# CSCI 565 – Compiler Design

Spring 2010

# Solution to the Midterm Exam

---

**Problem 1: LR Top-Down Parsing [30 points]**
Given the following CFG grammar G = ({S,A,B}, S, {a, b, x), P) with P:

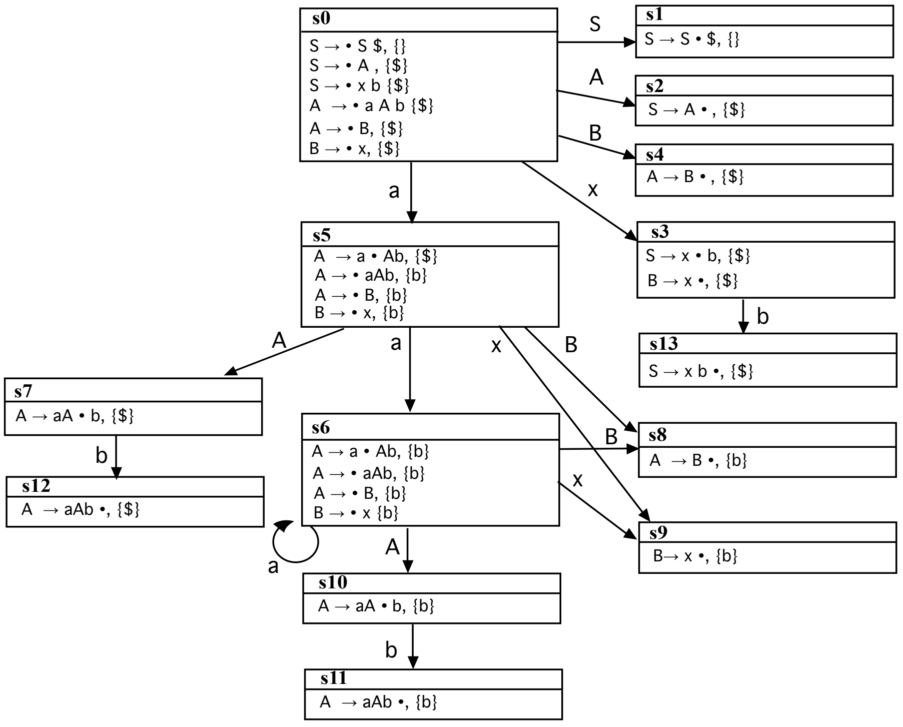
(1) S → A
(2) S → xb
(3) A → aAb
(4) A → B
(5) B → x

For this grammar answer the following questions:

a) [10 pts] Compute the set of LR(1) items for this grammar and the corresponding DFA. Do not forget to augment the grammar with the default initial production S' → S$ as the production (0).
b) [05 pts] Construct the corresponding LR parsing table.
c) [05 pts] Would this grammar be LR(0)? Why or why not? (Note: you do not need to construct the set of LR(0) items but rather look at the ones you have already made for LR(1) o answer this question.
d) [05 pts] Show the stack contents, the input and the rules used during parsing for the input string w = axb$
e) [05 pts] Would this grammar be suitable to be parsed using a top-down LL parsing method? Why?

## Solution:

a) The set of LR(1) items and the corresponding DFA are shown below:

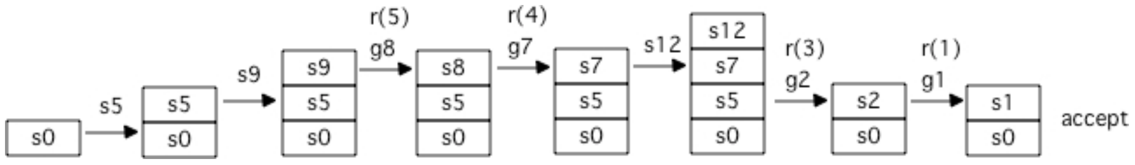


b) The parsing table is as shown below:

| | **terminals** | | | | **goto** | | |
|---|---|---|---|---|---|---|---|
| **State** | a | b | x | $ | S | A | B |
| **0** | s5 | | s3 | | g1 | g2 | g4 |
| **1** | | | | acc | | | |
| **2** | | | | r(1) | | | |
| **3** | | s13 | | r(5) | | | |
| **4** | | | | r(4) | | | |
| **5** | s6 | | s9 | | | g7 | g8 |
| **6** | s6 | | | | | | |
| **7** | | s12 | | | | | |
| **8** | | r(4) | | | | | |
| **9** | | r(5) | | | | | |
| **10** | | s11 | | | | g10 | |
| **11** | | r(3) | | | | | |
| **12** | | | | r(3) | | | |
| **13** | | | | r(2) | | | |

c) If we were to compute the set of LR(0) items for state s3 we would have a shift-reduce conflict as on "b" as in this method for building the parse table we would have a shift action and for all the input tokens but the second item in the s3 set of items would suggest a reduction. This grammar is thus not LR(0) parseable.

d) The stack contents for the input string w = axb$.



e) This grammar does not have the LL(1) property as FIRST(A) and FIRST(xb) both productions of S will have "x" as a common terminal symbol.

## Problem 2: Syntax-Directed Translation [30 points]

Consider the following syntax-directed definition over the grammar defined by G = ({S, A, Sign}, S, {‘,’, ‘-‘, ‘+’, ‘n’}, P) with P the set of production and the corresponding semantic rules depicted below. There is a special terminal symbol “n” that is lexically matched by any string of one numeric digit and whose value is the numeric value of its decimal representation. For the non-terminal symbols in G we have defined two attributes, *sign* and *value*. The non-terminal A has these two attributes whereas S only has the *value* attribute and Sign only has the *sign* attribute.

```
S    → A Sign      ||   S.val = A.val; A.sign = Sign.sign; print(A.val);
Sign → +           ||   Sign.sign = 1
Sign → -           ||   Sign.sign = 0
A    → n           ||   A.val = value(n)
A    → A1 , n      ||   A1.sign = A.sign;
                          if(A.sign = 1) then
                            A.val = min (A_1.val,value(n));
                          else
                            A.val = max(A_1.val,value(n));
```

For this Syntax-Directed Definition answer to the following questions:

a) [05 pts] Explain the overall operation of this syntax-directed definition and indicate (with a brief explanation) which of the attributes are either synthesized or inherited.

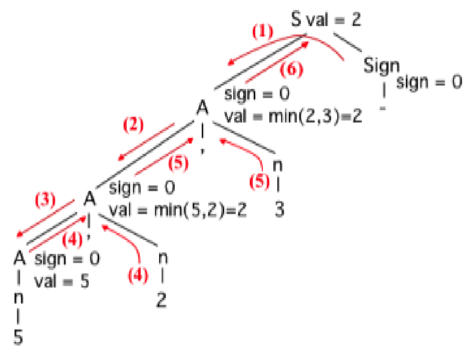
b) [10 pts] Give an attributed parse tree for the source string “5,2,3-“ and evaluate the attributes in the attributed parse tree depicting the order in which the attributes need to be evaluated (if more than one order is possible indicate one.)

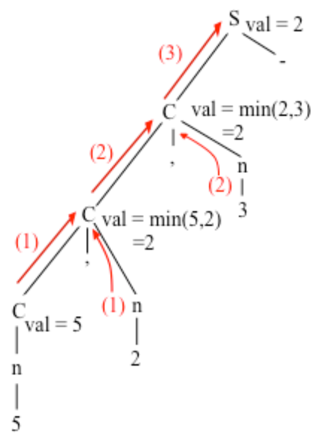
c) [15 pts] Suggest a modified grammar and actions exclusively using synthesized attributes. Explain its basic operation.

**Solution:**

a) This syntax-directed definition computes the maximum or minimum of a sequence of integers depending on the suffix ‘-‘ or ‘+’ respectively. As to the attributes, clearly the sign attribute for the Sign non-terminal is synthesized. The *val* attribute is also synthesized but the *sign* attribute for A is inherited as it flows from “right-to-left” (thus from top to bottom in the parse tree) for a production of A.

b) The parse tree and the corresponding attribute values as well as an evaluation order is as shown below (left).





c) The idea is to decouple at the top the two situations, computing a minimum or maximum based on the value of the Sign non-terminal. To accomplish this we have two very similar non-terminal symbols B and C as shown below, which replace the A terminal symbol. Note also the presence of two productions for S for the two computation functions, *min* and *max* and the absence of the non-terminal Sign. All attributes are now synthesized as shown in the figure above (right).

```
S → B + ||  S.val = B.val; print(B.val)        C → n       ||  C.val = value(n)
S → C -  ||  S.val = C.val; print(C.val)        C → C1 , n  ||  C.val = min(C_1.val,value(n))
B → n    ||  B.val = value(n)
B → B1 , n || B.val = max(B_1.val,value(n))
```

## Problem 3: Intermediate Code Generation [40 points]

Suppose that your language supports the dynamic allocation of single-dimensional vectors or arrays and that the grammar productions are as shown below:

```
vec_expr → id [ expr ]
expr     → id
S        → id = vec_expr
S        → vec_expr = expr
```

where expr is a generic expression with attributes *place* and *code* that indicate, respectively the temporary variable where the result of the evaluation of the expression will reside and the corresponding code. Note that because an array reference might appear on the left-hand-side or on the right-hand-side of an assignment, you need to use a base address and an index to evaluate the expression in the context of an assignment statement.

Regarding the code generation for these array accesses answer the following questions:

a) [15 pts] Define an SDT for the generation of intermediate code that computes the value of vec_expr. Given that the sizes of the array are only known at run-time (after allocation) you need to have access to a dope vector structure. Assume you know the address of this vector given by the variable reference "dyn_vec(id.name)". The first location of this auxiliary array holds the base address of the vector (and index 0) and the second location of this vector holds the size of the single dimension of the vector (at index 1).

b) [15 pts] Show the results of your work on the statement "v = a[i]" indicating the AST with the values of the attributes.

c) [10 pts] Redo a) for the case where you want to include code that checks the bounds of the accesses to the vector. For instance when generating code for "a[i]" you need to check if the index value (in this case "i") is both non-negative and less than the dimension of the vector.

**Solution:**

a) As mentioned in the statement of the problem you need to have **two synthesized** attributed for a vector or array construct. As such as define the attribute *place* as well as the attribute *offset*. An *offset* value is set to zero it means that we have a scalar access. When the *offset* in non-zero means that we have an indexed access. We begin with the assignment statements as follows:

```
S  → id = vec_expr      ||    tk = newtemp();
                              S.code = append(vec_expr.code,
                              gen(tk '=' vec_expr.place[0]),
                              gen(id.place '=' tk '[' vec_expr.offset ']'))

S  → vec_expr = expr    ||    tk = newtemp();
                              S.code = append(expr.code,
                              gen(tk '=' vec_expr.place[0]))
                              if(expr.offset == 0)
                                 S.code=append(S.code,gen(tk '[' vec_expr.offset ']' '='expr.place))
                              else
                                 S.code=append(S.code,gen(tk '[' vec_expr.offset ']' '='expr.place '[' expr.offset ']))
```

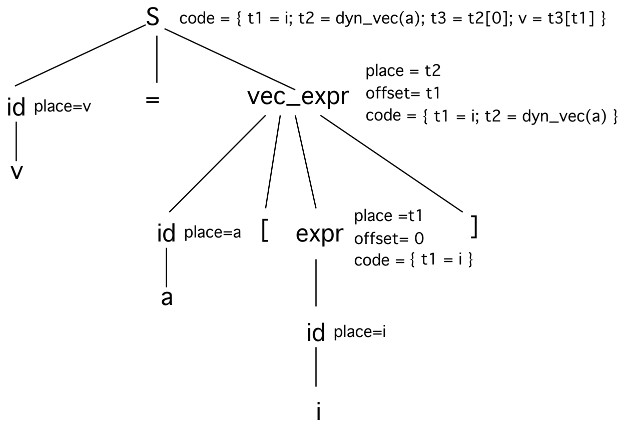
Now the semantic rule to generate code defining the values of the offset and place attributes for the vexpr production:

```
vec_expr → id [ expr ]    ||    tn = newtemp();
                                vec_expr.code = append(expr.code, gen(tn '=' dyn_vec '[' id.place ']' )));
                                vec_expr.place = tn;
                                vec_expr.offset = expr.place;
```

Notice that the semantic rule for the simple expression below needs to be adjusted to reflect the fact that this is a scalar expression:

```
expr   → id      ||    expr.place = newtemp(); expr.code = gen(expr.place '=' id.place); expr.offset = 0;
```

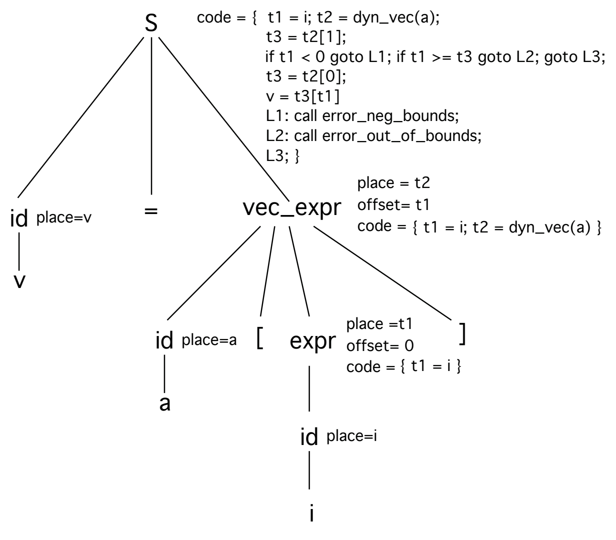
b) For this simple example, v = a[i] we would have the AST shown on the left-hand-side below along with the corresponding generated code at the top S non-terminal symbol's code attribute at the top.


S code = { t1 = i; t2 = dyn_vec(a); t3 = t2[0]; v = t3[t1] }
id place=v
v
=
vec_expr place = t2 offset= t1 code = { t1 = i; t2 = dyn_vec(a) }
id place=a
a
[
expr place =t1 offset= 0 code = { t1 = i }
]
id place=i
i

c) For this question we need to generate code that checks are run time the bounds of the index access value. This is done by generating code that checks that the offset attribute value is both non-negative and less than the maximum value stored at position 1 of the *dope* vector. As done in the previous a) section the base address of the dope vector is saved as the place of the vec_expr (in fact at index 0), so we need to generate code to add 1 to that value and fetch the bound of the array and compare it against the offset value. The semantic rule below does this, which we illustrated in the subsequent figure.

```
S    → id = vec_expr   ll   tk = newtemp();
                            S.code = append(vexpr.code,
                            gen(tk '=' vec_expr.place'['1']'),
                            gen(if vec_expr.offset '<' 0 goto L1),
                            gen(if vec_expr.offset '>=' tk goto L2),
                            gen(tk '=' vec_expr.place[0];),
                            gen(id.place '=' tk '[' vexpr.offset ']'),
                            gen(goto L3)
                            gen(L1: call error_neg_bounds);
                            gen(L2: call error_out_of_bounds);
                            gen(L3:))
```

Here we have generated code that makes calls to some predefined error handling routines for the two cases where the array subscripts is less than zero and more than the maximum allowed index value.


S code = { t1 = i; t2 = dyn_vec(a);
t3 = t2[1];
if t1 < 0 goto L1; if t1 >= t3 goto L2; goto L3;
t3 = t2[0];
v = t3[t1]
L1: call error_neg_bounds;
L2: call error_out_of_bounds;
L3; }
id place=v
v
=
vec_expr place = t2 offset= t1 code = { t1 = i; t2 = dyn_vec(a) }
id place=a
a
[
expr place =t1 offset= 0 code = { t1 = i }
]
id place=i
i