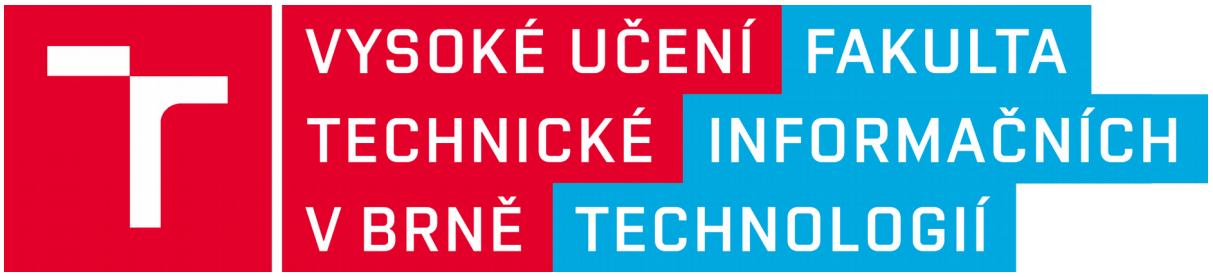


IFJ Projekt 2017

Dokumentace

IFJ17 (Podmnožina Basic) – tým 37, varianta I.

Vedoucí týmu:	Kristyna Jandová	**- xjando04**
Členové: týmu	Vilém Faigel	**- xfaige00**
	Nikola Timkova	**- xtimko01**
	Bc. Václav Doležal	**- xdolez76**

Obsah

1 Úvod

Zadáním bylo vytvořit program v jazyce C, který načte zdrojový kód zapsaný ve zdrojovém jazyce IFJ17 a přeloží jej do cílového jazyka IFJcode17 (mezikód). Jazyk IFJ17 je zjednodušenou podmnožinou jakyka FreeBasic, jenž staví na legendárním jazyce BASIC.

Jedná se o staticky typovaný, imperativní jazyk.

1.1 Varianta

Zvolili jsme variantu I, což znamená, že tabulka symbolů je implementována pomocí binárního vyhledávacího stromu.

2 Teorie řešení

Řešení jednotlivých modulů překladače.

2.1 Lexikální analýza

Lexikální analyzator – neboli Scanner je první fází překládacího procesu. Stará se o rozdělení vstupního kódu na jednotlivé časti – takzvané tokeny.

Proces tokenizace probíhá za pomocí metody ***_scanner_next(string *word)***, ve které je naprogramován konečný automat lexikální analýzy (Viz obrázek 1.). Tato metoda je volána v metodě ***scanner_next_token()***, která se stará o výsledné složení vraceného tokenu.

Token obsahuje **flag** - o jaký token se jedná (reprezentováno číslem definovaným v tokens), **ID** - hodnotu slova, **line a position** – definuje pozici tokenu

Tokenem **není:**

- Bílý znak
  - mezera
  - tabulátor
  - vícenásobné odřádkování
- Komentář
  - jednořádkový
  - víceřádkový

Narazí-li scanner na chybný token vrací LEXICAL_ERROR – tedy číslo 1 a ukončí program

2.2 Syntaktická analýza

Syntaktický analyzátor – neboli Parser je stěžejním modulem celého překladače. Náš parser je založen na metodě automatu řízeného stackem. (Viz příloha 3) Postupně volá ze Scanneru tokeny za pomocí metody **scanner_next_token()**. Kontroluje jejich souslednost pomocí vhodně navržené gramatiky. Tuto gramatiku naleznete popsanou v příloze 2.

Prací parseru je i kontrola sémantiky, a to v úrovni jestli nedochází ke střetu datových typů. Další sémantická kontrola se provádí až na úrovni interpretu.

Stane-li se že parser narazí v kódu na výraz, volá precedenční analýzu, která tento výraz vyhodnotí.

Další funkcí parseru je ta, že při kontrole syntaxe ukládá nalezené tokeny do tabulky symbolů.

V případě že parser narazí na chybu vrací SYNTAX_ERROR – tedy číslo 2 a ukončí program

2.3 Precedenční syntaktická analýza

Součástí parseru je precedenční syntaktický analyzátor, který jsme pro přehlednost oddělili do samostatného souboru. Tento analyzátor se stará o správnou syntaxi výrazů.

Kontrola probíhá za pomoci precedenční syntaktické tabulky (Viz příloha 4)

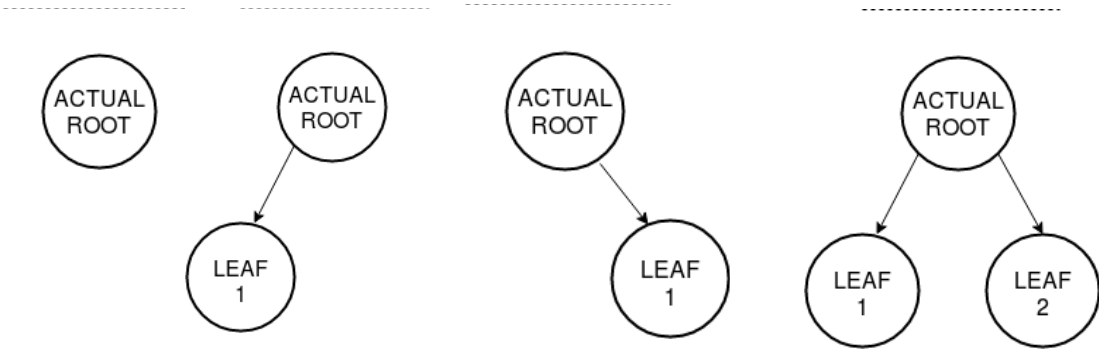
3. Algoritmy

popis použitých algoritmů při zpracování dat

3.1 Binární vyhledávací strom

Binární vyhledávací strom je typ stromu který má z uzlu právě 0 až 2 potomky.

Přípustné binární (pod)stromy:



Binární strom se může dále rozvíjet a to tak, že daný list může být i uzlem pro další listy.

Pro rovnoměrné rozložení binárního stromu je implementována funkce ***void tree_balance(struct tree *t);***

4 Práce v týmu

Popis spolupráce a rozdělení vývoje

4.1 Nástrahy a útrapy týmového projektu

V prvopočátku projektu byl náš tým velmi entuziastický – pravidelná komunikace, navržení milníků tak, aby bylo vše stihnuto s předstihem a zbyl čas na rozšíření, chuť všech členů do práce.

Bohužel s přibývajícím časem se množil neochota, chyběla pravidelná komunikace a čas mezi dotazem a odpovědí na stav práce se prodlužovaly

V poslední fázi vývoje jsme se museli uchýlit k drastické metodě vynucení spolupráce – k výhružce přidělení 0% za projekt, proto je procentuální rozdělení nevyvážené.

4.2 Rozdělení práce

4.2.1 Rozdělení vývoje

Kristyna Jandová	Scanner, dokumentace, příprava obhajoby
Vilém Faigel	Parser, instrukce, podpůrné systémy
Nikola Timkova	Precedenční analýza
Václav Doležal	IAL část, generování IFJcode17

4.2.1 Procentuální rozdělení

Kristyna Jandová	30%
Vilém Faigel	40%
Nikola Timkova	10%
Václav Doležal	20%

5 Závěr

Projekt se podařilo dokončit a otestovat na různých druzích vlastních testovacích příkladů, a to i na serveru Merlin.

Projekt pro nás byl velkým zdrojem zkušeností, které zajisté využijeme i v následujících projektech či v praxi.

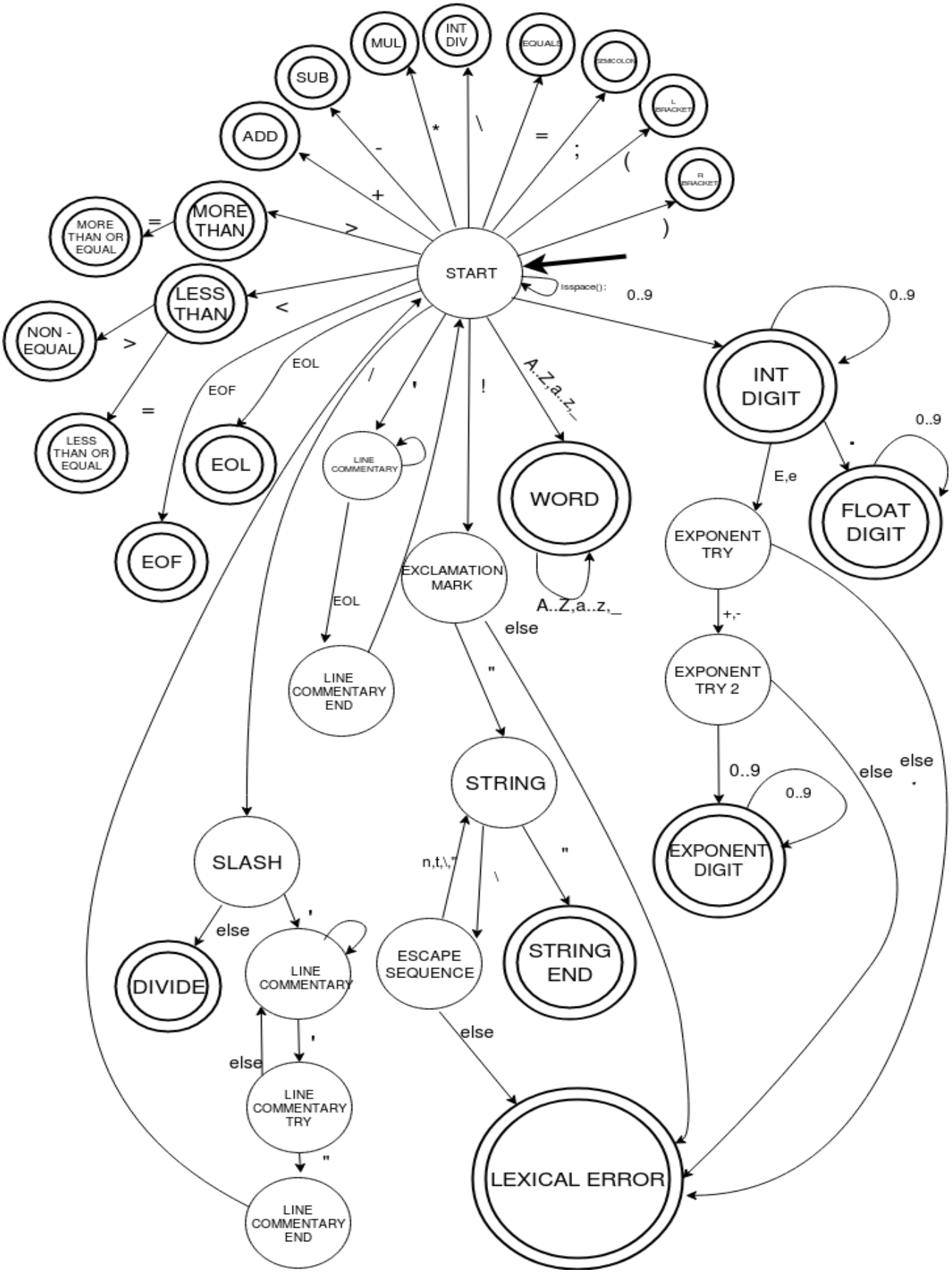
6 Metrika

Počet řádků kódu:

Language	files	blank	comment	code
C	16	872	800	2468
C/C++ Header	17	185	339	528
make	1	5	0	14
SUM:	34	1062	1139	3010

Velikost spustitelného souboru: 83 kiB

Příloha 1


START
MUL
INT DIV
SUB
ADD
EQUALS
SEMICOLON
L BRACKET
R BRACKET
MORE THAN
MORE THAN OR EQUAL
LESS THAN
LESS THAN OR EQUAL
NON - EQUAL
EOF
EOL
LINE COMMENTARY
LINE COMMENTARY END
EXCLAMATION MARK
WORD
A..Z,a..z,_
INT DIGIT
FLOAT DIGIT
EXPONENT TRY
EXPONENT TRY 2
EXPONENT DIGIT
STRING
ESCAPE SEQUENCE
STRING END
SLASH
DIVIDE
LINE COMMENTARY TRY
LEXICAL ERROR
isspace():
0..9
E,e
+,-
else
n,t,\,"

Příloha 2

1. <PROGRAM> -> <FUCTION_LIST>SCOPE EOL<STATEMENT_LIST> END SCOPE <FUNCTION_LIST>

2. <FUNCTION_LIST> -> ε
3. <FUNCTION_LIST> -> <FUNCTION> <FUNCTION_LIST>

4. <FUNCTION> -> DECLARE FUNCTION (id) lb <PARAM> rb AS <TYPE> EOL

5. <FUNCTION> -> FUNCTION (id) lb <PARAM> AS <TYPE> EOL <STATEMENT_LIST> END FUNCTION

6. <PARAM> -> ε
7. <PARAM> -> (id) AS <TYPE> <PARAM>
8. <PARAM> -> , (id) AS <TYPE> <PARAM>

9. <STATEMENT_LIST> -> ε
10. <STATEMENT_LIST> -> <STATEMENT> <STATEMENT_LIST>

11. <STATEMENT> -> DIM (id) AS <TYPE> EOL
12. <STATEMENT> -> DIM (id) AS <TYPE> <EXPRESSION> EOL

13. <STATEMENT> -> RETURN <EXPRESSION> EOL
14. <STATEMENT> -> (id) = <EXPRESSION> EOL
15. <STATEMENT> -> INPUT (id) EOL
16. <STATEMENT> -> PRINT <TERM> EOL
17. <STATEMENT> -> (id) = (func_id) lb <PARAM> rb EOL

18. <STATEMENT> -> IF <EXPRESSION> THEN EOL <STATEMENT_LIST> <ELSEIF> <ELSE> END IF
19. <STATEMENT> -> DO WHILE <EXPRESSION> EOL <STATEMENT_LIST> LOOP

20. <ELSEIF> -> ε

21. <ELSEIF> -> ELSE IF <EXPRESSION> THEN EOL <STATEMENT_LIST> <ELSEIF>

22. <ELSE> -> ε

23. <ELSE> -> ELSE EOL <STATEMENT LIST> END IF

24. <TERM> -> ;

25. <TERM> -> (id) <TERM>

26. <TERM> -> ;(id) <TERM>

27. <TERM> -> <STRING> <TERM>

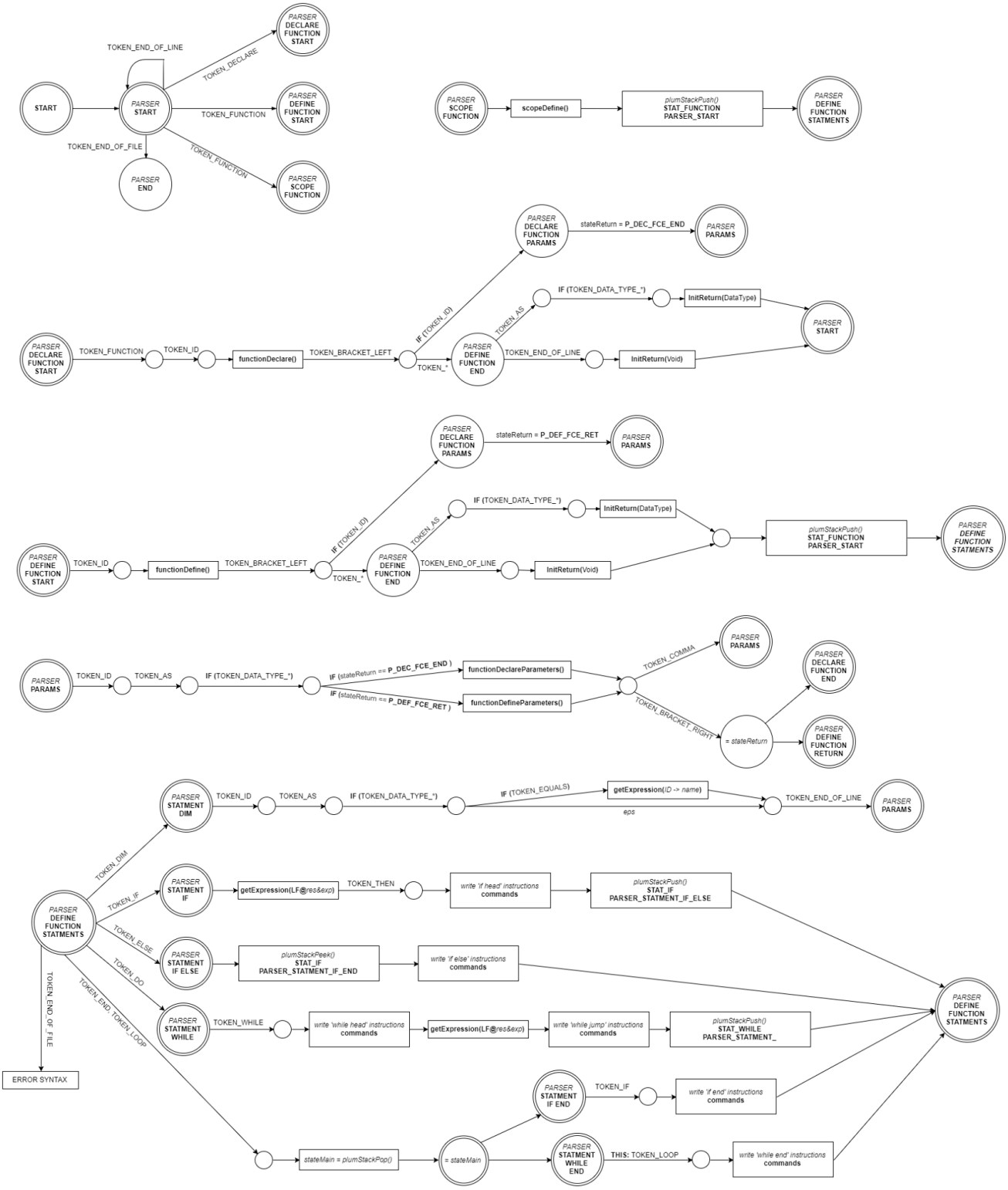
28. <TERM> -> ; <STRING> <TERM>

29. <TYPE> -> INTEGER

30. <TYPE> -> DOUBLE

31. <TYPE> -> STRING

Příloha 3


START
PARSER START
TOKEN_END_OF_LINE
TOKEN_DECLARE
TOKEN_FUNCTION
TOKEN_END_OF_FILE
TOKEN_FUNCTION
PARSER DECLARE FUNCTION START
PARSER DEFINE FUNCTION START
PARSER END
PARSER SCOPE FUNCTION
PARSER SCOPE FUNCTION
scopeDefine()
plumStackPush() STAT_FUNCTION PARSER_START
PARSER DEFINE FUNCTION STATMENTS
PARSER DECLARE FUNCTION START
TOKEN_FUNCTION
TOKEN_ID
functionDeclare()
TOKEN_BRACKET_LEFT
IF (TOKEN_ID)
PARSER DECLARE FUNCTION PARAMS
stateReturn = P_DEC_FCE_END
PARSER PARAMS
TOKEN_*
PARSER DEFINE FUNCTION END
TOKEN_AS
IF (TOKEN_DATA_TYPE_*)
InitReturn(DataType)
TOKEN_END_OF_LINE
InitReturn(Void)
PARSER START
PARSER DEFINE FUNCTION START
TOKEN_ID
functionDefine()
TOKEN_BRACKET_LEFT
IF (TOKEN_ID)
PARSER DECLARE FUNCTION PARAMS
stateReturn = P_DEF_FCE_RET
PARSER PARAMS
TOKEN_*
PARSER DEFINE FUNCTION END
TOKEN_AS
IF (TOKEN_DATA_TYPE_*)
InitReturn(DataType)
TOKEN_END_OF_LINE
InitReturn(Void)
plumStackPush() STAT_FUNCTION PARSER_START
PARSER DEFINE FUNCTION STATMENTS
PARSER PARAMS
TOKEN_ID
TOKEN_AS
IF (TOKEN_DATA_TYPE_*)
IF (stateReturn == P_DEC_FCE_END)
functionDeclareParameters()
IF (stateReturn == P_DEF_FCE_RET)
functionDefineParameters()
TOKEN_COMMA
TOKEN_BRACKET_RIGHT
PARSER PARAMS
PARSER DECLARE FUNCTION END
= stateReturn
PARSER DEFINE FUNCTION RETURN
PARSER DEFINE FUNCTION STATMENTS
TOKEN_DIM
TOKEN_IF
TOKEN_ELSE
TOKEN_DO
TOKEN_END, TOKEN_LOOP
TOKEN_END_OF_FILE
ERROR SYNTAX
PARSER STATMENT DIM
TOKEN_ID
TOKEN_AS
IF (TOKEN_DATA_TYPE_*)
IF (TOKEN_EQUALS)
getExpression(ID -> name)
eps
TOKEN_END_OF_LINE
PARSER PARAMS
PARSER STATMENT IF
getExpression(LF@res&exp)
TOKEN_THEN
write 'if head' instructions commands
plumStackPush() STAT_IF PARSER_STATMENT_IF_ELSE
PARSER STATMENT IF ELSE
plumStackPeek() STAT_IF PARSER_STATMENT_IF_END
write 'if else' instructions commands
PARSER STATMENT WHILE
TOKEN_WHILE
write 'while head' instructions commands
getExpression(LF@res&exp)
write 'while jump' instructions commands
plumStackPush() STAT_WHILE PARSER_STATMENT_
PARSER STATMENT IF END
TOKEN_IF
write 'if end' instructions commands
stateMain = plumStackPop()
= stateMain
PARSER STATMENT WHILE END
THIS: TOKEN_LOOP
write 'while end' instructions commands

Příloha 4

	+	-	*	/	\	>	<	>=	<=	=	<>	(	)	I	$
+	'>'	'>'	'<'	'<'	'<'	'>'	'>'	'>'	'>'	'>'	'>'	'<'	'>'	'<'	'>'
-	'>'	'>'	'<'	'<'	'<'	'>'	'>'	'>'	'>'	'>'	'>'	'<'	'>'	'<'	'>'
*****	'>'	'>'	'>'	'>'	'>'	'>'	'>'	'>'	'>'	'>'	'>'	'<'	'>'	'<'	'>'
/	'>'	'>'	'>'	'>'	'>'	'>'	'>'	'>'	'>'	'>'	'>'	'<'	'>'	'<'	'>'
****	'>'	'>'	'>'	'>'	'>'	'>'	'>'	'>'	'>'	'>'	'>'	'<'	'>'	'<'	'>'
>	'<'	'<'	'<'	'<'	'<'	' '	' '	' '	' '	' '	' '	'<'	'>'	'<'	'>'
<	'<'	'<'	'<'	'<'	'<'	' '	' '	' '	' '	' '	' '	'<'	'>'	'<'	'>'
>=	'<'	'<'	'<'	'<'	'<'	' '	' '	' '	' '	' '	' '	'<'	'>'	'<'	'>'
<=	'<'	'<'	'<'	'<'	'<'	' '	' '	' '	' '	' '	' '	'<'	'>'	'<'	'>'
=	'<'	'<'	'<'	'<'	'<'	' '	' '	' '	' '	' '	' '	'<'	'>'	'<'	'>'
<>	'<'	'<'	'<'	'<'	'<'	' '	' '	' '	' '	' '	' '	'<'	'>'	'<'	'>'
(	'<'	'<'	'<'	'<'	'<'	'<'	'<'	'<'	'<'	'<'	'<'	'<'	'='	'<'	' '
)	'>'	'>'	'>'	'>'	'>'	'>'	'>'	'>'	'>'	'>'	'>'	' '	'>'	' '	'>'
i	'>'	'>'	'>'	'>'	'>'	'>'	'>'	'>'	'>'	'>'	'>'	' '	'>'	' '	'>'
$	'<'	'<'	'<'	'<'	'<'	'<'	'<'	'<'	'<'	'<'	'<'	'<'	' '	'<'	'E'

Legenda:

‘>‘ - Redukce

‘<‘ - Handle

‘ ‘ - Nedefinováno

‘=‘ - Vyjímka

‘E‘ - Exit

Pravidla:

E -> E

E -> (E)

E-> E x E