CSCI 565 - Compiler Design

Spring 2014

Midterm Exam - Solution

Problem 1: Context-Free-Grammars and Parsing Algorithms [25 points]

Consider the CFG fragment with non-terminal symbols {S, A, B}, with start symbol S, terminal symbols { **id** } and the productions P listed below.

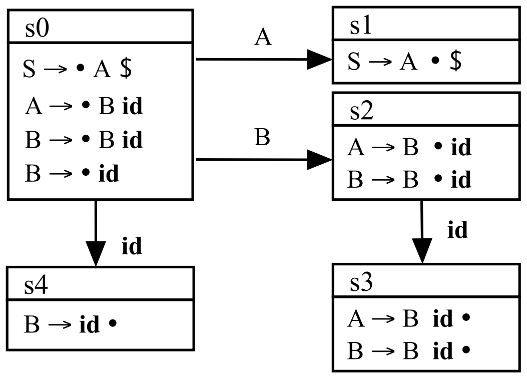
(1) $S \rightarrow A\ \$$
(2) $A \rightarrow B$ **id**
(3) $B \rightarrow B$ **id**
(4) $B \rightarrow$ **id**

Questions:

a) [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.
b) [10 points] Rewrite the grammar to avoid the conflicts found in b). If there are any additional conflicts describe how the SLR table-construction algorithm would resolve them.

Solution:

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



State	Action		Goto	
	id	$	A	B
s0	shift s4		goto s1	goto s2
s1		accept		
s2	shift s3			
s3	reduce (2/3)	reduce (2/3)		
s4	reduce (4)	reduce (4)		

As can be observed there are two reduce/reduce conflicts in state s3 corresponding to the productions (2) and (3). We can try to address these conflicts by computing the FOLLOW sets for the non-terminals A and B and see if we can solve them.

FOLLOW(A) = { \$ }, FOLLOW(B) = { id }

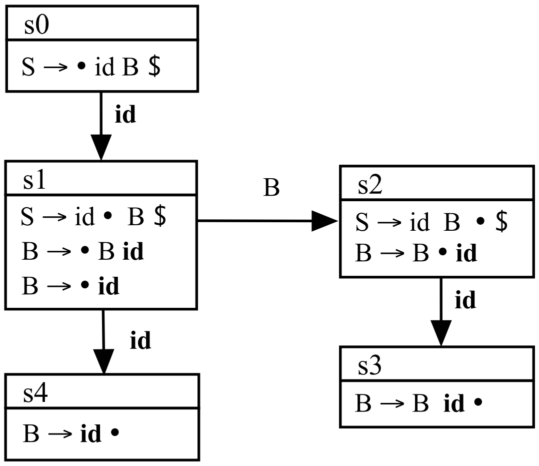
Using this knowledge, we can now construct the corresponding SLR table as shown below:

State	**Action**		**Goto**	
	id	$	A	B
s0	shift s4		goto s1	goto s2
s1		accept		
s2	shift s3			
s3	reduce (3)	reduce (2)		
s4	reduce (4)	reduce (4)		

As can be seen we are able to resolve the conflicts using this SLR table construction algorithm.

b) Another way to resolve this conflict is by observing a 'replication' of the RHS of production (3) and thus eliminate the non-terminal A and all of its productions using the appropriate substitution. The resulting grammar is as shown below. There was a trick here that one has to be careful about. Just replacing the non-terminal A with B in production (1) would not lead to the same language (see below).

(1) S → **id** B $
(2) B → B **id**
(3) B → **id**



State	**Action**		**Goto**
	id	$	B
s0	shift s1		
s1	shift s4		goto s2
s2	shift s3	accept	
s3	reduce (2)	reduce (2)	
s4	reduce (3)	reduce (3)	

With this revised grammar there is no conflict and no need to use the SLR table construction algorithm to resolve them.

As alluded to above the transformed grammar:

(1) S → B $
(2) B → B **id**
(3) B → **id**

is not equivalent to the original grammar as the language generated by the original grammar would not include the word "id" and this transformed one would.

Problem 2: Syntax-Directed Translation [30 points]
In Pascal a programmer can declare two integer variables a and b with the syntax

```
var a, b : integer
```

This declaration might be described with the following grammar:

VarDecl	→	'var' IDList ':' TypeID
IDList	→	IDList ',' ID
	|	ID

where IDList derives a comma separated list of variable names and TypeID 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 TypeID for an integer 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 `type` for the non-terminal symbols Id and IDList as well as the attribute `value` for the terminal symbol TypeID. The grammar productions and corresponding rules are as shown below where we have ignored the VarDecl non-terminal symbols as it plays no rule in the evaluation process.

VarDecl	→	'var' IDList ':' TypeID	|| IDList.type = TypeID.value
IDList$_0$	→	IDList$_1$ ',' ID	|| IDList$_1$.type = IDList$_0$.type; ID.type = IDList$_0$.type;
	→	ID	|| ID.type = IDList$_0$.type;

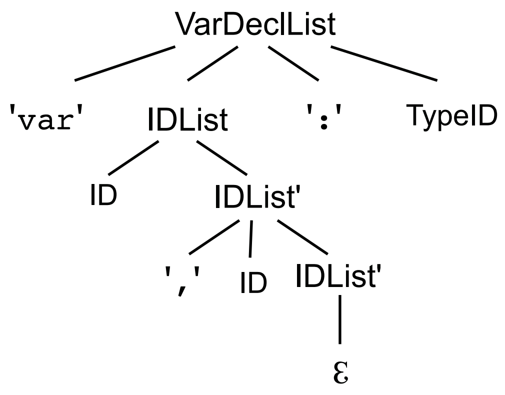
b) As can be observed the attribute type is an inherited attribute being first generated by the rule associated with the first production of the grammar, which is subsequently propagated down the parse tree along the IDList nodes.

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.

VarDecl	→	'var' ID List	||	ID.type = List.type
List$_0$	→	',' ID List$_1$	||	ID.type = List$_1$.type; List$_0$.type = List$_1$.type
	→	':' TypeID	||	List$_0$.type = TypeID.value

d) One can change the grammar as shown below where we have eliminated the left-recursion. On the right-hand side we show a sample AST using this grammar for a declaration of two variables of a generic type.

VarDecl	→	'var' IDList ':' TypeID
IDList	→	ID IDList'
IDList'	→	',' ID IDList'
	|	ε



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 typeID is left on the stack at the very bottom (we assume here that being a terminal symbol its attribute value is a constant). Then 'var' is shifted and removed from the stack, leaving IDList exposed. Once that is expanded we can 'access' the value of typeID two position down in the stack and set the attribute of IDList' as this is pushed onto the stack. With ID now on top, the attribute of IDList' is just underneath it. The situation repeats itself for the other production of IDList'. The inherited attributes can be evaluated and set on the correct symbols once they are passed onto the stack.

Problem 3. Intermediate Code Generation [30 points]

Suppose that your language had an exclusive-OR (EXOR) logical operator and the corresponding syntax as shown below:

$$E \rightarrow E_1 \;\hat{}\; E_2$$

Questions:

(a) [10 points] Write an SDT scheme for this EXOR operator using the arithmetic-based representation of boolean expression under the assumption that the target machine architecture does include an EXOR logic instruction. Show your work for an illustrative expression.

(b) [20 points] Write an SDT scheme using short-circuit evaluation for this logic operator. Notice that this is not as trivial as it sounds as the evaluation of the expression should only yield a true value if and only if only one of the operands is true. Show your work for an illustrative expression.

Solution:

a) For the arithmetic-based representation of boolean the code generation SDT would be a trivial modification of the rules described in class for the binary AND or OR operator as shown below.

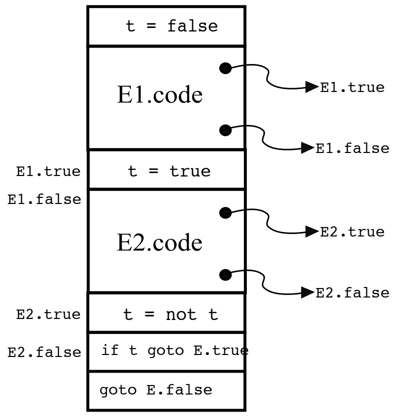
```
E → E1 ^ E2 ||  E.place = newtemp()
               E.code=append(E1.code,E2.code,gen(E.place=E1.place^ E2.place)
               E1.nextstat = E.nexstat
               E2.nextstat = E1.laststat
               E.laststat = E2.laststat + 1
```

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

```
00: if a < b goto 03
01: t1 = 0
02: goto 04
03: t1 = 1
04: if c < d goto 07
05: t2 = 0
06: goto 08
07: t2 = 1
08: t3 = t1 ^ t2
```

b) The difficulty of this control-flow generation scheme is that after the evaluation of the first term of the EXOR one still needs to evaluate the second term. If the firs term evaluates to false, then we would like the goto labels of the code corresponding to the second term to be the same as the goto of the overall boolean expression. If, however, the first term evaluates to true, we would like these goto labels to be switched.

An elegant way to deal with this is to use a single auxiliary boolean variable, say t, which is initialized to false. If the code corresponding to the first term of the expression evaluates to true, the corresponding target label could include an assignment of that variable to true. The evaluation of the second term of the expression should follow and should this evaluation hold true the value of the auxiliary variable should be reversed. Lastly, a single conditional goto will transfer control to the true or false label of the overall boolean expression. The code layout below depicts this code structure.



```
t = newtemp
E1.true  = newlabel
E1.false = newlabel
E2.true  = newlabel
E2.false = newlabel
E.code = append (
    gen('t = false;'),
    E1.code,
    gen('E1.true: t = true;'),
    gen('E1.false: '),
    E2.code,
    gen('E2.true: t = not t;'),
    gen('E2.false: if (t) goto E.true;'),
    gen('goto E.false;')
)
```

For the sample example of a boolean expression used in a) this SDT scheme would generate the code below:

```
00: t = false
01: if a < b goto 03
02: goto 04
03: t = true
04: if c > d goto 06
05: goto 07
06: t = not t
07: if (t) goto E.true
08: goto E.false
```

Another alternative code generation scheme would make partial use of the arithmetic-based approach by having the evaluation of the two boolean expression terms derive their results in two independent auxiliary variables, say t1 an t2, followed by a simple EXOR logical operand and the condition jump for the E.true and E.false labels in indicated by the inherited attributes.

Problem 4. SSA Representation [15 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 ϕ-functions.

```
            b = 0;
            d = ...;
            a = ...;
            i = ...;
 Lloop:     if(i > 0) {
              if(a < 0){
                b = i+1;
              } else {
                d = 1;
              }
              i = i - 1;
              if(i < 0)
                goto Lbreak;
              goto Lloop;
            }
Lbreak:     x = a;
            y = b;
```

Solution:

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

```
        b₁ = 0
        d₁ = ...;
        a₁ = ...;
        i₁ = ...;
X:      i₂ = ϕ(i₁,i₃)
        if (i₂ > 0) then goto Y
        if (a₁ >= 0) then goto Z
        b₂ = i₂ + 1
        goto Z
        d₂ = 1
Z:      i₃ = i₂ - 1
        if (i₃ > 0) then goto Y
        goto X
Y:      b₃ = ϕ(b₁,b₂)
        x₁ = a₁
        y₁ = b₃
```

As can be observed by inspection each use has a single definition point that reaches it and each value is defined only once.