

Compiler Design

Spring 2022

Attribute Grammars

Sample Exercises and Solutions

Prof. Pedro C. Diniz e Prof. Mário Florido

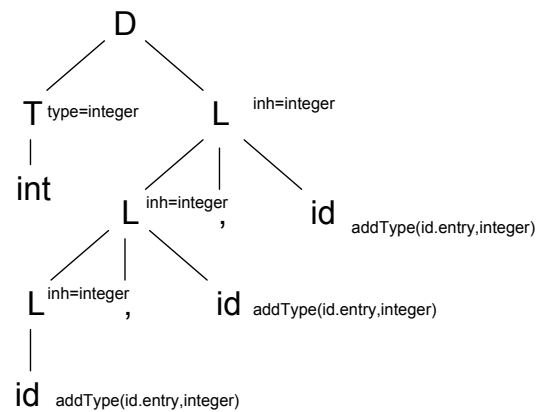
FEUP-FCUP

pedro@fe.up.pt, amflorid@fc.up.pt

Problem 1

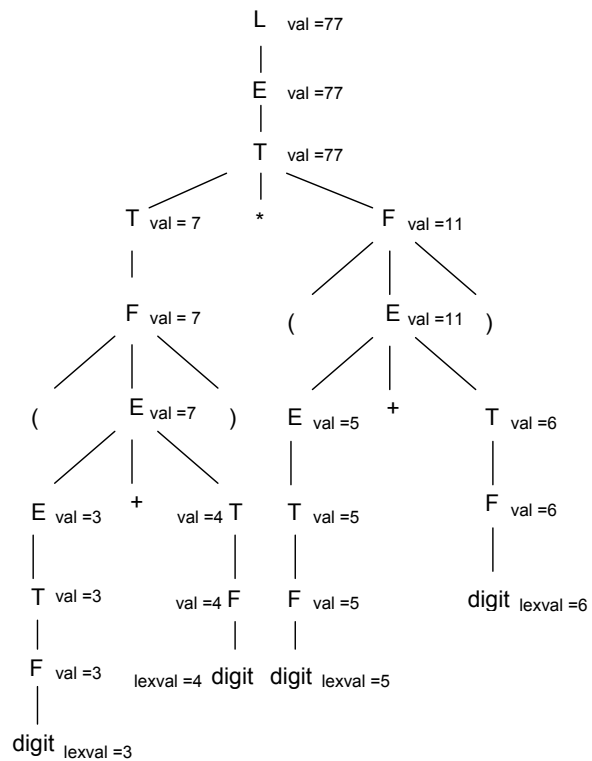
Given the attribute grammar below construct the annotated parse tree for the input expression: “int a, b, c”.

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

Solution:**Problem 2**

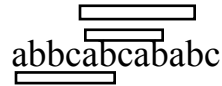
Given the attribute grammar below with the synthesized attribute *val*, draw the annotated parse tree for the expression $(3+4) * (5+6)$.

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

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 "abbcabcbabc" is "3".



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.

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

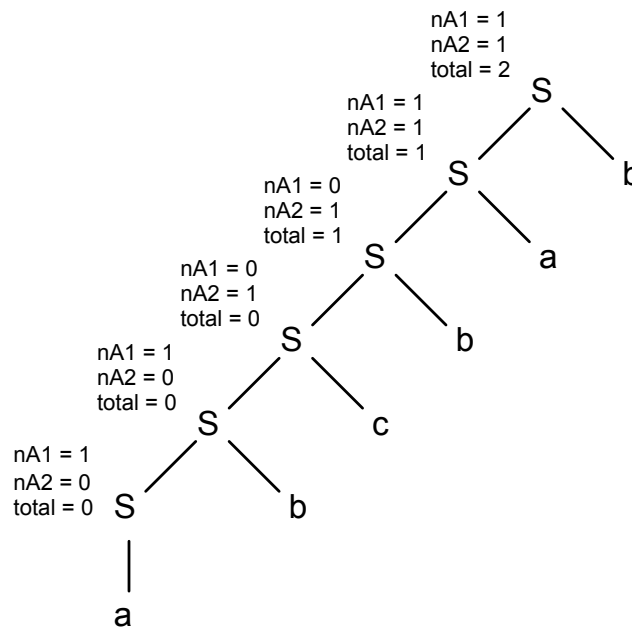
- Given the grammar above any string will be parsed and have a parse tree that is left-skewed, i.e., all the branches of the tree are to the left as the grammar is clearly left-recursive. We define three **synthesized** attributes for the non-terminal symbol S , namely $nA1$, $nA2$ and $total$. The idea 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.

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

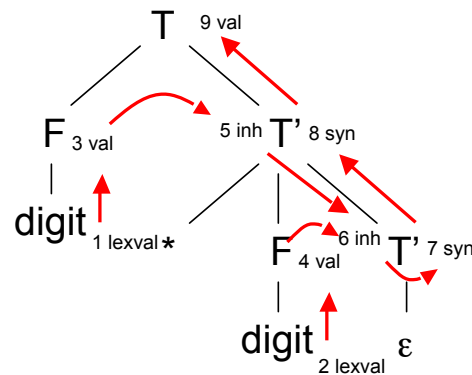
d)



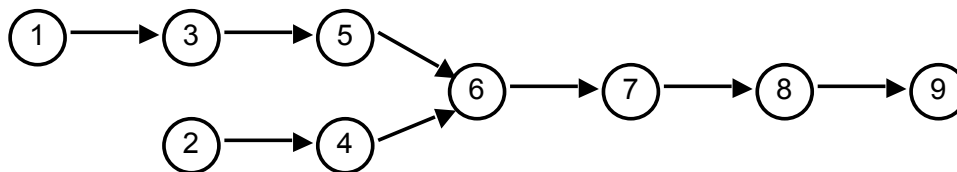
We have two rolling counters for the number of a's one keeping track of the number of a's in the current section of the input string (the sections are delimited by "c" or sequences of "c"s) and the other just saves the number of c's from the previous section. In each section we accumulate the number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

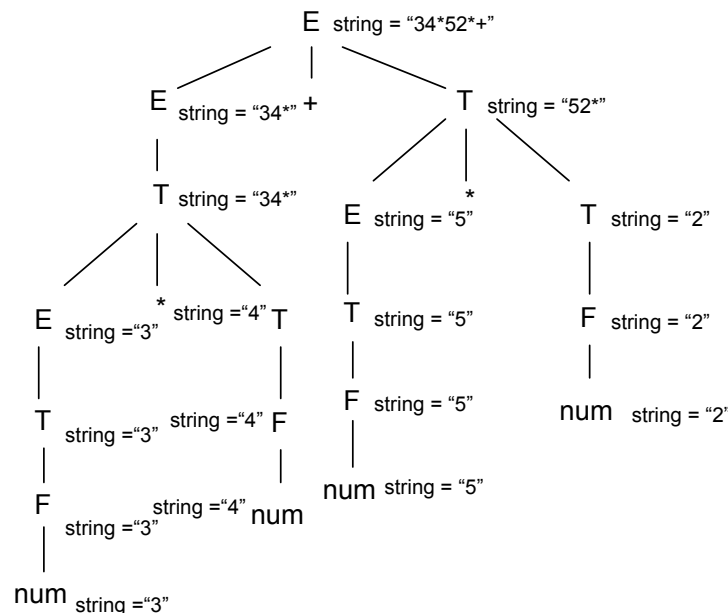
Grammar

$E_1 \rightarrow E_2 + T$
 $E_1 \rightarrow T$
 $T_1 \rightarrow T_2 * F$
 $T \rightarrow F$
 $F \rightarrow (E)$
 $F \rightarrow \text{num}$

Semantic Rules

$E_1.\text{string} = E_1.\text{string} \parallel T.\text{string} \parallel '+'$
 $E_1.\text{string} = T.\text{string}$
 $T_1.\text{string} = T_2.\text{string} \parallel F.\text{string} \parallel '*'$
 $T.\text{string} = F.\text{string}$
 $F.\text{string} = E.\text{string}$
 $F.\text{string} = \text{num}.\text{string}$

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:

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

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:

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

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

(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 attribute, 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
}
```

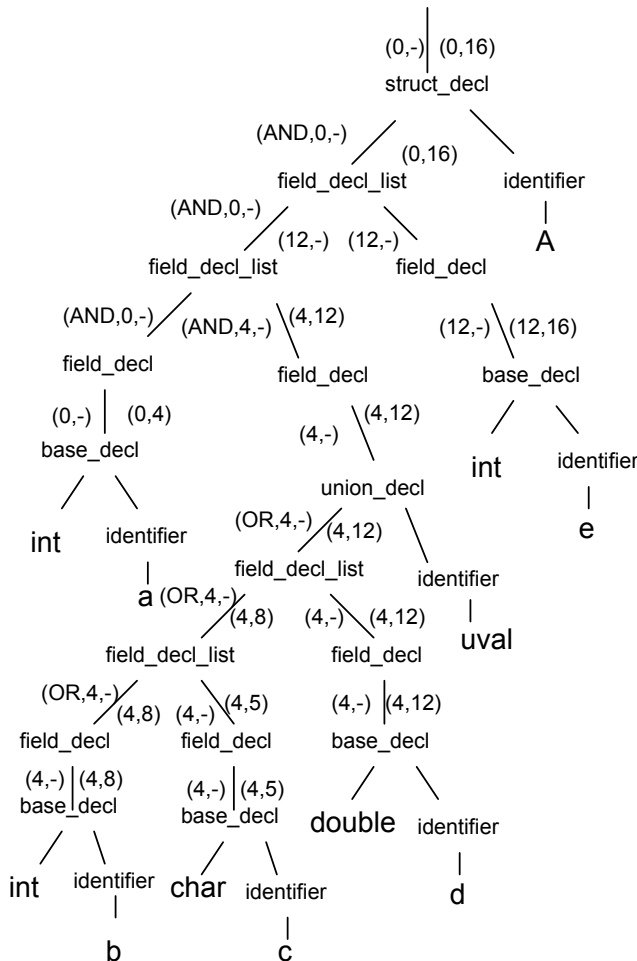
At last we need the rules that set at a given stage the mode attributes:

```
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; }

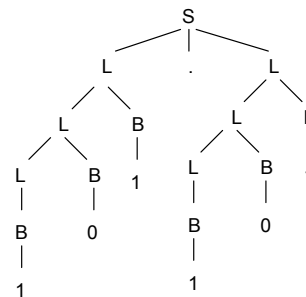
```


Problem 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 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:**

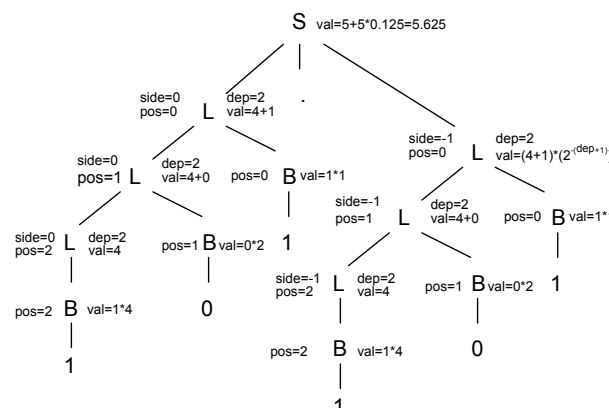
One possible solution is to define 4 attributes, 2 inherited and 2 synthesized, respectively, *side* and *pos* and the two synthesized attributes as *depth* and *value*. The *depth* attribute is to allow the RHS of the binary string to compute the length. When computing the final value the value on the 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

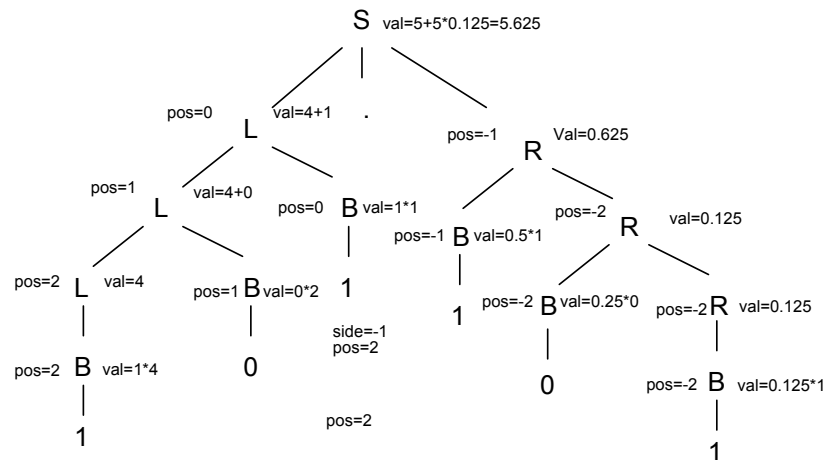
$L_1.side = 0; L_2.side = -1; L_1.pos = 0; L_2.pos = 0; S.val = L_1.val + (L_2.val * 2^{-(L_2.depth+1)});$
 $L_1.side = 0; L_1.pos = 0; S.val = L_1.val;$
 $L_1.side = L.side; L_1.pos = L.pos + 1; L.depth = L_1.depth; B.pos = L.pos; L.val = L_1.val + B.val;$
 $L.depth = L.pos; B.pos = L.pos; L.val = B.val;$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



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 11

In this problem you are asked to develop an attributive grammar based on an existing CFG to perform a verification of a specific string format as they are being parsed. The strings are composed of decimal digits aggregated in groups of exactly 3 digits per group. The number of such groups is not specified but they are separated by ‘.’. At the end of the string there is a numeric 2-digit suffix indicating the check-sum of the entire string as the summation of all the numbers module 100. As an example, the string ‘101:001-02’ is a correct string whereas the same string with termination other than ‘02’ is not. Strings are separated by a ‘;’ character.

The grammar you need to augment with attributes and rules is as follows:

- (1) `string_list` \rightarrow `string` ‘;’ `string_list`
- (2) `string_list` \rightarrow `string`
- (3) `string` \rightarrow `triplet_list` ‘-’ `check`
- (4) `triplet_list` \rightarrow `triplet` ‘:’ `triplet_list`
- (5) `trippel_list` \rightarrow `triplet`
- (6) `triplet` \rightarrow **digit digit digit**
- (7) `check` \rightarrow **digit digit**

The rules of the attribute grammar for the string should print out an error message whenever the corresponding string inputs has an invalid checksum. Show your work for a sample correct and incorrect string.

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 `id_list` as well as the attribute `value` for the terminal symbol `type`. The grammar productions and corresponding rules are as shown below where we have ignored the `decl` non-terminal symbols as it plays no rule in the evaluation process.

```

decl      →      'var' id_list ':' type ';'      || id_list.type = type.value
id_list0  →      id_list1 ';' ID                  || id_list1.type = id_list0.type; id.type =
id_list0.type;
           →      id                             || id.type = id_list0.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 `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.

```

decl      →      'var' id id_list ||      id.type = id_ist.type
id_list0  →      ';' id id_list1 ||      id.type = id_list1.type; id_list0.type = id_list1.type
           →      ':' type ';'    ||      id_ist0.type = type.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.

```
decl      →  'var' id_list ':' type ';'
id_list   →  ID id_list'
id_list'  →  ';' id id_ist'
```

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 `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 `var` is shifted and removed from the stack, leaving `id_list` exposed. Once that is expanded we can 'access' the value of `type` two position down in the stack and set the attribute of `id_list'` as this is pushed onto the stack. With `id` now on top of the stack, the attribute of `id_list'` is just underneath it. The situation repeats itself for the other production of `id_list'`. The inherited attributes can be evaluated and set on the correct symbols once they are passed onto the stack.