

Computer Science and Engineering Indian Institute of Technology Kharagpur

Compiler Laboratory: CS39003

3rd year CSE, 5th semester

Assignment - 3: Lexer and Parser

Marks: 100

Assign Date: Aug 09, 2017

Submit Date: 23:55, Sep 05, 2017

1 Preamble - miniMatlab

This assignment follows the lexical and phrase structure grammar specification of a new language named **miniMatlab** which makes use of the **International Standard ISO/IEC 9899:1999 (E)** to support basic arithmetic and matrix operations. The lexical and phrase structure grammar specification quoted here is written using a precise yet compact notation typically used for writing language specifications. We first outline the notation and then present the Lexical and Phrase Structure Grammar that we shall work with.

2 Notation

In the syntax notation used here, syntactic categories (non-terminals) are indicated by italic type, and literal words and character set members (terminals) by bold type. A colon (:) following a non-terminal introduces its definition. Alternative definitions are listed on separate lines, except when prefaced by the words "one of". An optional symbol is indicated by the subscript "opt", so that the following indicates an optional expression enclosed in braces.

$$\{expression_{opt}\}$$

3 Lexical Grammar for miniMatlab

1. Lexical Elements:

token:

keyword

identifier

constant

string-literal

punctuator

2. Keywords:

keyword: one of

unsigned	break	return	void
case	float	short	char
for	signed	while	goto
Bool	continue	if	default
do	int	switch	double
long	else	Matrix	

3. Identifiers:

identifier:

identifier-nondigit

identifier identifier-nondigit

identifier digit

identifier-nondigit: one of

-	a	b	c	d	e	f	g	h	i	j	k	l	m
	n	o	p	q	r	s	t	u	v	w	x	y	z
	A	B	C	D	E	F	G	H	I	J	K	L	M
	N	O	P	Q	R	S	T	U	V	W	X	Y	Z

digit: one of

0 1 2 3 4 5 6 7 8 9

4. Constants:

constant:

integer-constant

floating-constant

character-constant

zero-constant

zero-constant:

0

integer-constant:

nonzero-digit

integer-constant digit

nonzero-digit: one of

1 2 3 4 5 6 7 8 9

floating-constant:

fractional-constant exponent-part_{opt}
digit-sequence exponent-part

fractional-constant:

digit-sequence_{opt} . digit-sequence
digit-sequence .

exponent-part:

e sign_{opt} digit-sequence
E sign_{opt} digit-sequence

sign: one of

+ -

digit-sequence:

digit
digit-sequence digit

character-constant:

'c-char-sequence'

c-char-sequence:

c-char
c-char-sequence c-char

c-char:

any member of the source character set except the single-quote ' , backslash \ , or
new-line character
escape-sequence

escape-sequence:

*\' \\" \? *
\a \b \f \n \r \t \v

5. String Literals:

String Literal:

"s - char - sequence_{opt}"

s-char-sequence:

s-char
s-char-sequence s-char

s-char:

any member of the source character set except the single-quote ' , backslash \ , or
new-line character
escape-sequence

6. Punctuators:

punctuator: one of

[] () . - >

++ -- & * + - ~ !
 / % << >> < > <= >= == != ^ | && ||
 ? : ;
 = *= /= %= += -= <<= >>= &= ^= |=
 , # '

7. Comments:

(a) Multi-line Comments

Except within a character constant, a string literal, or a comment, the characters `/*` introduce a comment. The contents of such a comment are examined only to identify multibyte characters and to find the characters `*/` that terminate it. Thus, `/* ... */` comments do not nest.

(b) Single-line Comments

Except within a character constant, a string literal, or a comment, the characters `//` introduce a comment that includes all multibyte characters up to, but not including, the next new-line character. The contents of such a comment are examined only to identify multibyte characters and to find the terminating new-line character.

4 Phrase Structure Grammar for miniMatlab

1. Expressions

primary-expression:

identifier

constant

string-literal

(expression)

postfix-expression:

primary-expression

postfix-expression [expression]

postfix-expression (argument-expression-list_{opt})

postfix-expression . identifier

postfix-expression - > identifier

postfix-expression ++

postfix-expression --

postfix-expression . '

argument-expression-list:

assignment-expression

argument-expression-list , assignment-expression

unary-expression:

postfix-expression

++ unary-expression

- unary-expression

unary-operator cast-expression

unary-operator: one of

*& * + -*

cast-expression:

unary-expression

multiplicative-expression:

cast-expression

*multiplicative-expression * cast-expression*

multiplicative-expression / cast-expression

multiplicative-expression % cast-expression

additive-expression:

multiplicative-expression

additive-expression + multiplicative-expression

additive-expression - multiplicative-expression

shift-expression:

additive-expression

shift-expression << additive-expression LEFTSH

shift-expression >> additive-expression

relational-expression:


shift-expression

relational-expression < *shift-expression*
relational-expression > *shift-expression*
relational-expression <= *shift-expression*
relational-expression >= *shift-expression*
equality-expression:
relational-expression ^{ET} *equality-expression* == *relational-expression*
equality-expression ! ^{NE} *relational-expression*
AND-expression:
equality-expression
AND-expression & *equality-expression*
exclusive-OR-expression:
AND-expression ^{POW} *exclusive-OR-expression* ^ *AND-expression*
inclusive-OR-expression:
exclusive-OR-expression ^{OR} *inclusive-OR-expression* | *exclusive-OR-expression*
logical-AND-expression:
inclusive-OR-expression ^{DAND} *logical-AND-expression* && *inclusive-OR-expression*
logical-OR-expression:
logical-AND-expression ^{DOR} *logical-OR-expression* || *logical-AND-expression*
conditional-expression:
logical-OR-expression ^{QM} *logical-OR-expression* ? *expression* : *conditional-expression* ^{COL}
assignment-expression:
conditional-expression
unary-expression *assignment-operator* *assignment-expression*
assignment-operator: one of
= * = / = % = + = - = <<= >>= & = ^ = | =
ASSIGN MULTA DIVA REMA PLUSA LSA RSA ANDA POWA ORA
expression:
assignment-expression
expression , *assignment-expression*

constant-expression:
conditional-expression

2. Declarations

declaration:
declaration-specifiers *init-declarator-list*_{opt} ;
declaration-specifiers:

*type-specifier declaration-specifiers*_{opt}
init-declarator-list:
 init-declarator
 init-declarator-list, init-declarator
init-declarator:
 declarator
 declarator = initializer

type-specifier:
 void
 char
 short
 int
 long
 float
 double
 Matrix
 signed
 unsigned
 Bool
declarator:
 *pointer*_{opt} *direct-declarator*
direct-declarator:
 identifier
 (*declarator*)
 direct-declarator [*assignment-expression*_{opt}]
 direct-declarator (*parameter-type-list*)
 direct-declarator (*identifier-list*_{opt})
pointer:
 * *pointer*_{opt}
parameter-type-list:
 parameter-list
parameter-list:
 parameter-declaration
 parameter-list , *parameter-declaration*
parameter-declaration:
 declaration-specifiers declarator
 declaration-specifiers
identifier-list:
 identifier
 identifier-list , *identifier*
initializer:
 assignment-expression
 LCB, RCB { *initializer-row-list* }
initializer-row-list:
 initializer-row

initializer-row-list ; initializer-row
initializer-row:
 designation_{opt} initializer
 initializer-row, designation_{opt} initializer
designation:
 designator-list = ASSIGN
designator-list:
 designator
 designator-list designator
designator:
 [*constant-expression*]
 . *identifier*

3. Statements

statement:
 labeled-statement
 compound-statement
 expression-statement
 selection-statement
 iteration-statement
 jump-statement
labeled-statement:
 identifier : statement
 case *constant-expression : statement*
 default : *statement*
compound-statement:
 { *block-item-list_{opt}* }
block-item-list:
 block-item
 block-item-list block-item
block-item:
 declaration
 statement
expression-statement:
 expression_{opt} ;
selection-statement:
 if (*expression*) *statement*
 if (*expression*) *statement* **else** *statement*
 switch (*expression*) *statement*
iteration-statement:
 while (*expression*) *statement*
 do *statement* **while** (*expression*) ;
 for (*expression_{opt} ; expression_{opt} ; expression_{opt}*) *statement*
 for (*declaration expression_{opt} ; expression_{opt}*) *statement*

jump-statement:
 goto *identifier* ;
 continue ;
 break ;
 return *expression*_{opt} ;

4. External definitions

translation-unit:
 external-declaration
 translation-unit external-declaration
external-declaration:
 function-definition
 declaration
function-definition:
 *declaration-specifiers declarator declaration-list*_{opt} *compound-statement*
declaration-list:
 declaration
 declaration-list declaration

5 The Assignment

1. Write a flex specification for the language of **miniMatlab** using the lexical grammar specified in Section 3. Name your .l file as `ass3_YourRollNo.l`. Use this specification to generate `lex.yy.c`. **Marks: 30**
2. Write a Bison specification for the language of **miniMatlab** using the phrase structure grammar specified in Section 4. Name your .y file as `ass3_YourRollNo.y`. Use this specification to generate `y.tab.c`. **Marks: 35**

While writing the Bison specification, you may need to make some changes to the grammar. For example, some non-terminals like `argument-expression-listopt` are shown as optional on the right-hand-side as:

postfix-expression:

`postfix-expression (argument-expression-listopt)`

One way to handle them (you may come up with your own method) would be to introduce a new non-terminal, `argument-expression-list-opt`, and a pair of new productions:

argument-expression-list-opt:

`argument-expression-list`

`ε`

and change the above rule as:

postfix-expression:

`postfix-expression (argument-expression-list-opt)`

3. Create a file named `ass3_YourRollNo_lexer.c` containing the `main()` function to test your lexer. This `main()` function should generate the output token stream when given a lexically correct code of **miniMatlab** as input and should return failure otherwise. Use the stdout for printing the token stream/failure message. **Marks: 5**

Hint: You may consider using `yylex()` here.

4. Create a file named `ass3_YourRollNo_parser.c` containing the `main()` function to test your parser. This `main()` function should generate the derivation rules when given a syntactically correct code of **miniMatlab** as input and should return failure otherwise. Use the stdout for printing the derivation rules/failure message. Please note that the .l and .y files should not contain the function `main()`. **Marks: 5**

Hint: You may consider using `yyparse()` here.

5. Prepare a Makefile for compiling the specifications and generating the lexer and the parser. First generate `lex.yy.c` from your .l file, `y.tab.c` using your .y file and then use these two along with the .c files to generate the binaries. **Marks: 5**

Hint: You may consider creating two different binaries, one from lex.yy.c, _lexer.c & y.tab.c and another from lex.yy.c, _parser.c & y.tab.c.

6. Test your lexer and parser using the test files (ass3_test_1.mm and ass3_test_2.mm) provided along with this assignment. Generate output token stream and derivation rules corresponding to these test files. Save these outputs in files named ass3_test_1_tokens_YourRollNo, ass3_test_1_dr_YourRollNo, ass3_test_2_tokens_YourRollNo, and ass3_test_2_dr_YourRollNo. Make sure these outputs are as per the flex and Bison specification of **miniMatlab**. **Marks: 20**

Marking scheme: *Testing lexer and parser using supplied test files (5+5) and using TA's test files (5+5).*

Prepare a zip file named ass3_YourRollNo.zip containing the following files: ass3_YourRollNo.1, ass3_YourRollNo.y, ass3_YourRollNo_lexer.c, ass3_YourRollNo_parser.c, Makefile, ass3_test_1_tokens_YourRollNo, ass3_test_1_dr_YourRollNo, ass3_test_2_tokens_YourRollNo, and ass3_test_2_dr_YourRollNo. Upload only ass3_YourRollNo.zip to Moodle.

Please ensure that the name of your files are strictly according to the supplied guidelines.