Syntax-Directed Translation – In-Class Assignment

1. Read carefully the following definitions

S*yntax-directed definition* (SDD) is a context-free grammar together with attributes and rules. Attributes are associated with grammar symbols and rules are associated with productions.

Example: for an infix to postfix translator, we might add to the production $T \rightarrow T_1 + F$ the rule T.code = T_1.code || F.code || '+'

There are two kinds of attributes for nonterminals:

A *synthesized attribute* for a nonterminal *A* at a parse-tree node N is defined by a semantic rule associated with the production at N. Note that the production must have A as its head. A synthesized attribute at node N is defined only in terms of attribute values at the children of N and at N itself.

An *inherited attribute* for a nonterminal B at a parse-tree node N is defined by a semantic rule associated with the production at the parent of N. Note that the production must have B as a symbol in its body. An inherited attribute at node N is defined only in terms of attribute values at N's parent, N itself, and N's siblings.

Terminals can have synthesized attributes, but not inherited attributes.

To visualize the translation specified by an SDD, it helps to work with parse trees. A parse tree, showing the value(s) of its attribute(s) is called an *annotated parse tree.*

Now, for the below SDD, give the annotated parse trees for the following expression: 2 * 8.

	PRODUCTION	SEMANTIC RULES
1)	$T \rightarrow F\ T'$	$T'.inh = F.val$ $T.val = T'.syn$
2)	$T' \rightarrow *\ F\ T'_1$	$T'_1.inh = T'.inh \times F.val$ $T'.syn = T'_1.syn$
3)	$T' \rightarrow \epsilon$	$T'.syn = T'.inh$
4)	$F \rightarrow \textbf{digit}$	$F.val = \textbf{digit}.lexval$

2. Read carefully the following definitions

A dependency graph depicts the flow of information among the attribute instances in a particular parse tree; an edge from one attribute instance to another means that the value of the first is needed to compute the second.

A SDD is *S-attributed* if every attribute is synthesized. S-attributed definitions can be implemented during bottom-up parsing.

A SDD is *L-attributed* if every attribute is either: synthesized or inherited. The rules for inherited attributed are mentioned in the nearby box.

Inherited, but with the rules limited as follows. Suppose that there is a production $A \rightarrow X_1 X_2 \cdots X_n$, and that there is an inherited attribute $X_i.a$ computed by a rule associated with this production. Then the rule may use only:

(a) Inherited attributes associated with the head A.

(b) Either inherited or synthesized attributes associated with the occurrences of symbols $X_1, X_2, \ldots, X_{i-1}$ located to the left of X_i.

(c) Inherited or synthesized attributes associated with this occurrence of X_i itself, but only in such a way that there are no cycles in a dependency graph formed by the attributes of this X_i.

Is the SDD from exercise 1 L-attributed? Explain.

How about a SDD containing the following rules? Explain your answer.

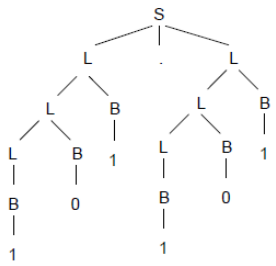
PRODUCTION	SEMANTIC RULES
$A \rightarrow B\ C$	$A.s = B.b;$ $B.i = f(C.c, A.s)$

3. 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 specific 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 → L . L | L
L → L B | B
B → 0 | 1

Parse Tree



4. Read carefully the following information

Syntax-Directed Translation (SDT): A translation scheme is another way of specifying a syntax-directed translation: semantic actions (enclosed in braces) are embedded within the right-sides of the productions of a context-free grammar. For example, rest --> + term { print ('+') } rest1. This indicates that a plus sign should be printed between the depth-first traversal of the term node and the depth-first traversal of the rest1 node of the parse tree.

How to implement a syntax-directed translation using a predictive parser? It is not obvious how to do this, since the predictive parser works by building the parse tree top-down, while the syntax directed translation needs to be computed bottom-up.

We give the parser a second stack called the semantic stack:

- The semantic stack holds nonterminals' translations; when the parse is finished, it will hold just one value: the translation of the root nonterminal (which is the translation of the whole input).
- Values are pushed onto the semantic stack (and popped off) by adding **actions** to the grammar rules. The action for one rule must:
  - Pop the translations of all right-hand-side nonterminals.
  - Compute and push the translation of the left-hand-side nonterminal.
- The actions themselves are represented by **action numbers**, which become part of the right-hand sides of the grammar rules. They are pushed onto the (normal) stack along with the terminal and nonterminal symbols. When an action number is the top-of-stack symbol, it is popped and the action is carried out.

So what actually happens is that the action for a grammar rule `X -> Y1 Y2 ... Yn` is pushed onto the (normal) stack when the derivation step `X -> Y1 Y2 ... Yn` is made, but the action is not actually performed until complete derivations for all of the `Y`'s have been carried out.

Exercise:

For the following grammar, give (a) translation rules, (b) translation actions with numbers, and (c) a CFG with action numbers, so that the translation of an input expression is the value of the expression. Is the grammar LL(1)?

```
exp     -> exp + term
        -> exp - term
        -> term
term    -> term * factor
        -> term / factor
        -> factor
factor -> INTLITERAL
        -> ( exp )
```

Transform the grammar rules with actions that you wrote to LL(1) form. Trace the actions of the predictive parser on the input 2 + 3 * 4.

For example, consider the following syntax-directed translation for the language of balanced parentheses and square brackets. The translation of a string in the language is the number of parenthesis pairs in the string.

```
CFG                          Translation Rules
===                          =================
exp -> epsilon               exp.trans = 0
    -> ( exp )               exp1.trans = exp2.trans + 1
    -> [ exp ]               exp1.trans = exp2.trans
```

The first step is to replace the translation rules with **translation actions**. Each action must:

- Pop all right-hand-side nonterminals' translations from the semantic stack.
- Compute and push the left-hand-side nonterminal's translation.

Here are the translation actions:

```
CFG                          Translation Actions
===                          ===================
exp -> epsilon               push(0);
    -> ( exp )               exp2Trans = pop(); push( exp2Trans + 1 );
    -> [ exp ]               exp2Trans = pop(); push( exp2Trans );
```

Next, each action is represented by a unique action number, and those action numbers become part of the grammar rules:

```
CFG with Actions
================
exp -> epsilon #1
    -> ( exp ) #2
    -> [ exp ] #3

#1: push(0);
#2: exp2Trans = pop(); push( exp2Trans + 1 );
#3: exp2Trans = pop(); push( exp2Trans );
```

Note that since action #3 just pushes exactly what is popped, that action is redundant, and it is not necessary to have any action associated with the third grammar rule. Here's a picture that illustrates what happens when the input "([])" is parsed (assuming that we have removed action #3):

```
input so far   stack            semantic stack  action
------------   -----            --------------  ------
   (           exp EOF                          pop, push "( exp ) #2"
   (           (exp) #2 EOF                     pop, scan
   ([          exp) #2 EOF                      pop, push "[ exp ]"
   ([          [exp] ) #2 EOF                   pop, scan
   ([]         exp] ) #2 EOF                    pop, push epsilon #1
   ([]         #1 ] ) #2 EOF                    pop, do action
   ([]         ] ) #2 EOF          0            pop, scan
   ([])        ) #2 EOF            0            pop, scan
   ([]) EOF    #2 EOF              0            pop, do action
   ([]) EOF    EOF                 1            pop, scan
   ([]) EOF                                     empty stack: input accepted!
                                                translation of input = 1
```

```
declsTrans = pop();
push( stmtsTrans + declsTrans );
```

Note that the right-hand-side nonterminals' translations are popped from the semantic stack *right-to-left*. That is because the predictive parser does a leftmost derivation, so the `varDecls` nonterminal gets "expanded" first; i.e., its parse tree is created before the parse tree for the `stmts` nonterminal. This means that the actions that create the translation of the `varDecls` nonterminal are performed first, and thus its translation is pushed onto the semantic stack first.

Another issue that has not been illustrated yet arises when a left-hand-side nonterminal's translation depends on the value of a right-hand-side *terminal*. In that case, it is important to put the action number *before* that terminal symbol when incorporating actions into grammar rules. This is because a terminal symbol's value is available during the parse only when it is the "current token". For example, if the translation of an arithmetic expression is the value of the expression:

```
CFG Rule:               factor -> INTLITERAL
Translation Rule:       factor.trans = INTLITERAL.value
Translation Action:     push( INTLITERAL.value )
CFG rule with Action:   factor -> #1 INTLITERAL       // action BEFORE terminal
                        #1: push( currToken.value )
```

In the example above, there is no grammar rule with more than one nonterminal on the right-hand side. If there were, the translation action for that rule would have to do one pop for each right-hand-side nonterminal. For example, suppose we are using a grammar that includes the rule: `methodBody -> { varDecls stmts }`, and that the syntax-directed translation is counting the number of declarations and statements in each method body (so the translation of `varDecls` is the number of derived declarations, the translation of `stmts` is the number of derived statements, and the translation of `methodBody` is the number of derived declarations and statements).

```
CFG Rule:               methodBody -> { varDecls stmts }
Translation Rule:       methodBody.trans = varDecls.trans + stmts.trans
Translation Action:     stmtsTrans = pop(); declsTrans = pop();
                        push( stmtsTrans + declsTrans );
CFG rule with Action:   methodBody -> { varDecls stmts } #1
                        #1: stmtsTrans = pop();
```

Handling Non-LL(1) Grammars

Recall that a non-LL(1) grammar must be transformed to an equivalent LL(1) grammar if it is to be parsed using a predictive parser. Recall also that the transformed grammar usually does not reflect the underlying structure the way the original grammar did. For example, when left recursion is removed from the grammar for arithmetic expressions, we get grammar rules like this:

```
exp  -> term exp'
exp' -> epsilon
     -> + term exp'
```

It is not at all clear how to define a syntax-directed translation for rules like these. The solution is to define the syntax-directed translation using the *original* grammar (define translation rules, convert them to actions that

push and pop using the semantic stack, and then incorporate the action numbers into the grammar rules). Then convert the grammar to be LL(1), *treating the action numbers just like grammar symbols*!

For example:

```
Non-LL(1) Grammar Rules With Actions
====================================
exp  -> exp + term #1
     -> term
term -> term * factor #2
     -> factor

#1: TTrans = pop(); ETrans = pop(); push Etrans + TTrans;
#2: FTrans = pop(); TTrans = pop(); push Ttrans * FTrans;

After Removing Immediate Left Recursion
=======================================
exp  -> term exp'
exp' -> + term #1 exp'
     -> epsilon

term  -> factor term'
term' -> * factor #2 term'
      -> epsilon
```
