IMPLEMENTING SYNTAX DIRECTED DEFINITIONS

Using the SableCC Parser Generator

Syntax Directed Definitions

A *syntax directed definition* is a context free grammar augmented with attributes and evaluation rules. The semantic and non-context-free aspects of a language can be defined by associating one or more attributes with the symbols of a grammar. An *attribute* is a semantic property of a grammar symbol, and both terminal and non-terminal symbols can have semantic attributes. The *evaluation rules* specify how to calculate the attributes associated with each grammar symbol. Evaluation rules are sometimes called *semantic actions.*

A syntax directed definition is a high-level specification of the translation rules for a language; it hides many implementation details such as the order of evaluation and the programming language specific types of its attributes. Parser generating tools, like SableCC, make it easy to implement a syntax directed definition because they provide an infrastructure for inserting semantic actions.

There are two kinds of attributes, inherited and synthesized. Intuitively, a *synthesized attribute* is determined from information arising from the internal constituents of a construct (i.e., the right-hand side of a production rule), whereas an *inherited attribute* is determined by information coming from the external context of a construct.

Another way to think about synthesized versus inherited attributes is in terms of the parse tree (i.e., syntax tree). Production rules define the parse tree structure, that is, the symbol on the left-hand side of a production rule is the parent node and the symbols on the right hand side are its children (an example is given in Figure 3). A synthesized attribute is calculated using the attributes of the node's children. For example, the type of an expression is calculated using the types of its sub-expressions. Specifically, in Java, the type of expression `1 + 0.01' (float) is calculated from the types of its sub-expressions '1' (int) and `0.01' (float), its internal constituents. On the other hand, an inherited attribute is an attribute of a child node that is calculated from the attributes of its parent and possibly its siblings. Intuitively, calculating the value of inherited attributes involves, at least in part, attribute information flowing from parent to child in the parse tree; however information flows only from child to parent in the calculation of synthesized attributes.

The process of computing the attribute values of the nodes of a parse tree is called annotating or decorating the tree. For example, running your lab 5 type checker decorates the expression nodes of the syntax tree with their types, that is, it annotates the sub-expression nodes with their types and then uses these types to determine and annotate the parent expression node with its type. Running the **ParserDriver.main** method shown in Figure 7 also decorates a parse tree. The details will be explained in the following sections.

Implementing Syntax Directed Definitions Using SableCC

When using SableCC, we create syntax directed definitions as follows. First we specify a parser for a context free grammar. The parser generates a syntax tree that can be walked by classes defined in the "analysis" package. We add evaluation rules by overriding methods of one of these analysis (tree traversal) classes, e.g., **DepthFirstAdapter**. We associate attributes with nodes using the **setOut** method provided by the tree traversal classes. We retrieve these attributes using the method **getOut**. Methods **setOut** and **getOut** are used to set and retrieve synthesized attributes.

To implement a syntax directed definition, all we need to know is which class to extend and which methods to override. The class to be extended is the **DepthFirstAdapter** class generated by SableCC and located in the analysis package (subdirectory); this class does a depth first traversal of the parse tree although the default is not to do anything during this traversal. To implement a syntax directed definition, we will have to add evaluation rules by overriding some of the methods of **DepthFirstAdapter**, that is, we will need to create a subclass that overrides these methods. The names of the methods we override depend on the names of the classes generated by SableCC. Recall that each grammar rule is represented by a class in the node package; the names of these classes are determined by the grammar specification. (Figure 6 gives many examples of how we can override the methods of **DepthFirstAdapter**.)

```
class YourTreeDecorator extends DepthFirstAdapter {
    void inAWholePart(AWholePart node) {
        // your code for execution prior to traversing an AWholePart node.
    }
    void outAWholePart(AWholePart node) {
        // your code for execution after traversing an AWholePart node.
    }
    void inABitWholePart(ABitWholePart node) {
        // your code for execution prior to traversing an ABitWholePart node.
    }
    void outABitWholePart(ABitWholePart node) {
        // your code for execution after traversing an ABitWholePart node.
    }
}
```

Figure 1: A skeleton program of a class that traverses parse trees generated by SableCC.

For example, consider the following SableCC grammar rules.

```
whole_part = whole_part  bit
           | { bit } bit
           ;
```

The names of the "node" classes for these rules are **AWholePart** and **ABitWholePart**. The names come from the non-terminal on the left hand side of the two production rules (whole_part) prefixed by "A" and the name inside the braces (when there is such a name as in the second rule above). The name of the method you have to override and the type of its parameter is based on these class names.

Attribute Evaluation

Attribute evaluation rules are implemented and the order of evaluation is determined through method overrides. For example, in order to insert an evaluation rule before traversing an **AWholePart** node we would have to override method **inAWholePart**. Similarly, to insert an evaluation rule after traversing an **AWholePart** node, we have to override **outAWholePart**. Both of these methods take a single argument of type **AWholePart** (See Figure 1).

We only have to override a method when we need to insert some action either before or after the traversal of the parent node's children. The following describes a small sample part of such a class. The superclass **DepthFirstAdapter** does a depth first traversal of the parse tree generated by SableCC. This example shows the method overrides required if actions have to be inserted both before and after the traversal of an **AWholePart** node and an **ABitWholePart** node.

Usually "in" methods have to be overridden to implement inherited attributes, and "out" methods must be overridden to implement synthesized attributes. For example, by definition, a synthesized attribute of a parent node is computed from the attributes of its children. Therefore, the child nodes must have been traversed (so the attributes of the children have been computed) prior to computing the attribute of the parent. Hence, the evaluation rules for computing a synthesized attribute for a parent must be evaluated after evaluating the rules for (i.e. traversing) the child nodes, i.e., the "out" methods would have to be overridden. Figure 6 gives an example of the computation of a synthesized attribute, i.e., the value of a binary numeral.

SableCC Syntax Specification	Evaluation Rule
whole_part = whole_part bit	$whole\_part_0.value = whole\_part_1.value * 2 + bit.value$

```
    public void outAWholePart(AWholePart node) {
(1)   Long wholeValue = (Long) getOut(node.getWholePart());
(2)   long wValue = wholeValue.longValue();
(3)   Long bitValue = (Long) getOut(node.getBit());
(4)   Long newValue = new Long(wValue * 2 + bitValue.longValue());
(5)   setOut(node, newValue);
    }
```

Figure 2: An example of a production rule and its associated action; the method above shows how the action would be implemented.

Setting and Retrieving Attributes

Before we can implement a syntax directed definition, it is important that we understand how to set and retrieve attributes; this will be important for labs 4 and 5 and for the final exam. The calls of methods **setOut** and **getOut** allow us to set an attribute for a particular node and then to later retrieve that attribute. In the examples that follow (and in labs 4 and 5), only one attribute will be associated with any particular node. Consider for example the following production rule and its associated evaluation rule. The subscripts are so the two occurrences of **whole_part** in the production rule (left and right-hand side) can be distinguished (the subscript 0 denotes the occurrence on the left-hand side).

Figure 2 illustrates the way that attributes are set and then later retrieved. The **node** parameter in the method in Figure 2 represents the syntax tree for the parent node, the left-hand side of the rule shown. The SableCC tool generates methods so the children of a syntax tree node can easily be retrieved. For example, the above rule has two children (**whole_part** and **bit**); they can be retrieved through calls to **getWholePart()** and **getBit()** (see lines 1 and 3 above).

Line 1 of the method in Figure 2 shows how an attribute is retrieved. The method **getOut** retrieves the attribute of its argument, in this case, the attribute of **node.getWholePart()** (i.e., $\mathbf{whole\_part_1.value}$). (obviously, the attribute for the node representing $\mathbf{whole\_part_1}$ must have been set by another rule previously, otherwise the value attribute will be null). In this example, the attribute has type **Long** (the built-in Java class). An attribute can be any user-defined type or built-in class (i.e., a subclass of **Object**). Often, attributes will have to be downcast as in line 1 since method **getOut** returns an **Object**.

Line 2 shows how to retrieve the long value stored in a Long object (see Figure C below for the complete example).

Line 4 of the method shows the action needed to compute the value of the new attribute (a new **Long** object). Line 5 shows how the attribute for the parent node is set. In this case, the attribute for the non- terminal on the left-hand side is set to the value computed in line 4 which must have type **Long**.

Figures 4, 5, 6, and 7 give a complete example of how a syntax directed definition (SDD) can be implemented. Figure 4 defines the SDD. Figure 5 is a SableCC parser specification. Figures 6 is a Java program that decorates the tree and Figure 7 is a driver program that runs the SDD.

Retrieving the text associated with a token

Sometimes we have to retrieve the string value of a token. Retrieving the string value of a token is done through method **getText()**; this will be needed for tokens such as identifiers in labs 4 and 5. Note, however, that method **getText()** is ONLY defined for token classes (those class names that start with a "T") and not for classes representing non-terminals.

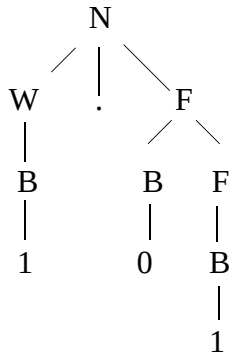

Figure 3: The parse tree for binary numeral "1.01".

Important Note on Exceptions

When using the framework generated by SableCC, we always have to be very careful about setting the attributes. That means we have to make sure that we know what the type of the attribute should be so that we set it properly. This is because attributes can only have type **Object**, so there are no error messages if we make a mistake when setting an attribute (or when failing to set an attribute).

If the parent expects the child node to have an attribute, then the action associated with the child node must properly set that attribute using **setOut**. Note carefully the attribute types in Figure 6 below. **PWholePart** always has an attribute with type **Long** because its two subclasses, **AWholePart** and **ADigitWholePart** (the nodes representing the two alternatives), set the attribute to a **Long** object. However, **PFractionPart**, i.e., **AFractionPart** and **ADigitFractionPart**, always has an attribute with type **Double**. Also, the type of the attribute for **PNumeral** is always **Double**.

NullPointer Exceptions

When creating a Syntax Directed Definition, we always have to be very careful to set the attribute of the base case of recursive rules, e.g., rules 6 and 8 of Figure 4. If the attributes of these rules are not set we will get a **NullPointerException** when the program runs because the attribute will not have been set and will thus be null. Consider the derivation for the binary numeral "1.01".

N ==> W . F (rule 3)
==> B . F (rule 6)
==> 1. F (rule 10)
==> 1 . B F (rule 7)
==> 1 . 0 F (rule 9)
==> 1 . 0 B (rule 8)
==> 1 . 0 1 (rule 10)

The corresponding parse tree would be as in Figure 3. If any node in the parse tree does not pass a synthesized attribute up the tree (by setting the attribute of the parent), then your program will get a **NullPointerException** at run-time. For example, if method `outABitWholePart` in Figure 6 is omitted or is not overridden, then the program will not run properly.

A Simple Example

Figures 4, 5, and 6 give a very simple example of a syntax directed definition and how it would be implemented using the SableCC tool and the Java programming language. This syntax directed definition translates a binary numeral into its base 10 decimal equivalent. Notice that the syntax directed definition (attribute grammar) in Figure 4 is more mathematical and "abstract" than the implementation given in Figure 6; it is more abstract in the sense that the type of the "value" attribute is not specified (we just know that it is a decimal number). However, in the implementation in Figure 6, the type of the attribute is very impor-

				Evaluation Rule
(1)	N	→	W	N.value = W.value
(2)	N	→	W .	N.value = W.value
(3)	N	→	W . F	N.value = W.value + F.value
(4)	N	→	. F	N.value = F.value
(5)	W	→	W B	$W_0 = W_1$.value * 2 + B.value
(6)	W	→	B	W.value = B.value
(7)	F	→	B F	F_0.value = (B.value + F_1.value) * 0.5
(8)	F	→	B	F.value = B.value * 0.5
(9)	B	→	'0'	B.value = 0
(10)	B	→	'1'	B.value = 1

Figure 4: An example of a syntax directed definition.

---

tant. If the type of the attribute has not been properly set by any of the children nodes, then the parent will get a null pointer exception or class cast exception. Carefully study this example before starting lab 4. Labs 4 and 5 assume that this example is well understood.

First, one has to determine the type of the attribute for each non-terminal in the grammar. For example, in the grammar and implementation below, the type of the attributes are as follows:

numeral.value has type Double (N.value in rules 1, 2, 3, and 4 above)

whole_part.value has type Long (W.value in rules 5 and 6 above)

fraction_part.value has type Double (F.value in rules 7 and 8 above)

bit.value has type Long (D.value in rules 9 and 10 above)

Decorating the Syntax Tree

The Java program in Figure 7 shows how to instantiate the parser (line (1)), create a syntax tree representing the input program (line (2)), and then decorate the tree with its attributes. Line (3) below shows the instantiation of the tree traversal class object. Line (4) shows how to decorate the tree, i.e., compute the value attributes for each tree node. Line (5) shows how to get the attribute assigned to the tree node. Line (6) shows the printing of the final attribute value for the input numeral.

The framework generated by SableCC uses the "visitor pattern" to walk and decorate the syntax tree. The visitor pattern is described in "Design Patterns: Abstraction and Reuse of Object-Oriented Design" by E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides. This pattern is also introduced in the Advanced Software Development course that some of you may have already taken. However, it is NOT necessary to know how the visitor pattern works in order to implement a syntax directed definition using SableCC.

Before you begin labs 4 and 5, make sure you understand this example thoroughly; this will save you a lot of time and effort.

```
// ********************** Helpers *********************************

Helpers
    all = [0..0xffff];
    space  = ' ';
    lf  = 0x000a; // line feed
    cr  = 0x000d; // carriage return
    ff  = 0x000c; // form feed
    ht  = 0x0009; // tab

    new_line = lf | cr | cr lf ;
    not_cr_lf = [all - [cr + lf]];

// ********************** Tokens **********************************

Tokens
    white_space = (space | ht | ff | new_line);

// ********************** literals ********************************

    zero     =     '0' ;
    one      =     '1' ;
    dot      =     '.' ;

// Ignored Tokens

// ********************** Productions *****************************

Productions

goal
       = white_space*  numeral  [ws]:white_space*
       ;
numeral
       = {whole} whole_part  dot?                      // rules 1 and 2
       | {fraction} whole_part?  dot  fraction_part    // rules 3 and 4
       ;
whole_part
       = whole_part  bit           // rule 5
       | {bit} bit                 // rule 6
       ;
fraction_part
       = bit  fraction_part        // rule 7
       | {bit} bit                 // rule 8
       ;
bit
       = {zero} zero              // rule 9
       | {one} one                // rule 10
       ;
```

Figure 5: The parser specification for the syntax directed definition given in Figure 4.

```
public class NumeralTranslator extends DepthFirstAdapter {
    public void outStart(Start node) {
       Double numValue = (Double) getOut(node.getPGoal());
       setOut(node, numValue);
    }
    public void outAGoal(AGoal node) {
       setOut(node, getOut(node.getNumeral()));
    }
    public void outAWholeNumeral(AWholeNumeral node) {
       Long wholeValue = (Long) getOut(node.getWholePart());
       long wValue = wholeValue.longValue();
       setOut(node, new Double(wValue));
    }
    public void outAFractionNumeral(AFractionNumeral node) {
       Double fractionValue = (Double) getOut(node.getFractionPart());
       double fValue = fractionValue.doubleValue();
       double wValue = 0.0;
      // the next if-statement is needed because whole_part is optional
       if (node.getWholePart() != null) {
           Long wholeValue = (Long) getOut(node.getWholePart());
           wValue = wholeValue.longValue();
       }
       setOut(node, new Double(wValue + fValue));
    }
    public void outAWholePart(AWholePart node) {
       Long wholeValue = (Long) getOut(node.getWholePart());
       long wValue = wholeValue.longValue();
       Long bitValue = (Long) getOut(node.getBit());
       setOut(node, new Long(wValue * 2 + bitValue.longValue()));
    }
    public void outABitWholePart(ABitWholePart node) {
       Long bitValue = (Long) getOut(node.getBit());
       setOut(node, bitValue);
    }
    public void outAFractionPart(AFractionPart node) {
       Long bitValue = (Long) getOut(node.getBit());
       long bValue = bitValue.longValue();
       Double fractionValue = (Double) getOut(node.getFractionPart());
       double newValue = (bValue + fractionValue.doubleValue()) * 0.5;
       setOut(node, new Double(newValue));
    }
    public void outABitFractionPart(ABitFractionPart node) {
       Long bitValue = (Long) getOut(node.getBit());
       double newValue = bitValue.longValue() * 0.5;
       setOut(node, new Double(newValue));
    }
    public void outAZeroBit(AZeroBit node) {
       setOut(node, new Long(0));
    }
    public void outAOneBit(AOneBit node) {
       setOut(node, new Long(1));
    }
}
```

Figure 6: The implementation of the syntax directed definition given in Figure A based on the parser specification of Figure B.

```
import java.io.*;
import lexer.*;
import parser.*;
import node.*;

public class ParserDriver {

    public static void main(String[] args)
    {
       try {
           System.out.println("\nStarting Lexer");
           Lexer lex = new Lexer(
              new PushbackReader( new InputStreamReader(
                                   new FileInputStream(args[0])),
                                  1024));

           System.out.println("Starting Parser");
(1)        Parser p = new Parser(lex);

(2)        Start tree = p.parse();             // Parse the input.

           System.out.println("\nInterpreting decimal numerals\n");
(3)        NumeralTranslator interpreter = new NumeralTranslator();
(4)        tree.apply(interpreter);
(5)        Double val = (Double)interpreter.getOut(tree);
(6)        System.out.println(tree.toString() + "--> " + val);

       } catch (ParserException e) {
           System.out.println(e.getMessage());
       } catch (Exception e) {
           System.out.println(e.getMessage());
           e.printStackTrace();
       }
    }
}
```

Figure 7: The parser driver program for the syntax directed definition given in Figure A and its implementation in Figures B and C.

Exercises

1. (a) The syntax directed definition and implementation given in Figures 4, 5, and translates a binary numeral into a decimal number. Define a syntax directed translation that translates a decimal numeral into a decimal number (similar to Figure A except it must be for base 10 numerals instead of base 2). Note that a *numeral* is a string of characters, but a *number* is a numerical value.

1. (b) Give a SableCC parser specification for the grammar and translation defined in part (a).

1. (c) Implement the syntax directed definition for decimal numerals based on the SableCC parser specification you defined in part (b). That is, define the class **NumeralTranslator** that calculates the value of a decimal numeral. The type of the attributes should be **Long** and **Double** or all **Double** objects.

2. (a) Define a syntax directed definition (CFG and evaluation rules) that translates octal numerals into decimal numbers (i.e., create a syntax directed definition like the one in Figure A, except it should be for octal

numerals). Octal numerals are strings over the alphabet { 0, 1, 2, 3, 4, 5, 6, 7 }. The value of an octal numeral is computed using the usual base 8 arithmetic. For example, 1275 in octal has the value 701 as a decimal number, i.e.,

$$\mathbf{1} * 8^3 + \mathbf{2} * 8^2 + \mathbf{7} * 8^1 + \mathbf{5} * 8^0 = \mathbf{512 + 128 + 56 + 5 = 701}$$

Hint: ((**1** * 8 + **2**) * 8 + **7**)*8 + **5** = 701 That is, take the left-most octal digit (1) and multiply by 8 and add the next octal digit (2), i.e., 1*8 + 2. Now take that value (10), multiply by 8 and add the next digit (7), i.e., 10*8 + 7. Now take that value (87), multiply by 8 and add the next digit (5), i.e., 87*8 + 5 = 701. We did something similar for binary numerals except we multiplied by 2 for whole numbers and .5 for fractions. This question is only asking for whole octal numbers (there is no octal point or fraction part).

2. (b) Implement the syntax directed definition defined in part (a) (SableCC parser specification and tree decorator class) that computes the decimal value of octal numerals (i.e., compute the "**value**" attribute for the production rules in the CFG defined in part (a)).