Computer Science and Engineering
Indian Institute of Technology Kharagpur

Compiler Laboratory: CS39003
3rd year CSE, 5th semester

miniMatlab Summary

Provided below are the actions that you have already done for Assignment 3 and Assignment 4.

1 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.

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

2 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
$+ \quad -$

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 double-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.

3 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*
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
equality-expression == *relational-expression*
equality-expression != *relational-expression*
AND-expression:
equality-expression
AND-expression & *equality-expression*
exclusive-OR-expression:
AND-expression
exclusive-OR-expression ^ *AND-expression*
inclusive-OR-expression:
exclusive-OR-expression
inclusive-OR-expression | *exclusive-OR-expression*
logical-AND-expression:
inclusive-OR-expression
logical-AND-expression && *inclusive-OR-expression*
logical-OR-expression:
logical-AND-expression
logical-OR-expression || *logical-AND-expression*
conditional-expression:
logical-OR-expression
logical-OR-expression ? *expression* : *conditional-expression*
assignment-expression:
conditional-expression

unary-expression assignment-operator assignment-expression

assignment-operator: one of

$= \quad *= \quad /= \quad \%= \quad += \quad -= \quad <<= \quad >>= \quad \&= \quad \wedge= \quad |=$

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

{ 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 =

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

4 Scope of Machine-Independent Translation

Focus on the following from the different phases to write actions for translation.

4.1 Expression Phase

Support all arithmetic, shift, relational, bit, logical (boolean), and assignment expressions excluding:

1. Comma (,) operator
2. Compound assignment operators:
   *= /= %= += -= <<= >>= &= ^= |=

4.2 Declarations Phase

Support for declarations should be provided as follows:

1. Simple variable, pointer, matrix and function declarations should be supported. For example, the following would be translated:

```
double d = 2.3;
int i;
Matrix mat[2][2] = {1.2, 2.0; 3.5, 4.3};
int a = 4, *p, b;
void func(int i, double d);
```

2. Consider only **void**, **char**, **int**, **double** and **matrix** type-specifiers. Here, char and int are to be taken as signed. For computation of offset and storage mapping of variables, assume the following sizes [1] (in bytes) of types:

Type	**Size**	**Remarks**
void	*undefined*	
char	1	
int	4	
double	8	
void*	4	*All pointers have same size*
Matrix	(no of elements * **size_of_double**) + (no of dimensions * **size_of_int**)	*All elements of matrix are stored as double*

It may also help to support an implicit **bool** (boolean) type with constants **1** (TRUE) and **0** (FALSE). This type may be inferred for a logical expression or for an **int** expression in logical context. Note that the users cannot define, load, or store variables of **bool** type explicitly, hence it is not storable and does not have a size.

3. Please consider **only 2 dimensional Matrices**.

4. Store the dimensions of the matrix as two integer constants (NumberRows × NumberColumns) in the first 8 bytes (4 bytes for each dimension) of

[1] Using hard-coded sizes for types does not keep the code machine-independent. Hence you may want to use constants like **size_of_char**, **size_of_int**, **size_of_double**, and **size_of_pointer** for sizes that can be defined at the time of machine-dependent targeting.

the space allocated for the matrix. Matrix elements will be stored from the 9th byte onwards.

5. Function declaration with only parameter type list may be skipped. Hence,

   void func(int i, double d);

   should be supported while

   void func(int, double);

   may not be.

4.3 Statement Phase

Support all statements excluding:

1. Declaration within **for**.
2. All Labelled statements *(labeled-statement)*.
3. **switch** in *selection-statement*.
4. All Jump statements *(jump-statement)* except **return**.

4.4 External Definitions Phase

Support function definitions and skip external declarations.

5 The 3-Address Code

Use the following 3-Address Code specification. Every 3-Address Code:

- Uses only up to 3 addresses.
- Is represented by a **quad** comprising - opcode, argument 1, argument 2, and result; where argument 2 is optional.

5.1 Address Types

- *Name:* Source program names appear as addresses in 3-Address Codes.
- *Constant:* Many different types and their (implicit) conversions are allowed as deemed addresses.
- *Compiler-Generated Temporary:* Create a distinct name each time a temporary is needed - might help in optimization.

5.2 Instruction Types

For Addresses x, y, z, and Label L

- *Binary Assignment Instruction:* For a binary op (including arithmetic, shift, relational, bit, and logical operators for **int**, **double** and **Matrix**):

  x = y op z

- *Unary Assignment Instruction:* For a unary operator op (including unary minus or plus, logical negation, bit, and conversion operators). Also support **Matrix** transpose (.'):

  x = op y

- *Copy Assignment Instruction:*

  x = y

- *Unconditional Jump:*

  goto L

- *Conditional Jump:*

  - *Value-based:*

    if x goto L

ifFalse x goto L

- *Comparison-based:* For a relational operator op (including $<, >, ==$ $, != , <=, >=$):

  if x relop y goto L

- *Procedure Call:* A procedure call p(x1, x2, ..., xN) having N $\geq$ 0 parameters is coded as (for addresses p, x1, x2, and xN):

  param x1
  param x2
  ...
  param xN
  y = call p, N

  Note that N is not redundant as procedure calls can be nested.

- *Return Value:* Returning a return value and / or assigning it is optional. If there is a return value v it is returned from the procedure p as:

  return v

- *Indexed Copy Instructions (for **Matrix**):*

  x = y[z]
  a[b] = c

- *Address and Pointer Assignment Instructions:*

  x = &y
  x = *y
  *x = y

6 Design of the Translator

6.1 Lexer and Parser

Use the Flex and Bison specifications[2] you had developed in Assignment 3 and write semantic actions for translating the subset of **miniMatlab** as specified in Section 4 and 5. Note that many grammar rules of your **miniMatlab** parser may not have any action or may just have propagate-only actions. Also, some of the lexical tokens may not be used.

6.2 Augmentation

Augment the grammar rules with markers and add new grammar rules as needed for the intended semantic actions. Justify your augmentation decisions within comments of the rules.

6.3 Attributes

Design the attributes for every grammar symbol (terminal as well as non-terminal). List the attributes against symbols (with brief justification) in comment on the top of your Bison specification file. Highlight the inherited attributes, if any.

6.4 Symbol Table

Use symbol tables for user-defined (including arrays and pointers) variables, temporary variables and functions.

Name	Type	Initial value	Size	Offset	Nested Table
...	...	...	...	...	...

For example, for

[2] You may correct your specification/s if you need.

```
double d = 2.3;
int i;
Matrix m[2][2] = {1.2, 2.0; 3.5, 4.3};
int a = 4, *p, b;
void func(int i, double d);
char c;
```

The Symbol Tables will look like:

Name	Type	Initial value	Size	Offset	Nested Table
d	**double**	2.3	8	0	null
i	**int**	null	4	8	null
m	***Matrix(2, 2, double)***	{1.2, 2.0; 3.5, 4.3}	32+8	12	null
a	**int**	4	4	52	null
p	***ptr(int)***	null	4	56	null
b	**int**	null	4	60	null
func	***function***	null	0	64	ptr-to-ST(func)
c	**char**	null	1	64	null

Table 1: **ST(global):** The Symbol Table for global symbols

Name	Type	Initial value	Size	Offset	Nested Table
i	**int**	null	4	0	null
d	**double**	null	8	4	null
retVal	**void**	null	0	12	null

Table 2: **ST(func):** The Symbol Table for function **func**

The Symbol Tables may support the following methods:

(a) **lookup(...)**
A method to lookup an id (given its name or lexeme) in the Symbol Table. If the id exists, the entry is returned, otherwise a new entry is created.

(b) **gentemp(...)**
A static method to generate a new temporary, insert it to the Symbol Table, and return a pointer to the entry.

(c) **update(...)**
A method to update different fields of an existing entry.

(d) **print(...)**
A method to print the Symbol Table in a suitable format.

Note:

- The fields and the methods are indicative. You may change their name, functionality and also add other fields and/or methods that you may need.
- It should be easy to extend the Symbol Table as further features are supported and more functionality is added.
- The global symbol table is unique.
- Every function will have a symbol table of its own parameters and automatic variables. This symbol table will be nested in the global symbol table.
- Symbol definitions within blocks are naturally carried in separate symbol tables. Each such table will be nested in the symbol table of the enclosing scope. This will give rise to an implicit stack of symbol tables (global one being the bottom-most) the while symbols are processed during translation. The search for a symbol starts from the top-most (current) table and goes down the stack up to the global table.

6.5 Quad Array

Quad array stores the 3-address **quad**'s. Index of a **quad** in the array is the *address* of the 3-address code. The quad array will have the following fields (having usual meanings).

op	**arg1**	**arg2**	**result**
...	...	...	...

Note:

- **arg 1** andor **arg 2** may be a variable (address) or a constant.
- **result** is variable (address) only.
- **arg 2** may be null.

For example, if

```
int i = 2;
double v = 3.0;
Matrix m[2][2] = {1.2, 2.0; 3.5, 4.3};
        ...
do i = i - 1; while (m[i][i] < v);
```

translates to

```
100: t1 = i - 1
101: i = t1
102: t2 = i * 4
103: t3 = m[4]            // Number of Columns
104: t4 = t2 - 4
105: t5 = t4 * t3
106: t6 = i * 4
107: t7 = t5 + t6
108: t8 = t7 + 8
109: t9 = m[t8]
110: if t9 < v goto 100
```

the quad's are represented as:

Index	op	arg 1	arg 2	result
...	...	...	...	...
100	-	i	1	t1
101	=	t1		i
102	*	i	4	t2
103	=[]	m	4	t3
104	-	t2	4	t4
105	*	t4	t3	t5
106	*	i	4	t6
107	+	t5	t6	t7
108	+	t7	8	t8
109	=[]	m	t8	t9
110	<	t9	v	100

The Quad Array may support the following methods:

(a) **emit(...)**
An overloaded static method to add a (newly generated) **quad** of the form: **result** = **arg1 op arg2** where **op** usually is a binary operator. If **arg2** is missing, **op** is unary. If **op** also is missing, this is a copy instruction.

(b) **print(...)**
A method to print the **quad** array in a suitable format.

For example the above state of the array may be printed (with the symbol information) as:

```
void main()
{
  int i = 2;
  double v = 3.0;
  Matrix m[2][2] = {1.2, 2.0; 3.5, 4.3};
  int t1;
  int t2;
  int t3;
  int t4;
  int t5;
  int t6;
  int t7;
  int t8;
  int t9;

  L100: t1 = i - 1
  L101: i = t1
  L102: t2 = i * 4
  L103: t3 = m[4]
  L104: t4 = t2 - 4
  L105: t5 = t4 * t3
  L106: t6 = i * 4
  L107: t7 = t5 + t6
  L108: t8 = t7 + 8
  L109: t9 = m[t8]
  L110: if t9 < v goto 100
}
```

Note:

- The fields and the methods are indicative. You may change their name, functionality and also add other fields and/or methods that you may need.

6.6 Global Functions

Following (or similar) global functions and more may be needed to implement the semantic actions:

(a) **makelist(i)**
A global function to create a new list containing only i, an index into the array of **quad**'s, and to return a pointer to the newly created list.

(b) **merge(p1, p2)**
A global function to concatenate two lists pointed to by **p1** and **p2** and to return a pointer to the concatenated list.

(c) **backpatch(p, i)**
A global function to insert **i** as the target label for each of the **quad**'s on the list pointed to by **p**.

(d) **typecheck(E1, E2)**
A global function to check if **E1** & **E2** have same types (that is, if <**type_of_E1**> = <**type_of_ E2**>). If not, then to check if they have compatible types (that is, one can be converted to the other), to use an appropriate conversion function
conv<type_of_E1>2<type_of_E2>(E) or
conv<type_of_E2>2<type_of_E1>(E) and to make the necessary changes in the Symbol Table entries. If not, that is, they are of incompatible types, to throw an exception during translation.

(e) **conv<type1>2<type2>(E)**
A global function to convert[3] an expression E from its current type type1 to target type type2, to adjust the attributes of E accordingly, and finally to generate additional codes, if needed.

Naturally, these are indicative and should be adopted as needed. For every function used clearly explain the input, the output, the algorithm, and the purpose with possible use at the top of the function.

[3] It is assumed that this function is called from **typecheck(E1, E2)** and hence the conversion is possible.