CS 302 : Implementation of Programming Languages
TUTORIAL 5 (Static Semantics); March 9, 2017

P1. Write program fragments in C / C++ to prove or disprove the following statements:

- there exist type expressions for which structural equivalence is used for checking type equivalence
- there exist type expressions for which name equivalence is used for checking type equivalence

P2. The algorithm for structural equivalence used for type checking is given below, where s and t are input type expressions.

```
function sequiv(s, t)  : boolean;
begin
        if s  and  t  are the same basic_type then return true
        else    if s   =  array (s1, s2) and   t =  array(t1, t2)
                then return sequiv(s1, t1) and sequiv (s2, t2)
        else    if s = s1 × s2 and  t = t1 × t2
                 then return sequiv (s1 ,t1) and sequiv (s2,  t2)
        else    if s = pointer (s1)  and t = pointer(t1)
                then return sequiv (s1 ,t1)
        else if s = s1 → s2  and t = t1 → t2
                 then  return sequiv (s1 ,t1) and sequiv (s2, t2)
                else return false
end
```

(a) Comment on whether this algorithm works for all type expressions in a language like C or C++. In case your answer is no, produce code fragments where it does not work with reasons.

(b) Change the algorithm to work for type checking for type expressions which have cycles.

P3. Write an algorithm for checking name equivalence of two type expressions, s and t.

```
function name_equiv(s, t)  : boolean;
begin

end
```

P4. The translation scheme given below performs conversion of infix expressions to postfix expressions using the terminals { addop, num}, where addop = { +, –} that works for both top down and bottom up parsing.

E → T R

R → *addop* T { print(addop.lexeme) } R_1 | ε

T → num { print(num.val) }

Write a translation scheme for converting postfix expressions to infix over the same alphabet. Report whether your translation scheme works for both the parsing methods.

P5. Refer to the intermediate code language described in the lecture. Write intermediate code manually that would be semantically equivalent to the following code fragments. If possible, generate at least 2 different intermediate code sequences for each fragment. You should not write a grammar of a SDD or a translation scheme for this question. Assume the following declarations for all the fragments. Numeric constants used in the programs are int or float as appropriate.

```
int a1, a2;
float x1 , x2;
double d1, d2;
bool b1, b2;
int f( int, float, bool);
```

1. a1 = x1 * a1 + d1 * b2 – x3;
2. b1 = a1+10 > 12 && d1 <= d2 or ! B2;
3. a2 = f(5*x1, d1+2.5, b1 || b2);
4. if (a1 + a2 != d1 + d2)
   { a1 = a1 + 10; a2 = a2 – 10;}
   else
   { d1 = a1 + 0.1; x1 = x2 – 3.14;}
5. while (a1 + x1 > a2 + x2 || b1)
   { a1 = d1 * x1; x2 = a2*d2; b1 = b1 and b2;}

P6. The template feature of C++ permits one to write template functions from which concrete functions and are generated by the compiler through instantiation at compile time. Mention the issues in the compilation of this feature and suggest in English a possible design strategy for its solution. Use the following example to illustrate how your solution handles the issues. Template function definition is given in column 1 and calls are given in column 2.

```
 template <class T>
T min ( T*array, int SIZE)
{  T res = array[0];
   for ( int i = 0; i < SIZE ; i++ )
     if ( res > array[i] )
           res = array[i];
   return res;
}
```

```
int  a[] = { 2, 5, 3, 4, 1, 5};
int  b[] = { 12, 15, 3, 4, 19, 5};
float  f[] = { 2.3 , 2.1, 2.7, 1.5, 0.3};
string  str[] = { "sb", "as", "dmd", "pb"};
cout << " minimum is " << min(a,6) << endl;
cout << " minimum is " << min(f,5) << endl;
cout << " minimum is " << min(str,4) << endl;
cout << " minimum is " << min(b,6) << endl;
```

P7. You are required to write Syntax Directed Translation Scheme (SDTS) for performing semantic analysis and intermediate code generation for assignment statements of the form given below. Variables are restricted to basic types, int and float and binary arithmetic operators {*, /, %, +, – } and unary minus. Assume that types of all variables are available and can be accessed through an appropriate function.

c = – a + b * c – d % b / g;

(a) State all the attributes of grammar symbols chosen by you and mention their intended use in semantic analysis and intermediate code generation

(b) Provide for detection, reporting and recovery from semantic errors to the extent possible. Type conversion rules for C++ is to be used.

(c) Generate the intermediate code produced for the input given above using your SDTS. You may adapt the sample grammar given below for your purpose.

P8. You are required to write Syntax Directed Translation Scheme (SDTS) for performing complete evaluation of a boolean expression. Your scheme should be able to handle input strings of the form given below. Variables are restricted to basic types, int and bool.

a + b > c + d and a*a >= 20 or not (a ==b or false)

(a) State all the attributes of grammar symbols chosen by you and mention their intended use in semantic analysis and intermediate code generation

(b) Provide for detection, reporting and recovery from semantic errors to the extent possible. The semantic actions for the rules for E are already addressed in P7 and hence may be omitted here.

(c) Generate the intermediate code produced for the input given above using your SDTS. You may adapt the sample grammar given below for your purpose.

$B \rightarrow B_1$ **or** B_2
$B \rightarrow B_1$ **and** B_2
$B \rightarrow$ **not** B_1
$B \rightarrow (B_1)$
$B \rightarrow$ **true**
$B \rightarrow$ **false**
$B \rightarrow E_1$ **relop** E_2
$B \rightarrow E$ { B.place := E.place}
$E \rightarrow E_1 + E_2$ | other rules as required

******* End of Tutorial Sheet 5 ********