOL <code>`
4. `<code> → <statement> EOL <code>`
5. `<code> → EOF`
6. `<eol> → EOL`
7. `<eol> → EPS`
8. `<type> → Int`
9. `<type> → Double`
10. `<type> → String`
11. `<definition> → func <eol> ID <eol> (<functionParams>) <eol> <funcDefMid>`
12. `<funcDefMid> → {<statements>}`
13. `<funcDefMid> → -> <eol> <type> <eol> {<statements>}`
14. `<functionParams> → EPS`
15. `<functionParams> → EOL <functionParams>`
16. `<functionParams> → <functionParam> <functionParamsN> <eol>`
17. `<functionParamsN> → , <eol> <functionParam> <functionParamsN>`
18. `<functionParamsN> → EPS`
19. `<functionParam> → _<eol> ID <eol> : <eol> <type> <eol>`
20. `<functionParam> → ID <eol> ID <eol> : <eol> <type> <eol>`
22. `<statements> → <eol> <statementsBlock> <eol>`
23. `<statements> → EPS`
24. `<statementsBlock> → <statement> EOL <statementsBlock>`
25. `<statementsBlock> → EPS`
26. `<statement> → <varDec>`
27. `<statement> → if <eol> <letExp> <eol> {<statements>} <eol> else <eol> {<statements>}`
28. `<statement> → while <eol> <letExp> <eol> {<statements>}`
29. `<statement> → return <returnExpression>`
30. `<statement> → ID <callOrAssign>`
66. `<statement> → <parseBuildInFunctions>`
68. `<letExp> → let <eol> ID`
69. `<letExp> → <expression>`
36. `<callOrAssign> → <eol> <assign>`
37. `<callOrAssign> → (<arguments>)`
61. `<assign> → = <expression>`
62. `<assign> → += <expression>`
63. `<assign> → -= <expression>`

64. <assign> → \*= <expression>  
 65. <assign> → /= <expression>  
 38. <varDec> → let <eol> ID <eol> <varDecMid>  
 39. <varDec> → var <eol> ID <eol> <varDecMid>  
 40. <varDecMid> → : <eol> <type> <eol> <varDef>  
 41. <varDecMid> → = <expression>  
 42. <varDef> → EPS  
 43. <varDef> → = <expression>  
 44. <returnExpression> → <expression>  
 45. <returnExpression> → EPS  
 46. <arguments> → EPS  
 47. <arguments> → EOL <arguments>  
 48. <arguments> → <argument> <argumentsN> <eol>  
 49. <argumentsN> → , <eol> <argument> <argumentsN>  
 50. <argumentsN> → EPS  
 51. <argument> → <expression>  
 52. <argument> → ID <eol> <argWithName> <eol>  
 53. <argWithName> → : <eol> <expression>  
 54. <argWithName> → EPS  
 60. <expression> → SEND TO EXPRESSION PARSER  
 67. <expression> → KW\_READSTRING  
 70. <expression> → KW\_READINT  
 71. <expression> → KW\_READDOUBLE  
 72. <expression> → KW\_INT\_TO\_DOUBLE  
 73. <expression> → KW\_DOUBLE\_TO\_INT  
 74. <expression> → KW\_LENGTH  
 75. <expression> → KW\_SUBSTRING  
 76. <expression> → KW\_ORD  
 77. <expression> → KW\_CHR

LL-tabulka

	ID	IntData	Double Data	String Data	tunc	return	if	else	while	let	var	int	Double	String	(	)	{	}	=	+=	-=	*=	/=	->	:	-	,	EOL	EOF	\$	<exp>
<start>	1				1	1	1		1	1	1																	1	1	2	
<code>	4				3	4	4		4	4	4																				
<col>	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7				7	7	7	7	7	6	7		
<type>					11							8	9	10																	
<definition>																	12						13								
<functionParams>	16															14			36	36	36	36				16					
<functionParamsN>																18			61	62	63	64	65				17	18			
<functionParam>	20																	23								19					
<statements>	22					22	22		22	22	22							25										22	25		
<statementsBlock>	24					24	24		24	24	24																				
<statement>	30					29	27		28	26	26																				
<letExp>										68																					69
<callOrAssign>															37				36	36	36	36	36						36		
<assign>										38	39								61												
<varDec>																		41							40						
<varDecMid>																		42	43										42		
<returnExpression>																		45											45		
<arguments>	48	48	48	48												46													47		
<argumentsN>																50													49	50	
<argument>	52	51	51	51																											
<argWithNname>																54									53		54	54			
<expression>																															

Obrázek 3: LL-tabulka