CSCI 565 - Compiler Design

Spring 2015

Midterm Exam

March 04, 2015 at 8:00 AM in class (RTH 217)

Duration: 2h 30 min.

Please label all pages you turn in with your name and student number.

Name: ___________________________ Number: ________

Grade:
  Problem 1 [30 points]:
  Problem 2 [20 points]:
  Problem 3 [20 points]:
  Problem 4 [30 points]:

Total:

____________________________________________________________________________

INSTRUCTIONS:

1. This is a open-book exam and you may notes either typed or handwritten for your own personal use during the exam.

2. Please identify all the pages where you have answers that you wish to be graded. Also make sure to label each of the additional pages with the problem you are answering.

3. Use black or blue ink. No pencil answers allowed.

4. Staple or bind additional answer sheets together with this document to avoid being misplaced or worse, lost. Make sure this cover page is stapled at the front.

5. Please avoid laconic answers so that we can understand you understood the concepts being asked.
Problem 1: Context-Free-Grammars and Parsing Algorithms [30 points]

Consider the CFG fragment with non-terminal symbols \{ \text{decl, type, id\_list} \}, with start symbol S, terminal symbols \{ \text{id, , ; var, integer, real} \} and the productions P listed below.

(1) \text{decl} \rightarrow \text{var id\_list} : \text{type} ;$
(2) \text{type} \rightarrow \text{integer}$
(3) \text{type} \rightarrow \text{real}$
(4) \text{id\_list} \rightarrow \text{id\_list} , \text{id}$
(5) \text{id\_list} \rightarrow \text{id}$

Questions:

a) [05 points] Is this grammar suitable to be parsed using a predictive LL(0) parsing algorithm? Why?

b) [15 points] Compute the DFA that recognizes the LR(0) sets of items for this grammar and construct the corresponding LR(0) parsing table. Comment on the nature of the conflicts, if any.

c) [10 points] Typically the LR(0) table construction algorithm leads to tables that are very 'dense'. Construct the SLR parsing table by indicating the FOLLOW sets used to trim the entries on the table.

Solution:

a) No as the grammar is left-recursive in the definition of id\_list.

b) Below is the DFA for the LR(0) sets of items and the corresponding LR(0) table.

c) The SLR parsing table is constructed by indicating the FOLLOW sets used to trim the entries on the table.
c) The only difference between these two tables is that in the case of a state where there is a reduction, we will only include the 'reduce' action for the terminals that belong to the FOLLOW of the corresponding rule's LHS.

We only need to compute the following FOLLOW sets:

\[
\text{FOLLOW(type)} = \{ \cdot, \} \\
\text{FOLLOW(id\_list)} = \{ \cdot, \cdot, \cdot \}
\]

yielding the SLR parsing table below.

<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
<th>Goto</th>
</tr>
</thead>
<tbody>
<tr>
<td>s0</td>
<td>var</td>
<td>shift s1</td>
</tr>
<tr>
<td>s1</td>
<td>id</td>
<td>shift s3</td>
</tr>
<tr>
<td>s2</td>
<td>:</td>
<td>goto s3</td>
</tr>
<tr>
<td>s3</td>
<td>;</td>
<td>shift s4</td>
</tr>
<tr>
<td>s4</td>
<td>real</td>
<td>shift s5</td>
</tr>
<tr>
<td>s5</td>
<td>integer</td>
<td>shift s6</td>
</tr>
<tr>
<td>s6</td>
<td>$</td>
<td>goto s9</td>
</tr>
<tr>
<td>s7</td>
<td>var</td>
<td>reduce (5)</td>
</tr>
<tr>
<td>s8</td>
<td>id_list</td>
<td>reduce (5)</td>
</tr>
<tr>
<td>s9</td>
<td>;</td>
<td>reduce (4)</td>
</tr>
<tr>
<td>s10</td>
<td>$</td>
<td>shift s10</td>
</tr>
<tr>
<td></td>
<td>$</td>
<td>accept</td>
</tr>
</tbody>
</table>
### Problem 2: Syntax-Directed Translation [20 points]

In Pascal a programmer can declare two integer variables a and b with the syntax

\[
\text{var a, b : integer}
\]

This declaration might be described by the grammar in problem 1 above:

1. \( \text{decl} \rightarrow \text{var id_list : type ;} \)
2. \( \text{type} \rightarrow \text{integer} \)
3. \( \text{type} \rightarrow \text{real} \)
4. \( \text{id_list} \rightarrow \text{id_list , id} \)
5. \( \text{id_list} \rightarrow \text{id} \)

where \( \text{id_list} \) derives a comma separated list of variable names and \( \text{type} \) derives a valid Pascal type. You may find it necessary to rewrite the grammar.

(a) [10 points] Write an attribute grammar that assigns the correct data type to each declared variable and show the values of your attributes for the example given above (considering that the size of an integer scalar variable is say 4).

(b) [05 points] Determine an evaluation order of the attributes irrespective of the way the parse tree is constructed.

(c) [05 points] How would you change the grammar to ensure that all the attributes would be synthesized attributes?

(d) [10 points] Modify the original grammar to eliminate the left-recursion, and describe how to implement the corresponding attributive grammar using a table-based LL predictive parser.

### Solution:

a) We define an attribute grammar with an integer (or enumerated if you prefer) attribute named \( \text{type} \) for the non-terminal symbols \( \text{id} \) and \( \text{id_list} \) as well as the attribute \( \text{value} \) for the terminal symbol \( \text{type} \). The grammar productions and corresponding rules are as shown below where we have ignored the \( \text{decl} \) non-terminal symbols as it plays no rule in the evaluation process.

\[
\begin{align*}
\text{decl} & \rightarrow \text{'}var\text{'} \text{id_list} \text{'}:\text{'} \text{type} \text{'}:\text{'}; \quad \| \quad \text{id_list.type} = \text{type.value} \\
\text{id_list}_0 & \rightarrow \text{id_list}_1 \text{'}:\text{'} \text{ID} \quad \| \quad \text{id_list}_1.type = \text{id_list}_0.type; \quad \text{id.type} = \text{id_list}_0.type; \\
& \rightarrow \text{id} \quad \| \quad \text{id.type} = \text{id_list}_0.type;
\end{align*}
\]

b) As can be observed the attribute \( \text{type} \) is an inherited attribute being first generated by the rule associated with the first production of the grammar, which is subsequently propagated down the parse tree along the \( \text{id_list} \) nodes.
c) One can change the grammar without changing the accepted language as shown below where we have also depicted the revised attribute rules, now only involving synthesized attributes.

\[
\begin{align*}
\text{decl} & \rightarrow \ 'var' \ \text{id} \ \text{id\_list} \ || \ \text{id\_type} = \text{id\_list\_type} \\
\text{id\_list}_0 & \rightarrow \ ',;' \ \text{id} \ \text{id\_list}_1 \ || \ \text{id\_type} = \text{id\_list}_1\_type; \ \text{id\_list}_0\_type = \text{id\_list}_1\_type \\
& \rightarrow \ ',;' \ \text{type} \ ';' \ || \ \text{id\_list}_0\_type = \text{type\_value}
\end{align*}
\]

\[
\begin{align*}
\text{decl} & \rightarrow \ 'var' \ \text{id\_list} \ ':;' \ \text{type} \ ';'
\text{id\_list} & \rightarrow \ \text{ID} \ \text{id\_list}'
\text{id\_list}' & \rightarrow \ ',;' \ \text{id} \ \text{id\_list}' \\
& \mid \ \varepsilon
\end{align*}
\]

Regarding the implementation of the attributed grammar resulting after this transformation we are left with the implementation of an attributive grammar that can be evaluated using an LL table-based parsing algorithm as follows. When the top production is expanded the value of the inherited attribute of \text{type} is left on the stack at the very bottom (we assume here that being a terminal symbol its attribute value is a constant). Then \text{var} is shifted and removed from the stack, leaving \text{id\_list} exposed. Once that is expanded we can 'access' the value of \text{type} two position down in the stack and set the attribute of \text{id\_list}' as this is pushed onto the stack. With \text{id} now on top of the stack, the attribute of \text{id\_list}' is just underneath it. The situation repeats itself for the other production of \text{id\_list}'. The inherited attributes can be evaluated and set on the correct symbols once they are passed onto the stack.
Problem 3. Intermediate Code Generation [20 points]

Suppose your programming language provided support for the exponentiation operation in both integers and floating-point representations with the syntax as shown below where the exponent is \( E_2 \) and the base operand is \( E_1 \). For the purpose of this exercise, you can ignore the base type of the operand but that the exponent needs to be a positive integer.

\[
E \rightarrow E_1 ^ E_2
\]

Questions:

(a) [15 points] Write an SDT scheme for this exponentiation operator generating code that checks that the exponent is a non-negative number. Your solution will clearly include some localized control flow as you may not assume that there is a native exponentiation instruction. Show your work for an illustrative expression. When the exponent is a negative integer your generated code should execute a 'trap' instruction.

(b) [05 points] Revise your solution implementing specific code to test for the cases of \( E_2 = 0 \) and \( E_2 = 1 \) when this is known at compile time and/or at run-time. For this part you have to assume that the non-terminal expression includes other attributes in addition to the attributes place and code we have discussed in class. These could include for instance an attribute for type and an attribute to check if the corresponding value is a compile-time constant.

Solution:

a) A generic code generation scheme could be as shown below.

\[
E \rightarrow E_1 ^ E_2 \quad || \quad t1 = \text{newtemp()}
\quad \quad t2 = \text{newtemp()}
\quad \quad E.\text{place} = t2
\quad \quad E.\text{code} = \text{append}(E_1.\text{code},E_2.\text{code},
\quad \quad \quad \quad \quad \quad \quad \quad \text{gen}('\text{if } E_2.\text{place} < 0 \text{ goto } E.\text{nexstat}+7')
\quad \quad \quad \quad \quad \quad \quad \quad \text{gen}('t1 = E_2.\text{place}')
\quad \quad \quad \quad \quad \quad \quad \quad \text{gen}('t2 = 1')
\quad \quad \quad \quad \quad \quad \quad \quad \text{gen}('\text{if } t1 = 0 \text{ goto } E.\text{nexstat}+0')
\quad \quad \quad \quad \quad \quad \quad \quad \text{gen}('t2 = t2 * E_1.\text{place}')
\quad \quad \quad \quad \quad \quad \quad \quad \text{gen}('t1 = t1 - 1')
\quad \quad \quad \quad \quad \quad \quad \quad \text{gen}('\text{goto } E.\text{nexstat}+3')
\quad \quad \quad \quad \quad \quad \quad \quad \text{gen}('\text{trap}')
\quad \quad \quad \quad \quad \quad \quad \quad \text{gen}('E.\text{place} = t2')
\quad \quad \quad \quad \quad \quad E_1.\text{nextstat} = E.\text{nexstat}
\quad \quad \quad \quad \quad \quad E_2.\text{nextstat} = E_1.\text{laststat}
\quad \quad \quad \quad \quad \quad E.\text{laststat} = E_2.\text{laststat} + 1
\]

For an expression of the form "\( x = a ^ b \)" the resulting generated intermediate code would be:

\[
00: \text{if } E_2.\text{place} < 0 \text{ goto } 07 \\
01: \quad t1 = b \\
02: \quad t2 = 1 \\
03: \text{if } t1 = 0 \text{ goto } 08 \\
04: \quad t2 = t2 * a \\
05: \quad t1 = t1 - 1 \\
06: \text{goto } 03 \\
07: \text{trap} \\
08: \quad x = t2
\]
b) At the compiler level one could check if in fact the expression \( E_2 \) is a compile-time constant, and if so, what its value is. In the specific cases where the value is either 0 or 1, we can generate a very simplified code as shown below. In this particular case we assume that the expression non-terminal also include two additional attributes, namely a \texttt{constant} and a \texttt{type} attribute with obvious types. Further, in the case of a constant, we assume the existence of a \texttt{value} attribute.

\[
E \rightarrow E_1 \^ \ E_2 \parallel \\
\text{t1 = newtemp()} \\
\text{t2 = newtemp()} \\
\text{E.place = t2}
\]

\[
\text{if (E_2.type = integer) AND (E_2.constant = TRUE)} \\
\quad \text{if (E_2.value = 0)} \\
\qquad \text{E.code = append(E.code, gen('E.place = 1'))} \\
\quad \text{elseif (E_2.value = 1)} \\
\qquad \text{E.code = append(E.code, gen('E.place = E_1.place'))} \\
\quad \text{else} \\
\qquad \text{<generic rule shown above>} \\
\quad \text{else} \\
\qquad \text{<generic rule shown above>}
\]

\[
\text{E_1.nextstat = E.nexstat} \\
\text{E_2.nextstat = E_1.laststat} \\
\text{E.laststat = E_2.laststat + 1}
\]
Problem 4. Activation Records and Stack Layout [30 points]

Under the assumption that the AR are allocated on the stack with the individual layout as shown below, and given the PASCAL code on the right-hand-side answers the following questions:

(a) [05 points] Draw the call tree starting with the invocation of the main program.
(b) [15 points] Draw the set of ARs on the stack when the program reaches line 16 in procedure P2. Include all relevant entries in the ARs and use line numbers for the return addresses. Draw direct arcs for the access links and clearly label the values of local variables and parameters in each AR.
(c) [10 points] For this particular example would there be any advantage of using the Display Mechanism?

Solution:

(a) and (b) See the call tree on the left and the stack organization on the right below where the values of the access link and activation records pointers are explicit.
(c) None at all. Given that there are no nested procedures the only non-local variables that a given procedure needs to access are the global variables (like in C). So you only need to have the ARP (or Frame-Pointer – FP) and an second GP or Global Pointer registers. Most architectures do support these two registers in hardware.