CSCI565 - Compiler Design

Spring 2016

Homework 2 - Solution

Problem 1: Attributive Grammar and Syntax-Directed Translation [40 points]

In this problem you need to develop a grammar for regular expressions over the alphabet \{0,1\} and develop a L-attributed Syntax-Directed Definition for the translation of regular expressions to Non-Deterministic Finite Automata (NFA) using the notions of the Thompson’s construction described in class. Assume a generic “terminal” symbol production of the form \( \text{expr} \rightarrow \text{char} \) where char is a lexeme with a va1 attribute with either 0 or 1 value. Because your translation needs to create a graph, assume you have a set of auxiliary functions that allow you to create, reference the states of the NFA as well as add edges between them. In particular, assume you have a function \( \text{newState} \) that returns a reference to a newly created (and uniquely identified) state as well as a function \( \text{lookupState} \) that returns a reference to a state in the NFA. Also, use the function \( \text{addEdge} \) with two reference parameters and transition character that indicate a transition between states. You can also assign an attribute of ‘start’ or ‘final’ to a given state.

To help you structure your work, indicate the following:

- [05 points] Describe the grammar for regular expressions
- [10 points] Describe the attributes both synthesized and inherited used for your translation.
- [20 points] Describe the translation using the semantic rules you have developed
- [05 points] Show the translation for the Regular Expression \( \text{RE}=1.0^* \)

Solution:

a. A possible (simple) grammar for regular expression is given below with Goal symbol as \( \text{expr} \).

\[
\begin{align*}
(1) & \quad \text{expr} \rightarrow \text{'} ( \text{expr} \text{'}) \\
(2) & \quad \text{expr} \rightarrow \text{expr} \text{'}* \\
(3) & \quad \text{expr} \rightarrow \text{expr} \text{'}| \text{expr} \\
(4) & \quad \text{expr} \rightarrow \text{expr} \text{'}\cdot \text{expr} \\
(5) & \quad \text{expr} \rightarrow \text{char}
\end{align*}
\]

b. We associated with the ‘expr’ non-terminal symbol of the grammar a ‘start’ state and a (current) ‘final’ state attributes. These are references to states in the NFA. Both the ‘start’ state and ‘final’ state attributes are synthesized. We further define a ‘nfa’ attribute that includes the NFA associated with each non-terminal. As such we also assume a graph-copying and graph-manipulation primitives that allows the composition graphs.

c. A possible (simple) L-attributed grammar is given below.

\[
\begin{align*}
(1) & \quad \text{expr}_1 \rightarrow \text{'} ( \text{expr}_2 \text{'}) \quad \{ \text{expr}_1.\text{nfa} = \text{expr}_2.\text{nfa}; \text{expr}_1.\text{start} = \text{expr}_2.\text{start}; \text{expr}_1.\text{final} = \text{expr}_2.\text{final}; \} \\
(2) & \quad \text{expr}_1 \rightarrow \text{expr}_2 \text{'}* \\
& \quad \{ \text{s1} = \text{newState}(); \text{s2} = \text{newState}(); \text{expr}_1.\text{start} = \text{s1}; \text{expr}_1.\text{final} = \text{s2}; \text{expr}_1.\text{nfa}.\text{states} = \text{expr}_2.\text{nfa}.\text{states} + \{ \text{s1}, \text{s2} \}; \text{expr}_1.\text{nfa}.\text{edges} = \text{expr}_2.\text{nfa}.\text{edges} + \{ \text{newEdge}(\text{expr}_1.\text{start},\text{expr}_1.\text{start}); \text{newEdge}(\text{expr}_1.\text{final},\text{expr}_1.\text{final}); \} \}
\end{align*}
\]

\[
\begin{align*}
(3) & \quad \text{expr}_1 \rightarrow \text{expr}_2 \text{'}| \text{expr}_3 \\
& \quad \{ \text{s1} = \text{newState}(); \text{s2} = \text{newState}(); \text{expr}_1.\text{start} = \text{s1}; \text{expr}_1.\text{final} = \text{s2}; \text{expr}_1.\text{nfa}.\text{states} = \text{expr}_2.\text{nfa}.\text{states} + \text{expr}_3.\text{nfa}.\text{states} + \{ \text{s1}, \text{s2} \}; \text{expr}_1.\text{nfa}.\text{edges} = \text{expr}_2.\text{nfa}.\text{edges} + \text{expr}_3.\text{nfa}.\text{edges} + \{ \text{newEdge}(\text{expr}_1.\text{start},\text{expr}_1.\text{start}); \text{newEdge}(\text{expr}_1.\text{final},\text{expr}_1.\text{final}); \} \}
\end{align*}
\]
(4) \( expr_1 \rightarrow expr_2 \cdot expr_3 \)  
\{  
  expr_1.start = expr_2.start; expr_1.final = expr_2.final;  
  expr_1.nfa.states = expr_2.nfa.states + expr_3.nfa.states;  
  expr_1.nfa.edges = expr_2.nfa.edges + expr_3.nfa.edges + \{ \text{newEdge(expr_2.final, expr_3.start, } \varepsilon \text{)} \};  
\}  

(5) \( expr \rightarrow char \)  
\{  
  exp.nfa = \{ s1=newState(); s2 =newState(); addEdge(s1,s2,char.value)};  
  exp.final = s2; exp.start = s1;  
\}  

d. The translation for the RE = 1.0* is shown below.
Problem 2: Static-Single Assignment Representation [10 points]

For the sequence of instructions shown below depict an SSA-form representation (as there could be more than one). Comment on the need to save all the values at the end of the loop and how the SSA representation helps you in your evaluation of the code. Do not forget to include the \( \phi \)-functions.

\[
\begin{align*}
&b = 0; \\
&d = 1; \\
&a = \ldots; \\
&i = \ldots;
\end{align*}
\]

\textbf{L1:} if\( (i > 0) \) {
\begin{align*}
&d = 0; \\
&b = b + 1; \\
&d = 1; \\
&i = i - 1; \\
&\text{if}(i < 0) \\
&\quad a = \ldots; \\
&\quad d = 0; \\
&\quad \text{goto Lbreak;} \\
&\quad \text{goto L1;}
\end{align*}
\}

\textbf{Lbreak:} 
\begin{align*}
&x = a; \\
&y = b;
\end{align*}

\textbf{Solution:}

A possible representation in SSA is as shown below where each value associated with each variable is denoted by a subscripted index.

\[
\begin{align*}
&b_1 = 0 \\
&d_1 = 1 \\
&a_1 = \ldots \\
&i_1 = \ldots
\end{align*}
\]

\textbf{L1:} \( i_2 = \phi(i_1,i_3) \)

\[
\begin{align*}
&b_2 = \phi(b_1,b_3) \\
&\text{if} \ (i_2 \leq 0) \ \text{then goto Lbreak} \\
&d_2 = 0 \\
&b_3 = b_2 + 1 \\
&d_3 = 1 \\
&i_3 = i_2 - 1 \\
&\text{if} \ (i_3 \geq 0) \ \text{then goto Lbreak} \\
&a_2 = \ldots \\
&d_4 = 0; \\
&\text{goto L1}
\end{align*}
\]

\textbf{Lbreak:} \( b_3 = \phi(b_2,b_3) \)

\[
\begin{align*}
&d_5 = \phi(d_1,d_3,d_4) \\
&a_3 = \phi(a_1,a_2) \\
&x_1 = a_3 \\
&y_1 = b_3
\end{align*}
\]

An important aspect of this representation is that it makes explicit the flow of values that reach a given point. In particular and for this case the value for the \( d \) variable is defined by two assignments denoted by \( d_5 = \phi(d_1,d_3,d_4) \). This information can be useful as we now know that the only two possible values for the \( d \) variable on exit of the this code are either 0 or 1.
Problem 3: Symbol Table Organization [10 points]

For the PASCAL code below answer the following questions:

01:   procedure main
02:     integer a, b, c;
03:     procedure f1(w,x);
04:     integer w, x;
05:     procedure f2(y,z);
06:     integer a, y, z;
07:     function f3(m,n):integer;
08:     integer m, n;
09:     function f4(k): integer;
10:     integer k;
11:     x := a * (m+1);
12:     y := b * (n+1);
13:     f3 := x + y;
14:     f4 := (k + 1);
15:     f2 := f3(c,z) + f4(c);  
16:     for f4:
17:     end;
18:     for f3:
19:     end;
20:     for f2:
21:     end;
22:     f1(a,b);
23:     end

a) [05 points] Draw the symbol tables for each of the procedures in this code (including main) and show their nesting relationship by linking them via a pointer reference in the structure (or record) used to implement them in memory. Include the entries or fields for the local variables, arguments and any other information you find relevant for the purposes of code generation, such as its type and location at run-time.

b) [05 points] For the statement in line 19 what are the specific instance of the variables used in this statement the compiler needs to locate? Explain how the compiler obtains the data corresponding to each of these variables table at compile time.

Solution:

a) The figure below depicts the hierarchical structure of the procedure in this PASCAL program.

b) For the statement in line 19 we simply follow the symbol table entries to find out the specific instance of each of the symbols. Given than the statement is located lexically inside the body of procedure f2 the search for symbols always begins in the symbol table for f2. In this statement "f2 := f3(c,z) + f4(c)," the symbol z refers to the local variable of procedure f2, so trivially, this will be accessible from the f2's own activation record (AR) accessible via the ARP or stack pointer (SP) used in many processor architectures. The symbol c corresponds to the main variable c and is also located at the AR of main or depending on the implementation, as a specific location in the codes data segment.
Problem 4: Intermediate Code Generation [10 points]
In C++ and other object-oriented programming languages, the access to data members, or fields of an object, is accomplished by the ‘.’ and ‘->’ operators. The choice of the use of these two operators depends on the l-value of the object expression. If it denotes a reference to an object the ‘->’ should be used, if a name of an object allocated automatically or statically, the ‘.’ operator must be used.

In this context, consider the code generation for expressions that access the data members or fields of objects. To do this you need to have access to the offset of the object where the field is located. This information should be stored in the symbol table that saves the class declaration and computes the size of each data field. Your code must also determine the address (base address) of the object that should be captured in the lhs of an expression.

As an example, the code below:

\[ a = b.f; \]

should lead to the generation of the following intermediate code where we have symbolically used the function offset(f) as the function that looks up the integer offset of the f field for the class b:

\[
\begin{align*}
  t1 &= &b; \\
  t1 &= t1 + \text{offset(class(b),f)}; \\
  t1 &= *t1; \\
  a &= t1;
\end{align*}
\]

In this context answer the following:

a) [05 points] Derived a SDT code generation scheme that handles expressions such as the ones described above as well as the ones where the l-value of an expression is a reference (and thus the ‘->’ operator must be used).

b) [05 points] Show your code generation scheme for the simple expression “\(c = a.f1 + b->f2\)” assuming the the fields \(f1\) and \(f2\) do exist in the corresponding class declarations.

Solution:

a) All the attributes in this solution are synthesized as that helps the integration with a bottom-up parser and also with a single bottom-up traversal of the tree.

<table>
<thead>
<tr>
<th>attribute</th>
<th>definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>symbol to be used with the symbol table</td>
</tr>
<tr>
<td>pointer</td>
<td>boolean (true if this is a pointer field)</td>
</tr>
<tr>
<td>place</td>
<td>temporary name</td>
</tr>
<tr>
<td>code</td>
<td>list of instructions</td>
</tr>
</tbody>
</table>

Using these attributes, we also defined a set of simple auxiliary functions, namely:

\[
\begin{align*}
  \text{int} &:: \text{getOffset(user-defined class type, data_member_name);} \\
  \text{type} &:: \text{getClassDefinition(symbol_name);} \\
  \text{type} &:: \text{getTypeOfClassDataMember(class_name, data_member_name);} \\
\end{align*}
\]

As to the semantic rules we can define them as follows:

\[
\begin{align*}
  \text{assign} \rightarrow \text{id} \ '==' \ \text{exp}_1 & \quad \{ \ t4 = \text{newtemp}(); \\
  & \text{assign}.place = t4; \\
  & \text{assign}.code = \text{append(}\text{exp}_1.\text{code, gen(}'t4 = \text{exp}_1.'\text{)})\}; \\
  \text{assign}.pointer &= \text{exp}_1.\text{pointer}; \\
  \text{assign}.type &= \text{getClassDefinition(}\text{id}.\text{symbol})\}; \\
\end{align*}
\]

\[
\begin{align*}
  \text{exp}_1 \rightarrow \text{exp}_2 \ '++' \ \text{exp}_3 & \quad \{ \ t3 = \text{newtemp}(); \\
  & \text{exp}_1.\text{place} = t3; \\
  & \text{exp}_1.\text{code} = \text{append(}\text{exp}_2.\text{code, exp}_3.\text{code, gen(}'t3 = \text{exp}_1.\text{place} + \text{exp}_3.\text{place}')\}; \\
  & \text{exp}_1.\text{pointer} = \text{exp}_2.\text{pointer}; \\
  & \text{exp}_1.\text{type} = \text{exp}_2.\text{type}; \\
\end{align*}
\]
exp → rval
   { exp.code  = rval.code;
     exp.type  = rval.type;
     exp.pointer = rval.pointer;
     exp.place  = rval.place;
   }

rval₁ → rval₂ ’->’ field
   { offset = getFieldOffset(rval₂.type,field.symbol);
     t2 = newtemp();
     rval₁.place = t2;
     rval₁.type = getTypeOfClassDataMember (rval₂.type,field.symbol);
     if( isPointerType(rval₂.type) ){
       rval₁.code = append(rval₂.code,{'gen('t2 = rval₂.place + offset'; 't2 = *t2;');
       rval₁.pointer = true;
     } else {
     error
   }

rval₁ → rval₂ ’.’ field
   { offset = getFieldOffset(rval₂.type,field.symbol);
     t2 = newtemp();
     rval₁.place = t2;
     rval₁.type = getTypeOfClassDataMember (rval₂.type,field.symbol);
     if( isPointerType(rval₂.type) ){
     error;
   } else {
     rval₁.code = append(rval₂.code,{'gen('t2 = rval₂.place + offset'; 't2 = *t2;});
     rval₁.pointer = isPointerType(rval₁.type);
   }

rval₁ → id
   { rval₁.type = getType(id.symbol);
     rval₁.pointer = isPointerType(id.symbol);
     t1 = newtemp();
     rval₁.place = t1;
     rval₁.code = {'gen('t1 = id.symbol;');

b)
Problem 5: Fault-Tolerance Constructs and Back-patching [30 points]

We have covered in class an SDT scheme to generated code using the back-patching technique for a while loop construct. In this exercise you will develop a similar scheme for a construct akin to the try-throw in Java. Specifically, the construct has the syntax illustrated by the example below where in the presence of an error in the execution of the body, the computation is repeated up to a number N of times. It is of course the responsibility of the programmer that the repetition of the computation does not have un-intended side-effects.

```plaintext
try exp1 { 
  ... code ...
} throw exp2;
```

and where the expressions exp1 and exp2 evaluate to integer values. A subtle point is that the value of the number of retries (exp1) needs to be evaluated on entry of the construct and so changes to any of its operands in the code section of the construct have no effect on the possible maximum number of times the code is repeated.

As the execution of the construct needs to examine for errors at run time, the compiler will check at the end of the construct if a special environment variable, named hw_error is non-zero. The variable also needs to be explicitly cleared upon entry of the construct. In case the variable assumes the zero value at the end of the first execution of the body, there is no re-execution and the computation proceeds.

A possible excerpt of a grammar that captures this construct is depicted below where the constructs for the expr non-terminal symbols are analogous to the ones used to define arithmetic expressions as described in class.

1. stat → try exp1 statlist throw exp2
2. statlist → stat ';' statlist
3. statlist → stat
4. stat → ...

In this exercise you are asked to develop a SDT scheme possibly using the back-patching technique to generate three-address code that implements the semantics of this construct. Should it be needed do not forget to show the augmented production with the marker non-terminal symbols, M and possibly N along with the corresponding rules for the additional symbols and productions. Argue for the correctness of your solution without necessarily having to show an example.

**Solution:**

We are going to make use of auxiliary markers and the corresponding productions to set the addresses in the case where in a later phase of the code generation we can jump into. As a result these marker non-terminals have a synthesized attribute that is a ‘jumpList’ where an unresolved ‘goto’ will be patched later with the correct code.

```plaintext
(1) S → try exp1 M1 '{ StatList '}' try M2 exp2 
{ 
  t1 = newtemp(); // t1 - holds the max number of tries
  t2 = newtemp(); // t2 - holds the flag for error condition
  t3 = newtemp(); // t3 - holds the hw_error volatile variable value
  n = nextAddr();
  gen("t2 = 0");
  gen("t1 = exp1.place");
  emit("goto M1.address+1");
  gen("t3 = hw_error");
  gen("if t3 = 0 goto n+8");
  gen("if t2 >= t1 goto n+8");
  emit("t2 = t2 + 1");
  emit("t3 = 0");
  gen("goto M1.address+1");
  gen("if t3 = 0 goto __");
  gen("putparam exp1.place");
  gen("call exit, 1");
  S.nextlist = newList(n+8);
  backpatch(M1.jumplist, M1.address);
  backpatch(M2.jumplist, M2.address);
}
```
(2) $S \rightarrow \text{continue}$  
\{ S.skiplist = newList(nextAddr()); emit('goto __'); S.breaklist = nil; S.nextlist = nil; \}  

(3) $S \rightarrow \text{break}$  
\{ S.breaklist = newList(nextAddr()); emit('goto __'); S.nextlist = nil; S.skiplist = nil; \}  

(4) $\text{StatList}_1 \rightarrow \text{Stat} \; '; \; \text{M}_1 \; \text{StatList}_2$  
\{  
backpatch(S.nextlist, M_1.addr);  
L_1.nextlist = L_2.nextlist;  
L_1.breaklist = merge(S.breaklist,L_2.breaklist);  
L_1.skiplist = merge(S.skiplist,L_2.skiplist);  
\}  

(5) $\text{StatList} \rightarrow \text{Stat}$  
\{ L.breaklist = S.breaklist; L.nextlist = S.nextlist; L.skiplist = S.skiplist; \}  

(6) $M_1 \rightarrow \epsilon$  
\{ M_1.address = nextAddr; emit('goto __'); M_1.jumplist = nextAddr; \}  

(7) $M_2 \rightarrow \epsilon$  
\{ M_2.address = nextAddr; emit('goto __'); M_2.jumplist = nextAddr; \}  

(8) $M_3 \rightarrow \epsilon$  
\{ M_2.address = nextAddr; \}  

As an error correction step, and as part of the rules associated with this construct you could include a test to check if the ‘skiplist’ and ‘breaklist’ attributes of the list of statement are in fact empty, thus revealing that the enclosed statement do not include break or return statements.