

# Compiler Design

Spring 2014

*Syntax-Directed Translation*

*Sample Exercises and Solutions*

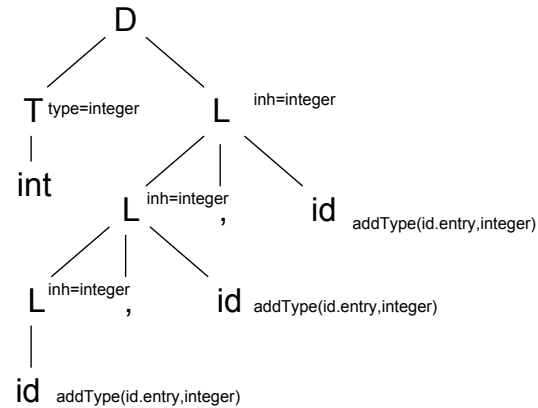
Prof. Pedro C. Diniz

USC / Information Sciences Institute  
4676 Admiralty Way, Suite 1001  
Marina del Rey, California 90292  
pedro@isi.edu

**Problem 1:** Given the Syntax-Directed Definition below construct the annotated parse tree for the input expression: “int a, b, c”.

$D \rightarrow T L$	$L.inh = T.type$
$T \rightarrow int$	$T.type = integer$
$T \rightarrow float$	$T.type = float$
$L \rightarrow L_1, id$	$L_1.inh = L.inh$
	$addType(id.entry, L.inh)$
$L \rightarrow id$	$addType(id.entry, L.inh)$

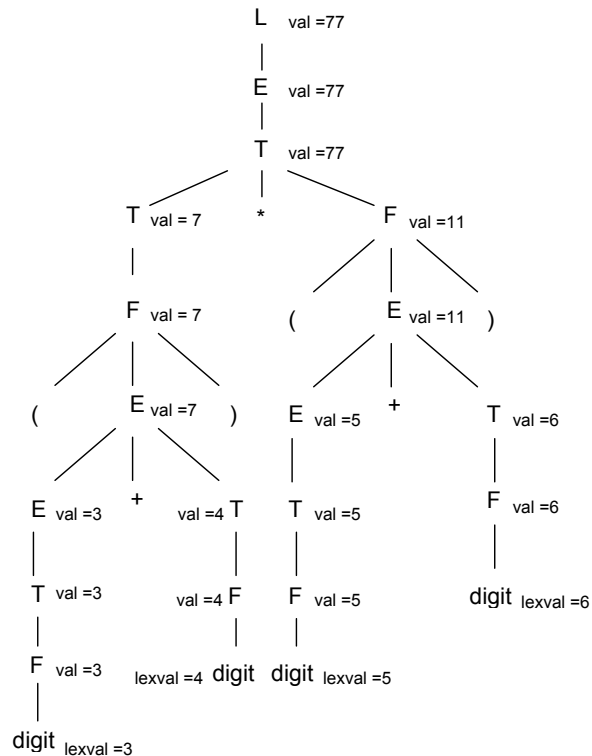
**Solution:**



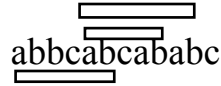
**Problem 2:** Given the Syntax-Directed Definition below with the synthesized attribute *val*, draw the annotated parse tree for the expression (3+4) \* (5+6).

$L \rightarrow E$	$L.val = E.val$
$E \rightarrow T$	$E.val = T.val$
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$T \rightarrow F$	$T.val = F.val$
$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
$F \rightarrow ( E )$	$F.val = E.val$
$F \rightarrow digit$	$F.val = digit.lexval$

**Solution:**



**Problem 3:** Construct a Syntax-Directed Translation scheme that takes strings of a's, b's and c's as input and produces as output the number of substrings in the input string that correspond to the pattern  $a(a|b)^*c+(a|b)^*b$ . For example the translation of the input string "abbcabababc" is "3".


  
 abbcabababc

Your solution should include:

- A context-free grammar that generates all strings of a's, b's and c's
- Semantic attributes for the grammar symbols
- For each production of the grammar a set of rules for evaluation of the semantic attributes
- Justification that your solution is correct.

**Solution:**

- The context-free grammar can be as simple as the one shown below which is essentially a Regular Grammar  $G = (\{a,b,c\}, \{S\}, S, P)$  for all the strings over the alphabet  $\{a,b,c\}$  with  $P$  as the set of productions given below.

$S \rightarrow S a$   
 $S \rightarrow S b$   
 $S \rightarrow S c$   
 $S \rightarrow a$   
 $S \rightarrow b$   
 $S \rightarrow c$

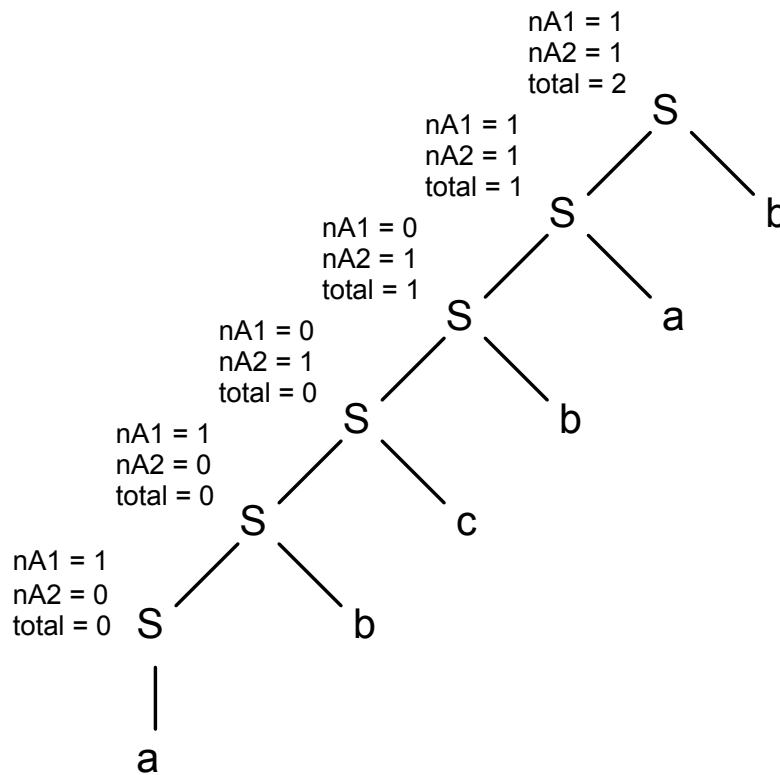
- Given the grammar above any string will be parsed and have a parse tree that is left-skewed, i.e., all the branches of the tree are to the left as the grammar is clearly left-recursive. We define three **synthesized** attributes for the non-terminal symbol  $S$ , namely  $nA1$ ,  $nA2$  and  $total$ . The idea of these attributes is that in the first attribute we will capture the number of a's to the left of a given "c" character, the second attribute,  $nA2$ , the number of a's to the right of a given "c" character and the last attributed,  $total$ , will accumulate the number of substrings.

We need to count the number of a's to the left of a "c" character and to the right of that character so that we can then add the value of  $nA1$  to a running total for each occurrence of a b character to the right of "c" which recording the value of a's to the right of "c" so that when we find a new "c" we copy the value of the "a's" that were to the right of the first "c" and which are now to the left of the second "c".

- c) As such a set of rules is as follows, here written as semantic actions given that their order of evaluation is done using a bottom-up depth-first search traversal.

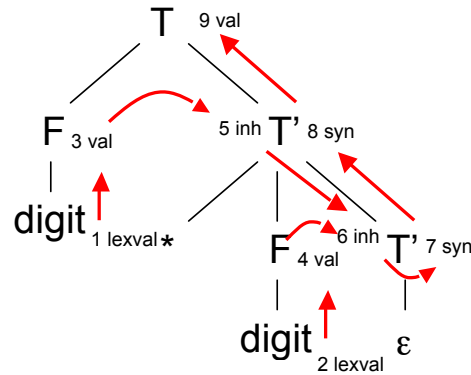
$S_1 \rightarrow S_2 a$	$\{ S_1.nA1 = S_2.nA1 + 1; S_1.nA2 = S_2.nA2; S_1.total = S_2.total; \}$
$S_1 \rightarrow S_2 b$	$\{ S_1.nA1 = S_2.nA1; S_1.nA2 = S_2.nA2; S_1.total = S_2.total + S_2.nA2; \}$
$S_1 \rightarrow S_2 c$	$\{ S_1.nA1 = 0; S_1.nA2 = S_2.nA1; S_1.total = S_2.total; \}$
$S_1 \rightarrow a$	$\{ S_1.nA1 = 1; S_1.nA2 = 0; S_1.total = 0; \}$
$S_1 \rightarrow b$	$\{ S_1.nA1 = 0; S_1.nA2 = 0; S_1.total = 0; \}$
$S_1 \rightarrow c$	$\{ S_1.nA1 = 0; S_1.nA2 = 0; S_1.total = 0; \}$

d)

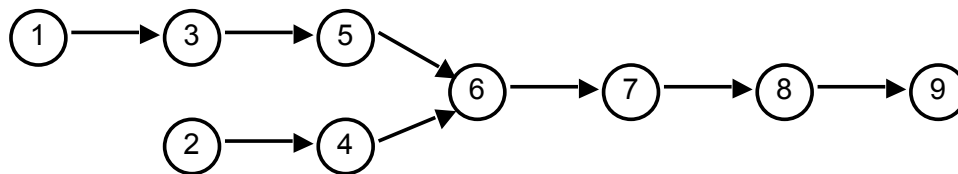


We have two rolling counters for the number of a's one keeping track of the number of a's in the current section of the input string (the sections are delimited by "c" or sequences of "c"s) and the other just saves the number of c's from the previous section. In each section we accumulate the number of a's in the previous section for each occurrence of the "b" characters in the current section. At the end of each section we reset one of the counters and save the other counter for the next section.

**Problem 4:** For the annotated parse tree with the attribute dependences shown below determine the many order of attribute evaluation that respect these dependences. In the parse tree below each attribute is given a specific numeric identifier so that you can structure your answer in terms of a graph with nodes labelled using the same identifiers.



**Solution:** Below you will find the dependence graph in a simplified fashion with the nodes indicating the nodes of the parse tree and the edges indicating precedence of evaluation. It is apparent that the last section of the dependence graph denoted by the nodes  $\{6\ 7\ 8\ 9\}$  needs to be evaluated in sequence. The only flexibility of the order of evaluation is on the two sub-sequences  $\{1\ 3\ 5\}$  and  $\{2\ 4\}$  which can with any interleaving provided the relative order of their nodes is preserved.



Base sequences are  $\{1\ 3\ 5\}$  and  $\{2\ 4\}$  with the suffix  $\{6\ 7\ 8\ 9\}$  being constant as the last nodes of the topological sorting need to remain fixed.

1 3 5 2 4 6 7 8 9  
 1 3 2 5 4 6 7 8 9  
 1 2 3 5 4 6 7 8 9  
 1 3 2 4 5 6 7 8 9  
 1 2 3 4 5 6 7 8 9

1 2 4 3 5 6 7 8 9  
 2 1 3 5 4 6 7 8 9  
 2 1 3 4 5 6 7 8 9  
 2 1 4 3 5 6 7 8 9  
 2 4 1 3 5 6 7 8 9

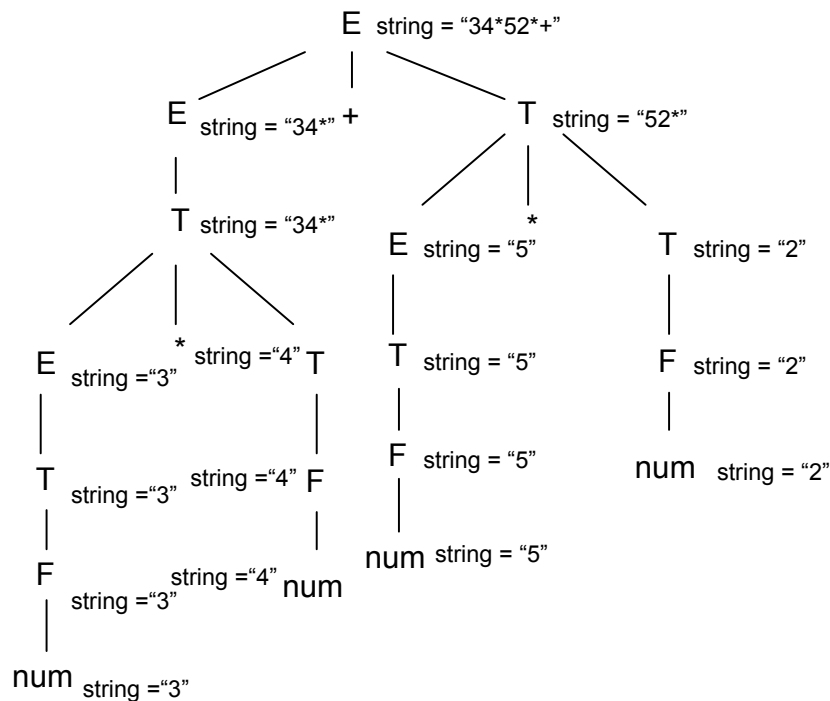
**Problem 5:** Construct a Syntax-Directed Translation scheme that translates arithmetic expressions from infix into postfix notation. Your solution should include the context-free grammar, the semantic attributes for each of the grammar symbols, and semantic rules. Shown the application of your scheme to the input and “3\*4+5\*2”.

**Solution:** In this example we define a synthesized attribute *string* and the string concatenation operator “||”. We also define a simple grammar as depicted below with start symbol *E* and with terminal symbol *num*. This *num* also has a string attribute set by the lexical analyzer as the string with the numeric value representation of *num*.

**Grammar****Semantic Rules**

$E_1 \rightarrow E_2 + T$	$E_1.string = E_2.string \parallel T.string \parallel '+'$
$E_1 \rightarrow T$	$E_1.string = T.string$
$T_1 \rightarrow T_2 * F$	$T_1.string = T_2.string \parallel F.string \parallel '*'$
$T \rightarrow F$	$T.string = F.string$
$F \rightarrow ( E )$	$F.string = E.string$
$F \rightarrow num$	$F.string = num.string$

For the input string “3\*4+5\*2” we would have the annotated parse tree below with the corresponding values for the string attributes.



**Problem 6:** In Pascal a programmer can declare two integer variables a and b with the syntax

```
var a,b: int
```

This declaration might be described with the following grammar:

$$\begin{array}{ll} \text{VarDecl} \rightarrow & \text{var IDList : TypeID} \\ \text{IDList} \rightarrow & \text{IDList , ID} \\ & | \quad \text{ID} \end{array}$$

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.

- Write an attribute grammar that assigns the correct data type to each declared variable.
- Determine an evaluation order of the attributes irrespective of the way the parse tree is constructed.
- Can this translation be performed in a single pass over the parse tree as it is created?

**Solution:**

$$\text{VarDecl} \rightarrow \text{var IDList : TypeID}$$

$$\begin{array}{ll} \text{IDList}_0 & \rightarrow \text{IDList}_1 , \text{ID} \\ & | \quad \text{ID} \end{array}$$

(a,b) As it is described the grammar would require an inherited attribute for passing the type of the identifier to the top of the parse tree where the IDList non-terminal would be rooted. This inherited attribute causes all sort of problems, the one being that it would preclude the evaluation in a single pass if the tree were created using a bottom-up and Left-to-Right parsing approach.

$$\begin{array}{llll} \text{VarDecl} & \rightarrow & \text{var ID List} & \parallel \quad \text{ID.type} = \text{List.type} \\ \\ \text{List}_0 & \rightarrow & , \text{ID List}_1 & \parallel \quad \text{ID.type} = \text{List}_0.\text{type} \\ & & & \parallel \quad \text{List}_1.\text{type} = \text{List}_0.\text{type} \\ & | & : \text{TypeID} & \parallel \quad \text{List}_0.\text{type} = \text{TypeID.value} \end{array}$$

(c) As such the best way would be to change the grammar as indicated above where all the attributes would be synthesized and hence allow the evaluation of the attribute in a single pass using the bottom-up Left-to-Right parsing approach.

**Problem 7:** A language such as C allows for user defined data types and structures with alternative variants using the union construct as depicted in the example below. On the right-hand-side you have a context-free-grammar for the allowed declarations which has as starting symbol a `struct_decl`. This CFG uses the auxiliary non-terminal symbol *identifier* for parsing identifiers in the input. You should assume the identifier as having a single production  $identifier \rightarrow string\_token$  where *string\_token* is a terminal symbol holding a string as its value.

***Please make sure you know what a union in C is before attempting to solve this problem.***

```
typedef struct {
  int a;
  union {
    int b;
    char c;
    double d;
  } uval;
  int e;
} A;
```

```
base_type      → int | double | char
base_type_decl → base_type identifier
field_decl     → base_type_decl
               | union_decl
               | struct_decl
field_list     → field_list field_decl
               | field_decl
               | ε
struct_decl    → typedef struct { field_list } identifier
union_decl     → union { field_list } identifier
```

### Questions:

- When accessing a field of a union the compiler must understand the offset at which each element of the union can be stored relative to the address of the struct/union. Using the context-free grammar depicted above derive an attributive grammar to determine the offset value for each field of a union. Be explicit about the meaning of your attributes and their type. Assume a double data types requires 8 bytes, an integer 4 and a character 1 byte. Notice that unions can be nested and you might want to rewrite the grammar to facilitate the evaluation of the attributes.
- Show the result of your answer in (a) for the example in the figure by indicating the relative offset of each field with respect to the address that represents the entire struct.
- In some cases the target architecture requires that all the elements of a structure be word-aligned, *i.e.*, every field of the struct needs to start at an address that is a multiple of 4 bytes. Outline the modifications to your answer in (a) and (b).



**Solution:** The basic idea of a set of attributes is to have a inherited attributes to track the current start of each field of either a union of a struct and another synthesized attributes to determine what the last address of each field is. The inherited attribute named start, starts from 0 and is propagated downwards towards the leaves of the parse tree and indicates at each level the current starting address of a given field. The synthesized attribute, called end, determines where each field ends with respect to the base address of the struct or union. Because in the struct we need to add up the sizes of the fields and in the union we need to compute the maximum size of each field we have another inherited attributes, called mode, with the two possible symbolic values of AND and OR.

In terms of the attribute rules we have some simple rules to copy the inherited attributes downwards and the synthesized attributes upwards as follows:

```
field_decl -> union_decl { union_decl.start = field_decl.start; field_decl.end = union_decl.end; }
field_decl -> struct_decl { struct_decl.start = field_decl.start; field_decl.end = struct_decl.end; }
field_decl -> base_decl { base_decl.start = field_decl.start; field_decl.end = base_decl.end; }
```

At the bottom of the parse tree we need to make the assignment of the end end attributes given an input start attributes as follows:

```
base_decl -> int identifier { base_decl.end = base_decl.start + 4; }
base_decl -> char identifier { base_decl.end = base_decl.start + 1; }
base_decl -> double identifier { base_decl.end = base_decl.start + 8; }
```

Finally we have the rules that do the propagation of the attributes in the manner that takes into account the fact that we are inside a union or a struct. The first set of rules is a simple copy of the values between the field\_decl\_list and a single field\_decl:

```
field_decl_list -> field_decl { field_decl.start = field_decl_list.start; field_decl_list.end = field_decl.end; }
```

The real work occurs during the recursive rule:

```
field_decl_list_0 -> field_decl_list_1 field_decl {
  field_decl_list_1.mode = field_decl_0.mode;
  field_decl_list_1.start = field_decl_list_0.start;
  if (field_decl_list_0.mode == AND) then
    field_decl.start = field_decl_list_1.end;
    field_decl_list_0.end = field_decl.end;
  else
    field_decl_list_1.start = field_decl_list_0.start;
    field_decl_list_0.end = MAX(field_decl_list_1.end, field_decl.end);
  end if
}
```

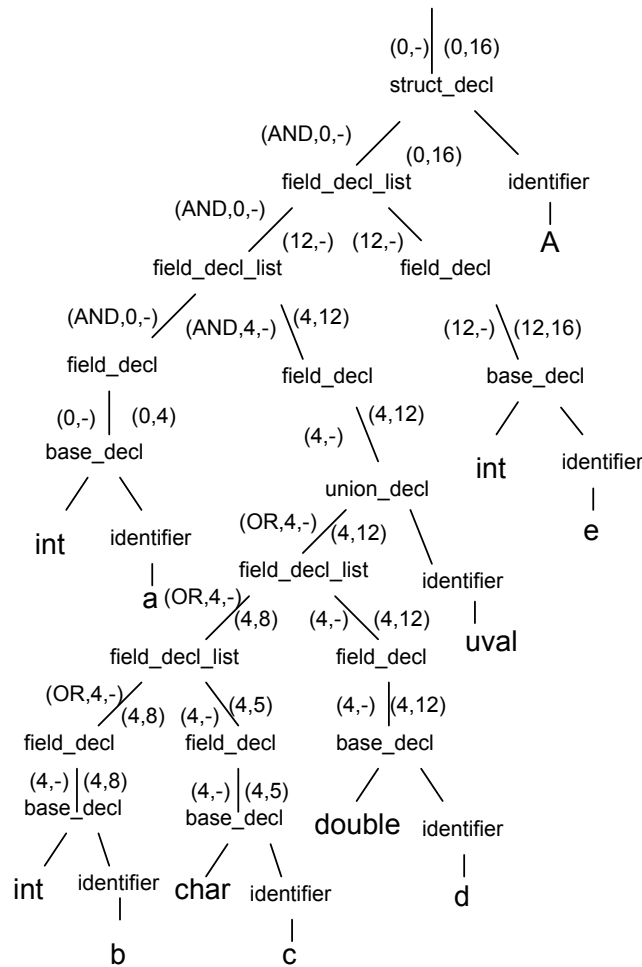
At last we need the rules that set at a given stage the mode attributes:

```
struct_decl -> struct { field_decl_list } identifier {
  field_decl_list.mode = AND;
  field_decl_list.start = struct_decl.start;
  struct_decl.end = field_decl_list.end;
}

union_decl -> union { field_decl_list } identifier {
  field_decl_list.mode = OR;
  field_decl_list.start = union_decl.start;
  union_decl.start = field_decl_list.end;
}
```

- (d) Show the result of your answer in (a) for the example in the figure by indicating the relative offset of each field with respect to the address that represents the entire struct.

In the figure below we illustrate the flow of the attributes downwards (to the left of each edge) and upwards (to the right of each edge) for the parse tree corresponding to the example in the text. In the cases where the mode attribute is not defined we omit it and represent the start and end attributes only.



- (e) In some cases the target architecture requires that all the elements of a structure be word-aligned, i.e., every field of the struct needs to start at an address that is a multiple of 4 bytes. Outline the modifications to your answer in (a) and (b).

In this case the rules for the base declaration need to take into account the alignment of the inherited start attribute and can be recast as follows using the auxiliary function `align_address` that returns the next word-aligned address corresponding to its input argument.

```

base_decl -> int identifier      { base_decl.end = align_address(base_decl.start) + 4; }
base_decl -> char identifier     { base_decl.end = align_address(base_decl.start) + 1; }
base_decl -> double identifier   { base_decl.end = align_address(base_decl.start) + 8; }

```

**Problem 8:** Consider the following grammar  $G$  for expressions and lists of statements ( $\text{StatList}$ ) using assignment statements ( $\text{Assign}$ ) and basic expressions ( $\text{Expr}$ ) using the productions presented below.

- (0)  $\text{StatList} \rightarrow \text{Stat} ; \text{StatList}$
- (1)  $\text{StatList} \rightarrow \text{Stat}$
- (2)  $\text{Stat} \rightarrow \text{Assign}$
- (3)  $\text{Expr} \rightarrow \text{Expr} + \text{Expr}$
- (4)  $\text{Expr} \rightarrow \text{int}$
- (5)  $\text{Expr} \rightarrow \text{id}$
- (6)  $\text{Assign} \rightarrow \text{Expr} = \text{Expr}$
- (7)  $\text{Assign} \rightarrow \text{Expr} += \text{Expr}$

Using a stack-based machine write a syntax-directed definition to generate code for  $\text{StatList}$ ,  $\text{Assign}$  and  $\text{Expr}$ . Argue that your translation is correct.

In this stack based-machine you need to push the operands of the expressions first onto the stack and then execute the operations with the topmost element(s) of the stack. For example the input “ $a = b + 5$ ” where the identifiers “ $a$ ” and “ $b$ ” have already been defined previously, would result in the following generated code:

```
push "b"
push 5
add
top "a"
```

Here the instructions “push” simply put the value of their argument on top of the stack. The instruction “add” adds the two topmost elements of the stack and replaces them with the numeric value corresponding to the addition of the numeric values at the top of the stack before the instruction executes. The instruction “top a” copies the top of the stack and assigns the value to the variable “ $a$ ”. The value of “ $a$ ” remains at the top of the stack as it can be used as the argument for another operand.

**Solution:** A possible solution is to have only synthesized attributes such as “str” for identifiers and “val” for integers. The non-terminal  $\text{Expr}$  also has a synthesized attribute that holds the variable name when defined by an identifier. When defined by an integer this attribute value is nil. It is assumed that the input program is well formed in the sense that there are no statements of the form “ $3 = 1 + 1$ ”, which have an integer expression on the LHS of the assignment.

A possible syntax-directed translation definition using the YACC-like assignment of non-terminal symbols in a production is shown below where is noted the absence of checks for production 7.

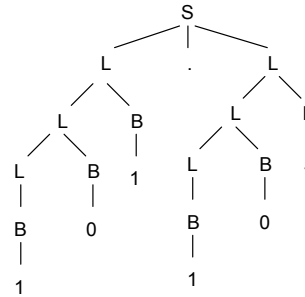
```
(0) StatList → Stat ; StatList    { }
(1) StatList → Stat              { }
(2) Stat → Assign                 { }
(3) Expr → Expr + Expr           { emit("add"); }
(4) Expr → int                   { emit("push int.val"); $$val = nil; }
(5) Expr → id                     { emit("push id.str"); $$val = id.str; }
(6) Assign → Expr = Expr         { emit("top $1.val"); }
(7) Assign → Expr += Expr { emit("add"); emit("push $1.val"); emit("top $1.val"); /*needs to check LHS
*/ }
```

**Problem 9:** For the grammar below design an L-attributed SDD to compute  $S.val$ , the decimal value of an input string in binary. For example the translation of the string 101.101 (whose parse tree is also shown below) should be the decimal number 5.625. Hint: Use an inherited attribute  $L.side$  that tells which side of the decimal point a given bit is on. For all symbols specify their attribute, if they are inherited or synthesized and the various semantic rules. Also show your work for the parse tree example given by indication the values of the symbol attributes.

### Grammar

$S \rightarrow L . L \mid L$   
 $L \rightarrow L B \mid B$   
 $B \rightarrow 0 \mid 1$

### Parse Tree



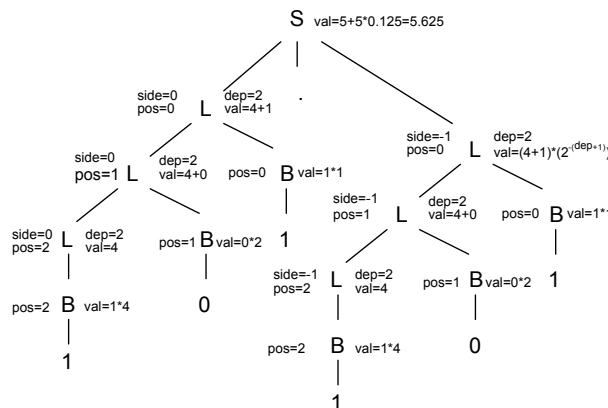
**Solution:** One possible solution is to define 4 attributes, 2 inherited and 2 synthesized, respectively,  $side$  and  $pos$  and the two synthesized attributes as  $depth$  and  $value$ . The  $depth$  attribute is to allow the RHS of the binary string to compute the length. When computing the final value the value on the RHS of the decimal point is simply shift to the right (i.e., divided by  $2^{-(depth+1)}$ ) of the length of the RHS string of the binary representation. The LHS is done by computing at each instance of  $L$  the value of the  $B$  symbol by using the value of position. The position is thus incremented at each stage downwards. The semantic rules for each production are given below along with the annotated parse tree for the input string “101.101”.

### Productions

$S \rightarrow L_1 . L_2$   
 $S \rightarrow L$   
 $L \rightarrow L_1 B$   
 $L \rightarrow B$   
 $B \rightarrow 0$   
 $B \rightarrow 1$

### Semantic Rules

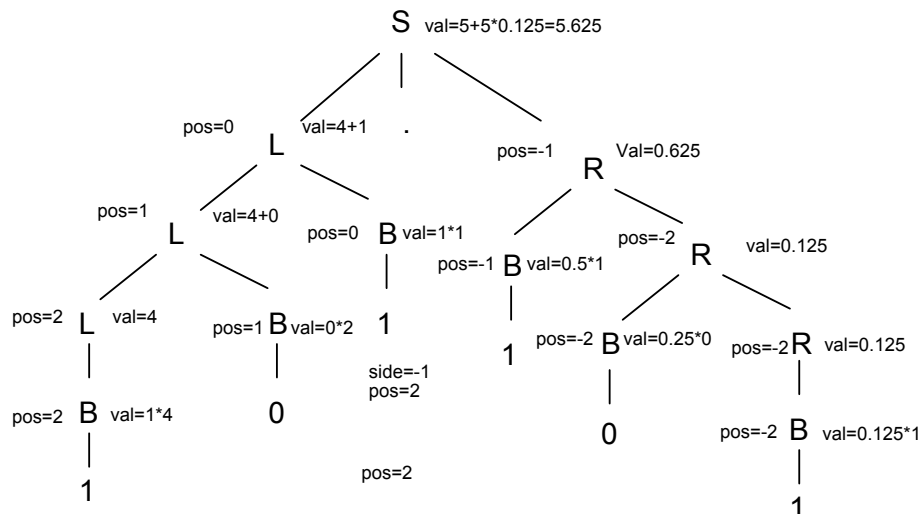
$L_1.side = 0; L_2.side = -1; L_1.pos = 0; L_2.pos = 0; S.val = L_1.val + (L_2.val * 2^{-(L_2.depth+1)});$   
 $L_1.side = 0; L_1.pos = 0; S.val = L_1.val;$   
 $L_1.side = L.side; L_1.pos = L.pos + 1; L.depth = L_1.depth; B.pos = L.pos; L.val = L_1.val + B.val;$   
 $L.depth = L.pos; B.pos = L.pos; L.val = B.val;$   
 $B.val = 0;$   
 $B.val = 1 * 2^{B.pos};$



An alternative, and much more elegant solution would change the grammar to allow the decimal part of the binary string to be right recursive and thus naturally “fall” to the right with a growing weight associated with each non-terminal B is the correct one in the evaluation of the corresponding bit.

We do not require the side nor the depth attribute, just the inherited pos attribute and the synthesized val attribute. Below we present the revised set of semantic actions and the example of the attributes for the input string “101.101”.

Productions	Semantic Rules
$S \rightarrow L . R$	$L.pos = 0; R.pos = -1; S.val = L.val + R.val$
$S \rightarrow L$	$L.pos = 0; S.val = L.val;$
$L \rightarrow L_1 B$	$L_1.pos = L.pos + 1; B.pos = L.pos; L.val = L_1.val + B.val;$
$L \rightarrow B$	$B.pos = L.pos; L.val = B.val;$
$R \rightarrow R_1 B$	$R_1.pos = R.pos - 1; B.pos = R.pos; L.val = L_1.val + B.val;$
$R \rightarrow B$	$B.pos = R.pos; L.val = B.val;$
$B \rightarrow 0$	$B.val = 0;$
$B \rightarrow 1$	$B.val = 1 * 2^{B.pos};$



**Problem 10:** In class we described a SDT translation scheme for array expressions using row-major organization. In this exercise you are asked to redo the SDT scheme for a column-major organization of the array data. Review the lectures note to understand the layout of a 2-dimensional array in column-major format. Specifically perform the following:

- Indicate the attributes of your SDT scheme along with the auxiliary functions you are using.
- Define a grammar and the semantic actions for this translation.
- Show the annotated parse tree and generated code for the assignment  $x = A[y, z]$  under the assumption that A is 10 x 20 array with  $low_1 = low_2 = 1$  and  $sizeof(baseType(A)) = sizeof(int) = 4$  bytes

*Hint:* You might have to modify the grammar so that instead of being left recursive is right recursive. Below is the grammar used for the row-major organization.

$L \rightarrow \text{Elist } ]$   
 $\text{Elist} \rightarrow \text{Elist}_1 , \text{E}$   
 $\text{Elist} \rightarrow \text{id } [ \text{E}$   
 $\text{E} \rightarrow \text{L}$   
 $\text{S} \rightarrow \text{L} = \text{E}$   
 $\text{L} \rightarrow \text{id}$

**Solution:**

- a.),b.) The basic idea is the same as in the row-major organization but rather than counting the dimensions from the left to the right we would like to count and accumulate the dimension sizes from the right to the left. This way the right-most index of the array indices is the last dimension of the rows and should be multiplied by the first dimension, in this case the size of each row. As such the revised grammar that is right-recursive would work better. Below is such a grammar to which we are associating the same **synthesized** attributed as in the row-major formulation, namely, *place*, *offset* and *ndim* as described in class.

$\text{S} \rightarrow \text{L} = \text{E}$   
 $\text{E} \rightarrow \text{L}$   
 $\text{L} \rightarrow \text{id}$   
 $\text{L} \rightarrow \text{id } [ \text{Elist}$   
 $\text{Elist} \rightarrow \text{E } ]$   
 $\text{Elist} \rightarrow \text{E}, \text{Elist}_1$

We also use the same auxiliary function such as `numElemDim(A, dim)` and `constTerm(Array)` as described in class. Given this grammar we define the semantic actions as described below:

```

Elist → E ]      {
                  Elist.place = E.place
                  Elist.code = E.code;
                  Elist.ndim = 1;
                  }

Elist → E , Elist1 {
                  t = newtemp();
                  m = Elist1.ndim + 1;
                  Elist.ndim = m;
                  code1 = gen(t = Elist1.place * numElemDim(m));
                  code2 = gen(t = t + E.place);
                  Elist.place = t;
                  Elist.ndim = m;
                  Elist.code = append(Elist1.code, E.code, code1, code2);
                  }

L → id [ Elist   {
                  L.place = newtemp();
                  L.offset = newtemp();
                  code1 = gen(L.place '=' constTerm(id));
                  code2 = gen(L.offset '=' Elist.place * sizeof(baseType(id)));
                  L.code = append(Elist.code, code1, code2);
                  }

E → L           {
                  if (L.offset = NULL) then {

```

```

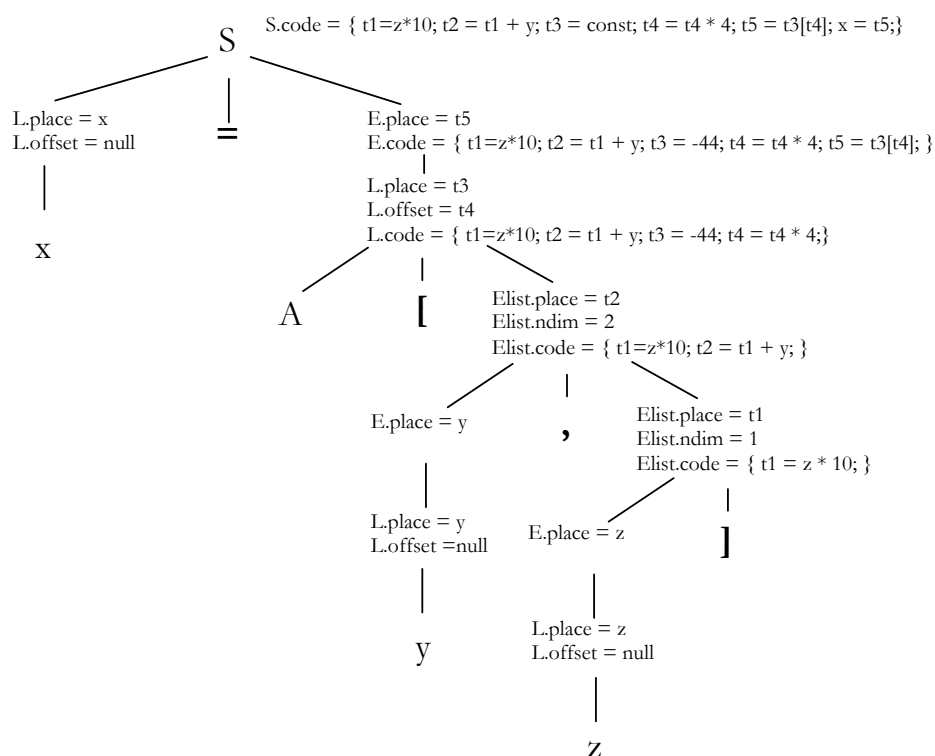
        E.place = L.place;
    } else {
        E.place = newtemp;
        E.code = gen(E.place = L.place[L.offset]);
    }
}

S → L = E    {
    if L.offset = NULL then
        E.code = gen(L.place = E.place);
    else
        S.code = append(E.code, gen(L.place[L.offset] = E.place));
}

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

```

- c) For the assignment example  $x = A[y, z]$  we have the annotated parse tree shown below under the assumption that the base address for the array  $A$  is 0.



**Problem 10:** In this problem you are asked to develop a SDD scheme for a simplified register allocation for the computation of expressions in assignment statements for a block of statements. You need to use the grammar provided and develop the set of attributes to be associated with each of the non-terminal and terminal symbols of the provided grammar. We can assume the input is a syntactically correct program building a well-formed abstract syntax tree (AST).

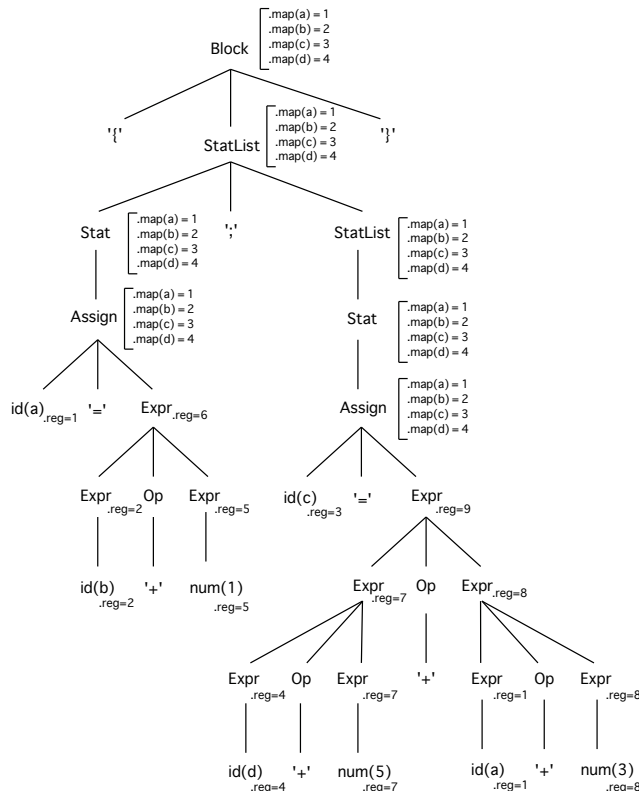
The output of the SDD scheme is an indication for each expression and sub-expression, which of the N registers will hold the results of the corresponding computation at run-time, say `reg = k`. Note that you are not being asked to generate code - that would be a secondary code generation phase. The assignment of registers to each of the variables is done via an attribute associated with the terminal **id** symbol, also named `reg` but whose lifetime spans the entire block of statements and are thus constant throughout the execution of the block of statements. You need to keep track of this mapping throughout your SDD, i.e., if there are `k` variables the temporary registers will begin at register number `k+1`. In this intermediate allocation of registers you are not supposed to minimize the number of uses temporaries, but in the end you need to determine the number of temporary registers your code block is using. The example below illustrates an example of the results of the SDD you need to develop for a sample input code alongside the corresponding AST.

For the specific grammar  $G$  below (left) answer the following:

- Define the set of attributes (synthesized and inherited) to define for each expression and sub-expression which of the  $N$  available register should be used to compute the sub-expression value. Assume that for identifiers the AST has been previously decorated with an attributed named register indicating the specific register holding the corresponding variable's value.
- Show your work for the block of statements whose AST is shown below (right).
- Can your solution be computed in a single pass of the AST? Why or why not?

CFG  $G = (\text{Block}, \{\text{Block}, \text{StatList}, \text{Stat}, \text{Assign}, \text{Expr}, \text{Op}\}, \{\{'\}', '\}', '\,', '=\', '+\', '-\', \text{id}, \text{num}\}, P)$  where the set of productions  $P$  is as shown below:

- (1) Block  $\rightarrow \{ ' \text{ StatList } ' \} \$$
- (2) StatList  $\rightarrow \text{Stat } ' \text{ StatList }'$
- (3) StatList  $\rightarrow \text{Stat}$
- (4) StatList  $\rightarrow \epsilon$
- (5) Stat  $\rightarrow \text{Assign}$
- (6) Assign  $\rightarrow \text{id } '=' \text{ Expr}$
- (7) Expr  $\rightarrow \text{Expr Op Expr}$
- (8) Expr  $\rightarrow '(' \text{ Expr } ')'$
- (9) Expr  $\rightarrow \text{id}$
- (10) Expr  $\rightarrow \text{num}$
- (11) Op  $\rightarrow '+'$
- (12) Op  $\rightarrow '-'$



**Solution:**

- a) There are many possible solutions to this SDT scheme. A simple solution involves the use of inherited and synthesized attributes. An inherited attribute consists in the mapping of the identifiers to the 'fixed' registers and has to be propagated downwards the tree so that when you make the assignment of register to expression you can determine what if any additional register is required. Also, inherited is the current starting assignment value of the registers which needs



to be propagated downwards and then to the sibling nodes in a binary expression or in a statement list. Other attributes are synthesized and correspond to the register attribute use to propagate upwards when you are allocating registers to expressions.

As there are no requirements to reuse and thus minimize the number of registers the semantic rule just checks the indices of the registers uses and assign an additional register to hold the result of the current expression node. As such we are going to have two register tracking attributes, one inherited named 'reg\_in' and another synthesized named 'reg\_out' to be used throughout all the non-terminal symbols of the grammar. For the Expr non-terminals and both the **id** and **num** terminals we have an attributed, named 'reg' that indicates the register identifier that holds the corresponding value. This 'reg' attribute is not defined for the other non-terminal symbols, such as Block, Assing, Stat, StatList or Op. In addition we have a simple inherited 'map' attribute that is strictly 'read-only' to be propagated from the 'Block' non-terminal symbol downwards to the 'Expr' non-terminal. Associated with this 'map' attribute we also have a 'MapId' function that given the strings of an identifier returns the corresponding register mapping (assumed to always be valid). In addition we make use of a 'NumbersId' function that given a mapping computes the total number of identifiers currently mapped to the 'fix' set of registers. The solution below makes explicit all the attributes with the invariably long list of 'copy' rules.

Block $\rightarrow$ '{' StatList '}' \$	StatList.reg_in = NumberIds(Block.map); StatList.map = Block.map
StatList <sub>0</sub> $\rightarrow$ Stat ';' StatList <sub>1</sub>	Stat.reg_in = StatList <sub>0</sub> .reg_in; StatList <sub>1</sub> .reg_in = Stat.reg_out; StatList <sub>0</sub> .reg_out = StatList <sub>1</sub> .reg_out Stat.map = StatList <sub>0</sub> .map; StatList <sub>1</sub> .map = StatList <sub>0</sub> .map;
StatList $\rightarrow$ Stat	Stat.reg_in = StatList.reg_in; StatList.reg_out = Stat.reg_out; Stat.map = StatList.map;
StatList $\rightarrow$ $\epsilon$	StatList.reg_out = StatList.reg_in;
Stat $\rightarrow$ Assign	Assign.reg_in = Stat.reg_in; Stat.reg_out = Assign.reg_out Assign.map = Stat.map;
Assign $\rightarrow$ <b>id</b> '=' Expr	Assign.reg_out = Expr.reg_out; Expr.reg_in = Assign.reg_in; Expr.map = Assign.map;
Expr <sub>0</sub> $\rightarrow$ Expr <sub>1</sub> Op Expr <sub>2</sub>	Expr <sub>1</sub> .reg_in = Expr <sub>0</sub> .reg_in; Expr <sub>2</sub> .reg_in = Expr <sub>1</sub> .reg_out; Expr <sub>0</sub> .reg_out = Expr <sub>2</sub> .reg_out + 1; Expr <sub>1</sub> .map = Expr <sub>0</sub> .map; Expr <sub>2</sub> .map = Expr <sub>0</sub> .map;
Expr <sub>0</sub> $\rightarrow$ '(' Expr <sub>1</sub> ')'	Expr <sub>1</sub> .reg_in = Expr <sub>0</sub> .reg_in; Expr <sub>0</sub> .reg_out = Expr <sub>1</sub> .reg_out; Expr <sub>1</sub> .map = Expr <sub>0</sub> .map;
Expr $\rightarrow$ <b>id</b>	Expr.reg_out = Expr.reg_in; id.reg = MapId(Expr.map(id.text));
Expr $\rightarrow$ <b>num</b>	Expr.reg_out = Expr.reg_in + 1; num.reg = Expr.reg_in + 1;

b) The figure below depicts the results of the use of the semantics rule for the solution described in a).

