CS 302 : Implementation of Programming Languages

TUTORIAL 7) : Control flow constructs and Array references March 23, 2017

P1. Consider the two SDTS given for evaluation of a boolean expression using complete evaluation and partial evaluation (given in the Appendix). For the input string,

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

(a) generate the intermediate code produced using both the SDTS, (b) compare the codes in terms of various attributes such as length, the number of incomplete targets, etc.

P2. Consider the grammar for translation of control flow constructs (given in Appendix)which uses partial evaluation of boolean expressions. Is the last semantic action, **emit ('goto' M_1.stat)**, in the rule for the **while – loop** absolutely necessary? If your answer is no, rewrite the semantic action for this rule, without changing those for the other rules. Otherwise, produce an input (as small as possible) such that removing the last action leads to generation of erroneous intermediate code.

P3. Apply the SDTS of control flow constructs (given in Appendix) to the following input and show how the intermediate code generation along with evaluation of semantic attributes takes place step by step.

```
while  a+b > c + d do
 begin
   if a > b then  if b > c then  begin  d := b + c;  b := b /2  end
                              else  while  a > 0 do  a := a – 1;
            else   while  b > 0 begin  sum := sum + b; b := b – 1 end
 end;
```

P4. It is intended to add **break statement** as an instance of S to the control flow constructs grammar. The **break** statement terminates the execution of the nearest enclosing loop or conditional statement in which it appears. Control passes to the statement that follows the terminated statement, if any. It is typically used in the switch and loop statements.

S → **break**

(a) Rewrite the grammar given to include the break statement, as described above, by making minimal changes. This part concerns with syntax only and semantic issues may be ignored.

(b) Explain your scheme for semantic action for the break statement and implication of this statement on others in English, justifying the same.

(c) Convert the semantic actions of part (b) above to SDTS to generate intermediate code, using synthesized attributes and globals only. You have to write only a) the actions for the new rules, and (b) new actions for the existing rules, if so required

(d) Use your translation scheme to generate intermediate code for the following input; given that a,b,c and d are integers.

```
while ( a <= b)
begin
      a := c + d;
      if ( a < 0 ) then break
      else
       begin   c := c + d;   if ( c > 0 ) then d := d /c    else  break   end
end;
```

P5. We would like to add a new construct named as **loop – endloop**, to the control flow constructs grammar. The loop – endloop construct is essentially an infinite loop, so its body must have at least one **break** statement for the loop to be semantically valid. The semantics of the break statement is that it causes the innermost enclosing loop to be exited immediately, for our context it is applicable only to the while and the loop - endloop constructs.

S → **loop** S_1 **endloop**

S → **break**

Perform the same actions as mentioned in parts (a) to (c) in P5 for this problem, (d) apply your SDTS to the following input and show how the intermediate code generation along with evaluation of semantic attributes takes place step by step.

```
if   a+b > c + d  then
  begin
    if a > b then  loop begin b := b + c; if b > c then  d := b + c else break end endloop
            else  while  a > 0 do  a := a – 1;
    else   loop  begin while b > 0 do sum := sum + b; break end endloop
   end;
```

P6. We would like to add a new construct **repeat S until B** to the control flow constructs grammar The semantics of this statement is that the body S is executed at least once. Subsequent execution of the body S depends on the boolean expression B : in case B evaluates to false, loop is exited.

S → **repeat** S_1 **until B**

Perform the same actions as mentioned in parts (a) to (c) in P5 for this problem, (d) apply your SDTS to the following input and show how the intermediate code generation along with evaluation of semantic attributes takes place step by step.

```
repeat while ( a <= b)
begin
      a := c + d;
      if ( a < 0 ) then break
      else
       begin   c := c + d;   repeat  d := d /c  until c > 0;  end
end
until ( a <= b)
```

P7. Add a rule for the **for loop** construct to the control flow constructs grammar. Use the C / C++ language syntax and semantics. Write the required semantic analysis in English. Write a SDTS on the same lines as given out in parts (a) to (c) in P5, (d) construct a suitable input string and show the intermediate code generation along with evaluation of semantic attributes using your SDTS.

P8. It is intended to add unconditional **goto statement** as an instance of S with the following syntax :

S → **goto id**

In order to support the semantics for the goto statement, statements are now permitted to be labeled, if so required. Unlabeled statements are also to be supported. Write a SDTS on the same lines as given out in parts (a) to (c) in P5, (d) construct a suitable input string and show the intermediate code generation along with evaluation of semantic attributes using your SDTS.

P9. Add the switch statement of C/C++ to the control flow constructs grammar. Use the C / C++ language syntax and semantics. Write the required semantic analysis in English. Write a SDTS on the same lines as given out in parts (a) to (c) in P5, (d) construct a suitable input string and show the intermediate code generation along with evaluation of semantic attributes using your SDTS.

P10. We would like to process program fragments, as given below for compilation and intermediate code generation.

```
integer  a [10, 20];
integer  b [10, 20];
integer i, j;
i = 10;
j = 16;
b[i+1, j+2] = a [i-1, j+1] + a [i , j – 2] + 25;
```

Make appropriate changes to the Translation Scheme given below for array references so that code as given above can be handled along with parsing. Grammar and semantic rules for processing of declarations (scalars and arrays) are not required but the arrays are to be assumed to be in row-major representation and their attributes may be accessed from the symbol table. The grammar that you write should be able to produce strings of the form given in the last 3 lines of the code above.

S → L := E { **if** L.offset = **null then** emit (L.place ':=' E.place)
else emit (L.place '[' L.offset ']' ':=' E.place) }

E → L { **if** L.offset = **null then** E.place ':=' L.place
else { E.place := newtemp(); emit (E.place ':=' L.place '[' L.offset ']')} }

L → elist] { L.place := newtemp(); L.offset := newtemp(); emit (L.place ':=' c(elist.array));
emit (L.offset '"=' elist.place '*' width (elist.array)) }

L → **id** { L.place := **id**.place; L.offset := **null** }

elist → $elist_1$, E { t := newtemp(); m := $elist_1$.dim + 1; emit (t ':=' $elist_1$.place '*' limit ($elist_1$.array);
emit (t ':=' t '+' E.place); elist.array := $elist_1$.array; elist.place := t; elist.dim := m }

elist → **id** [E { elist.array := **id**.place; elist.place := E.place; elist.dim := 1 }

Recall that for the n dimensional array reference **a**[u_1, u_2, ..., u_n], where u_i's are the upper bounds and the lower bounds are assumed to be 1, the relative address for the reference **a**[e_1, e_2, ..., e_n] is given by c + v, where the expressions of c and v are as given below :

c = addr **a**[1, 1, ..., 1] - ((.... ((u_2 + 1)*u_3 + 1) ... u_n + 1)*w

v = ((... ((e_1 * u_2 + e_2)*u_3+ e_3)* ...)*u_n + e_n)*w

The attribute elist.array is a pointer to the symbol table (ST) for the array name, elist.place is a placeholder for the v part of the address calculation, elist.dim keeps track of the dimension information, c(array_name) returns the c part of the address formula from the ST, width (array_name) gives the width information from ST and limit (array_name,m) returns the upper bound for the dimension m from the symbol table.

Show the intermediate code that should be generated for the last 3 lines for the program fragment.

P11. For the same context as P10. Above, change the grammar rules given above, if so needed, so that assignment expressions of the form shown in the program fragment can be processed for parsing. Assume that a separate terminal, **num**, is given by the lexer when an integer constant is recognized. The only operators permitted in expressions are addition (+) and multiplication (*). Now introduce semantic actions along with your grammar rules so that intermediate code is generated for the assignment expressions of part (a) above. Make sure that changes incorporated in the given rules and actions are minimal and absolutely necessary.

P12. We would like to process program fragments, as given below for compilation and intermediate code generation.

```
integer  a [10] [20];
integer  b [10] [20];
integer i;
integer j;
i = 10; j = 16;
```

b[i][j+2] = a [i-1] [j+1] + a [i + 1] [j – 2];

Make suitable changes to the SDTS given in P10 above so that processing of array references and assignments as given in the last 3 lines of the program fragment can be handled including generation of intermediate code.

P13. Using the same context as of P10 above, the problem here is to process array declarations of the form shown in the program fragment in the first two statements only and references of the form shown in the last statement can be processed for parsing. Introduce grammar rules with semantic actions so that array declarations of the kind given in the first two statements can be processed (ignore the next two). The semantic analysis should store relevant information in the ST for the array declaration, in particular the **c part** of the address calculation should be carried out and kept along with other entries for the array in the ST. Make sure that changes, if any, incorporated in the given rules and actions are minimal and absolutely necessary.

APPENDIX – 1

A. Scheme1 : Full evaluation of boolean expressions :

B → B_1 **or** B_2 { B.place := newtemp(); emit(B.place ':=' B_1.place '**or**' B_2.place)}
B → B_1 **and** B_2 { B.place := newtemp(); emit(B.place ':=' B_1.place '**and**' B_2.place)}
B → **not** B_1 { B.place := newtemp(); emit(B.place ':=' '**not**' B_1.place)}
B → (B_1) { B.place := B_1.place}
B → **true** { B.place := newtemp(); emit(B.place ':=' '1')}
B → **false** { B.place := newtemp(); emit(B.place ':=' '0')}
B → E_1 **relop** E_2 { B.place := newtemp();
emit ('if' E_1.place '**relop.op**' E_2.place 'goto' nextstat + 3);
emit (B.place ':=' '0'); emit ('goto' nextstat+2);
emit (B.place ':=' '1') }
B → E { B.place ;= E.place}

E → E_1 + E_2 | other rules for expressions { semantic actions as discussed earlier }

Scheme2 : Partial evaluation of boolean expressions :

B → B_1 **or** M B_2
{ backpatch (B_1.falselist, M.stat); B.truelist := merge (B_1.truelist, B_2.truelist);
B.falselist := B_2.falselist}
B → B_1 **and** M B_2
{ backpatch (B_1.truelist, M.stat); B.falselist := merge (B_1.falselist, B_2.falselist);
B.truelist := B_2.truelist}
B → **not** B_1 { B.truelist := B_1.falselist ; B.falselist := B_1.truelist }
B → (B_1) { B.truelist := B_1.truelist ; B.falselist := B_1.falselist }
B → **true** { B.truelist := makelist (nextstat); emit ('goto ___');
B → **false** { B.falselist := makelist (nextstat); emit ('goto ___');
B → E_1 **relop** E_2 { B.truelist := makelist (nextstat); B.falselist := makelist (nextstat + 1);
emit ('if' E_1.place 'relop.op' E_2.place 'goto ___'); emit ('goto ___') }
B → E { B.place := E.place; B.truelist := makelist (nextstat); B.falselist := makelist (nextstat + 1);
emit ('if' B.place 'goto ___'); emit ('goto ___') }

M → ε {M.stat := nextstat}

E → E_1 + E_2 | other rules for expressions { semantic actions as discussed earlier }

B. SDTS for control flow constructs

$S \rightarrow$ if B then M S_1
{backpatch (B.truelist, M.stat) ; S.nextlist := merge(B.falselist, S_1.nextlist) }

$S \rightarrow$ if B then M_1 S_1 N else M_2 S {backpatch (B.truelist, M_1.stat) ; backpatch (B.falselist, M_2.stat);
S.next := merge(S_1.nextlist, merge(N.nextlist, S_2.nextlist))}

$S \rightarrow$ while M_1 B do M_2 S_1 {backpatch (B.truelist, M_2.stat); backpatch (S_1.nextlist, M_1.stat);
S.nextlist := B.falselist ; emit ('goto' M_1.stat) }

$S \rightarrow$ begin Sl end {S.nextlist := Sl.nextlist}

$Sl \rightarrow Sl_1$; M S {backpatch (Sl_1.nextlist, M.stat); Sl.nextlist := S.nextlist}

$Sl \rightarrow$ S {Sl.nextlist := S.nextlist}

$S \rightarrow$ L := E { semantic actions as earlier; S.nextlist := nil}

$M \rightarrow \varepsilon$ {M.stat := nextstat}

$N \rightarrow \varepsilon$ {N.nextlist := makelist (nextstat); emit (‘goto ___ ')}

******* End of Tutorial Sheet 7 ********

Released on March 21, 2017
Supratim Biswas