University of Toronto

CSC 488S Compilers and Interpreters **Winter 2008/2009**

Mid Term Test Solution

1. [20 marks] A proposal has been made to extend the CSC488S project language with *scope expressions* using the syntax:

scopeExpression: '**begin**' declarations statements '**return**' expression '**end**' **,**
'**begin**' statements '**return**' expression '**end**'

and adding

expression: scopeExpression **,**

A scope expression behaves like an implicit parameterless function. It can be used anywhere that an expression can be used. Example:

Z : = **begin var** Y : **int** Y = 0 **while**(X > 0) **do begin** X := X / 2 Y := Y + 1 **end return** Y **end**

a) Describe the semantic analysis issues raised by this proposal, i.e what checking is required.

b) Describe the symbol table issues raised by this proposal. How would they be resolved?

c) Describe any other issues raised by this proposal.

a) Need to capture type of block from type of final expression for type checking.

b) Need to be able to create/hide symbol table entries for the block scope in the middle of expression processing.

c) This feature does not introduce any syntax problems. An interesting question is: what is the meaning of a **return** statment if it occurs in *statements* rather than at the end? Is it another way to exit a scope expression or is it a return from an enclosing procedure/function?

2. [20 marks] Show how storage would be mapped for the array of structures (A) declared below.
Show the subscript calculation for A[23]. Show your work.

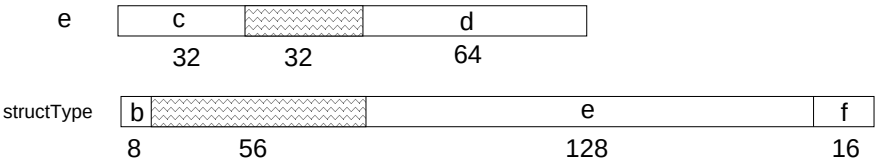
```
1   typedef struct {
2           char b ;
3           struct {
4                   int c ;
5                   double d ;
6           } e ;
7           short f;
8   } structType ;

9   structType A[ 97 ] ;
```

Sizes and Alignments

Type	Size	Alignment
char	8	8
short	16	16
int	32	32
double	64	64

Applying the structure mapping algorithm to structType results in this layout



The total size of structType is 208 bits (26 bytes).

However to make every element of the array A align properly, a fill of 48 bits (6 bytes) is required between elements of A. The size of an element of A is therefore 256 bits (32 bytes).

Address of A[23]: @ A[23] = Abase + (23 - 0) * 32 = Abase + 736

3. [10 marks] The Eiffel programming language is **case insensitive** for identifiers and reserved words. e.g.

test Test tEST TeSt TEST are all the same identifier

else ELSE Else eIsE eLsE are all the same reserved word

Describe the lexical and syntactic processing necessary to implement this feature.

During lexical analysis map all identifiers and reserved words to lower (or upper) case. A good compiler would record the original case in a bitmap (1 bit per character) for all identifiers (at least) so error messages could print identifers as the programmer wrote them. Syntax analysis would only need to deal with the lower case set of reserved words.

4. [25 marks] The grammar below describes a simplified version of declarations in C.

```
 1   declaration:        decl_specifiers  init_declarators  ';'
 2   decl_specifiers:    type_specifier  decl_specifier
 3   decl_specifier:     type_specifier  decl_specifier  ,
 4                       /* empty */
 5   type_specifier:     'int'  ,
 6                       'long'
 7   init_declarators:   init_declarator  init_declarators  ,
 8                       /* empty */
 9   init_declarator:    declarator  initializer
10   declarator:         identifier  ,
11                       declarator  '('  ')'  ,
12                       declarator  '['  ']'
13   initializer:        '='  expression  ,
14                       '='  '{'  initializer_list  '}'  ,
15                       /* empty */
16   initializer_list:   expression  ,
17                       initializer_list  ','  initializer_list  ,
18                       '{'  initializer_list  '}'
```

You may assume that *identifier* , *expression* and all of the symbols enclosed in single quote marks are terminal symbols.

`/* empty */` is a comment marking rules with empty right hand sides.

Convert this grammar to an LL(1) grammar. Show the director sets to prove the LL(1) property. You may assume any follow sets you need for other parts of C as long as you specify a sufficient condition (e.g. the follow set cannot contain X) for the grammar to be LL(1).

			Director Set
1	declaration:	decl_specifiers init_declarators ';'	{ 'int', 'long' }
2	decl_specifiers:	type_specifier decl_specifier	{ 'int', 'long' }
3	decl_specifier:	type_specifier decl_specifier ,	{ 'int', 'long' }
4		/* empty */	{ '=' , ';' }
5	type_specifier:	'int' ,	{ 'int' }
6		'long'	{ 'long' }
7	init_declarators:	init_declarator init_declarators ,	{ identifier }
8		/* empty */	{ ';' }
9	init_declarator:	declarator initializer	{ identifier }
10	declarator:	identifier identifierQualifier ,	{ identifier }
11	identifierQualifier	'(' ')' identifierQualifier ,	{ '(' }
12		'[' ']' identifierQualifier ,	{ '[' }
12a		/* empty */	{ '=' , ';' }
13	initializer:	'=' initializerBody ,	{ '=' }
15		/* empty */	{ ';' }
14	initializerBody:	expression	first(expression)
14a		'{' initializer_list '}' ,	{ '{' }
15	initializer_list:	oneInitializer moreInitializers	first(expression) ∪ { '{' }
16	oneInitializer:	expression ,	first(expression)
17		'{' initializer_list '}'	{ '{' }
18	moreInitializers:	',' oneInitializer moreInitializers	{ ',' }
18a		/* empty */	{ '}' }

Condition: first(expression) can not contain '{'

5. [25 marks] The function *mul* shown below is from class *Poly* that implements manipulation of polynomials in one variable. List the type and usage checks that a Java compiler would perform on this function.

```
        public class Poly  {
                private int [  ] trms ;
                private int deg ;
                . . .
1               public Poly mul (Poly q) throws NullPointerException {
2                       // returns the Poly this * q
3                       if (( q.deg == 0 ) && q.trms[ 0 ] == 0 ) | |
4                           ( deg == 0 ) && trms[ 0 ] == 0 ) return new Poly ;
5                       Poly r = new Poly( deg + q.deg ) ;
6                       r.trms[ deg+q.deg ] = 0 ;           // prepare to compute coeffs
7                       for ( int i = 0 ; i <= deg ; i ++ )
8                               for ( int j = 0 ; j <= q.deg ; j ++ )
9                                       r.trms[ i + j ] = r.trms[ i + j ] + trms[ i ] * q.trms[ j ] ;
10                      return r ;
11              }
```

Lines

1 Verify that nullPointerExecption is defined and is of a throwable exception type.

1 Check that *mul* with this signature has not been previously defined.

1,1,4,5,5 For each reference to *Poly*
Verify that *Poly* is defined and is a class.

3,5,6,8 For each reference to *q.deg*
Verify that *q* is defined and has accessible field *deg* Verify that *deg* is of an integer type

3,9 For each reference to *q.trms*
Verify that *q* is defined and has accessible field *trms* that is a one dimensional array
Verify that *trms[ ]* is of an integer type

6,9,9 For each reference to *r.trms*
Verify that *r* is defined and has accessible field *trms* that is a one dimensional array
Verify that *trms[]* is of an integer type

4,5,6,7 For each reference to *deg*
Verify that *deg* is a field of *this*
Verify that *deg* is of an integer type

4,9 For each reference to *trms*
Verify that *trms* is a field of *this*
Verify that *trms* is a one dimensional array
Verify that *trms[]* is of an integer type

3,3,4,4 For each instance of the == operator
Verify that its operands can be compared

3,4 For each instance of the && operator
Verify that both operands are of a Boolean type

Lines

4,10 For each instance of a **return** statement
Verify that the expression being returned is valid for the *mul* function

5,6,9,9,9 For each instance of the + operator
Verify that both operands are of an acceptable numeric type

7,9 For each instance of the ++ operator
Verify that both operands are of an acceptable numeric type

9 Verify that the operands of the * operator are of an acceptable numeric type

4,5 For each instance of the **new** operator
Verify that its operand is a class and that a suitable constructor exists.

5,6,9, For each instance of the = operator (assignment)
Verify that the assignment is valid

7,8 For each instance of the <= operator
verify that its operands can be compared.

3 Verify that the operands of the | | operator are of Boolean type

3,4,6,9,9,9,9 For each instance of the [ ] operator, verify that the subscript is valid.

7,7,9,9,9 For each reference to *i*
Verify that *i* is defined and accessible

8,8,9,9,9 For each reference to *j*
Verify that *j* is defined and accessible

5 Check that *r* has not been previously defined.