SableCC, an Object-Oriented Compiler Framework

Etienne M. Gagnon and Laurie J. Hendren *
Sable Research Group
School of Computer Science
McGill University, Quebec, Canada
[gagnon,hendren]@sable.mcgill.ca

Abstract

In this paper, we introduce SableCC, an object-oriented framework that generates compilers (and interpreters) in the Java programming language. This framework is based on two fundamental design decisions. Firstly, the framework uses object-oriented techniques to automatically build a strictly-typed abstract syntax tree that matches the grammar of the compiled language which simplifies debugging. Secondly, the framework generates tree-walker classes using an extended version of the visitor design pattern which enables the implementation of actions on the nodes of the abstract syntax tree using inheritance. These two design decisions lead to a tool that supports a shorter development cycle for constructing compilers.

To demonstrate the simplicity of the framework, we present all the steps of building an interpreter for a mini-BASIC language. This example could be easily modified to provide an embedded scripting language in an application. We also provide a brief description of larger systems that have been implemented using the SableCC tool.

We conclude that the use of object-oriented techniques significantly reduces the length of the programmer written code, can shorten the development time and finally, makes the code easier to read and maintain.

1: Introduction

The number of computer languages in use today is overwhelming. Ranging from general purpose to highly specialized, they are present in almost all areas of computing. There are mainstream programming languages like C, Fortran, Pascal, but also many other languages used in domain-specific applications. Computer languages can be used to describe many things, other than computer processing. HTML[12] or TeX[8] are used to describe formatted documents. A domain-specific language like HL7[1] is used to exchange health care information uniformly across the world. It would impossible to list all the uses here, but it is worth noting that these languages are often embedded in larger systems. For example, many word processing applications have their own tiny macro language to allow the automation of commands. In component-based development environments, small scripting languages are used to glue together components.

In the 1950's, writing a compiler was very difficult. It took 18 staff-years to implement the first FORTRAN compiler[2]. Since then, advances in the theory of compilers and the

---

* This work was partly supported NSERC.

0-8186-8482-8/98 $10.00 © 1998 IEEE

development of many compiler tools have simplified this task greatly. Writing a compiler is now a feasible task for any programmer with minimal knowledge of compiler techniques. This simplicity is achieved due to the use of *compiler compilers*. A compiler compiler is a program that translates a specification into a compiler for the programming language described in the specification. This relieves the programmer from the burden of writing the lexical and syntactical analysis code.

Over the years, many compiler compilers have been developed. The scope of these tools varies. While some will build a complete compiler (*end-to-end*) from a specification, others will only build the *front-end* of a compiler (lexer and/or parser). It may seem, at first glance, that an end-to-end compiler compiler will be more powerful. However, in practice, front-end compiler compilers are normally integrated with a general purpose programming language. This way, the implementation of complex data structures, optimizations, and code analyses is easier because it is done in the programmer's native programming language. Front-end compiler compilers exist for almost all major programming languages in use today.

In the last few years, the $Java^{TM}$[4] programming language has gained a remarkable popularity on the Internet. Although superficially Java has a syntax similar to C++, Java also has many additional features of modern high-level object-oriented programming languages. For example, Java has a garbage collector, a cleaner inheritance mechanism with classes and interfaces, and a rich standard cross-platform library with support for graphical user interfaces and network programming. One of the most interesting properties of Java is the portability of object code. Java source files are compiled to platform independent *bytecode* instructions. At runtime, these bytecodes are interpreted by a *Java Virtual Machine*[10] to perform the actual computation.

We have developed *SableCC*, to generate compilers written in the Java programming language. SableCC sits in the middle between front-end and end-to-end compiler compilers. It not only generates a lexer and a parser, but it also builds a complete set of classes. Our main goal was to provide a framework that would simplify the development of easy to maintain compilers. To achieve this goal, the framework abides by the two following properties:

- All data structures are self-preserving. By preventing the programmer from corrupting data structures, the system helps reducing debugging time. This property is achieved by strictly typing the abstract syntax tree.
- The addition of new functionality should be feasible through the addition of programmer-defined classes. This implies a clear separation between automatically generated classes and programmer written classes and thus, helps debugging. This is achieved by providing *ready-to-be-inherited-from* tree-walker classes. These classes implement the visitor design pattern to enable the use of inheritance as a means to add actions. (We use the term *actions* to refer to the code written by a programmer to be executed on specific nodes of the abstract syntax tree).

This paper is organized as follows. In section 2 we give a high level description of SableCC. In section 3 we explain the relation between a grammar and a typed abstract syntax tree. In section 4 we explore the visitor design pattern and explain the few extensions that we have implemented in our framework. In section 5 we use SableCC to build an interpreter for a mini-BASIC language. In section 6 we briefly describe other compilers that were built using the SableCC tool. In section 7 we discuss related work, and in section 8 we present our conclusions.

2: SableCC

SableCC represents the result of our research to develop a Java compiler compiler that meets new compiler implementation trends. More specifically:

- Modern compilers usually implement many passes over the compiled program. One pass compilers (like early PASCAL compilers) are seldom used anymore.
- Many compilers work on AST (*Abstract Syntax Tree*) representation of programs.
- As a compiler evolves over time, new analyses and optimizations are added to the compiler.
- A compiler, like any other software, must be maintainable.

To address these issues we have developed a new approach for compiler compiler tools. In our approach, the compiler compiler place in the development cycle has been reduced to merely build an initial object-oriented framework that is based solely on the lexical and grammatical definition of the compiled language. This has the advantage of limiting framework modifications to the case where the grammar of the compiled language is changed.

On the other hand, the richness of the generated environment has been increased. So, in the generated framework:

- The parser automatically builds the AST of the compiled program.
- Each AST node is strictly typed, ensuring no corruption occurs in the tree.
- Each analysis is written in its own class. Writing a new analysis only requires extending a tree-walker class and providing methods to do the work at appropriate nodes.
- Storage of analysis information is kept in the analysis class itself, outside the definition of node types. This ensures no modification to a node type is needed when a new analysis is added to or removed from the compiler.

The framework makes extensive use of object-oriented design patterns to achieve modularity of code. The resulting compiler becomes a very maintainable compiler. In some cases we have opted for good object-oriented design over fast code. It is our belief that over time, new processors get faster and memory gets cheaper, but the same old code base is often used to generate new compilers. So good software engineering is important in the long term.

We have developed SableCC in the Java programming language. It runs on any platform supporting the Java Development Kit 1.1 or newer.

2.1: General steps to build a compiler using SableCC

Producing a compiler using SableCC requires the following steps (as shown in figure 1):

1. Creating a SableCC specification file containing the lexical definitions and the grammar of the compiled language.
2. Launching SableCC on the specification file to generate a framework.
3. Creating one or more *working classes*, possibly inheriting from classes generated by SableCC.

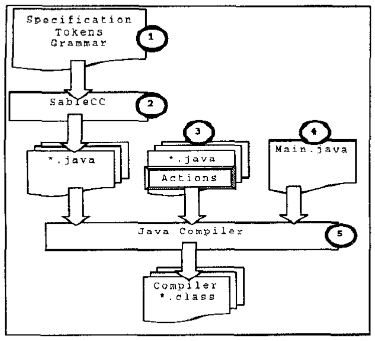

Figure 1. Steps to create a compiler using SableCC

4. Creating a main compiler class that activates the lexer, parser and working classes.
5. Compiling the compiler with the Java compiler.

By *working classes* we mean classes that contain the core compiler functionality. These classes can be analyses, transformations on the AST, or simply code generation classes.

We must note that SableCC, as other compiler compiler tools, can also be used to build interpreters. In such a case, as we will see in section 5, a working class can be the interpreter itself.

2.2: SableCC specification files

A SableCC specification file is a text file that contains the lexical definitions and the grammar productions of the language to be recognized by the generated compiler framework. It also specifies a destination root Java package for generated files.

Unlike other compiler compilers, there is no place to put action code associated with a token or a production. This design has the advantage of adding stability to the framework. Modifications to the framework are limited to when the grammar of the compiled language is changed. Adding, changing or even removing action code (in working classes) does not affect the generated framework in any way.

2.3: SableCC generated files

On output, SableCC generates files into four sub-packages of the specified root package. The packages are named: `lexer`, `parser`, `node` and `analysis`. Each file contains either a class or an interface definition.

- The `lexer` package contains the `Lexer` and `LexerException` classes. These classes are respectively the generated lexer and the exception thrown in case of a lexing error.
- The `parser` package contains the `Parser` and `ParserException` classes. As expected, these classes are the parser and the exception thrown in case of a parsing errors.
- The `node` package contains all the classes defining the typed AST.
- The `analysis` package contains one interface and three classes. These classes are used mainly to define AST walkers.

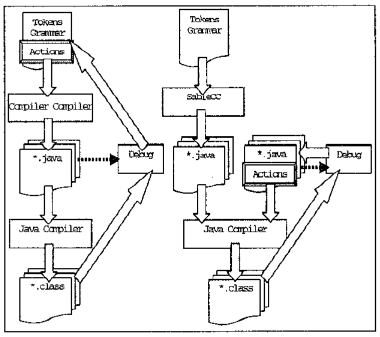

Figure 2. Traditional versus SableCC actions debugging cycle

2.4: Compiler development cycle

The choices we have made in the design of the SableCC compiler framework have a direct incidence on the development cycle of a compiler.

As illustrated by the left diagram of figure 2, with traditional *compiler compilers*, the cycle involves the following steps. First, the programmer writes or fixes the grammar and action code in the specification file. Then the source code for the compiler is generated. The source code is then compiled to an executable program. The program is then tested and debugged. The problem in this cycle is that the source files of the debugged program are generated. So, if a bug is found in this source code, the programmer has to find the corresponding code in the specification file and fix it there.

As shown by the right diagram of figure 2, with SableCC, this cycle is shortened. Since actions are directly written as Java classes, the source code of the debugged program is the programmer written code itself. This enables interactive debugging in an Integrated Development Environment.

3: The grammar and abstract syntax tree

The first phase of a compiler works as follows: The lexer module breaks the input into small meaningful sequences of characters called *tokens*. Then, the parser module verifies that the sequence of tokens returned by the lexer module conforms to a grammar. If not, the parser issues an error and exits. (Some compilers try to bypass the error to catch further errors, but this is beyond the scope of this paper).

A grammar is a set of rules that defines the syntax of a programming language. We will informally define a grammar by looking at an example:

```
Tokens
  l_par = '(';   r_par = ')';
  plus = '+';    number = ['0'..'9'];
Productions  /* grammar */
  exp = number | add;
  add = l_par exp plus exp r_par;
```

This grammar specifies a small arithmetic language. In this example `exp` is a *production* that has two *alternatives* (`number` and `add`), and `add` is a *production* that has a single *alternative* (`l_par`...`r_par`). An *alternative* has zero or more *elements*. For example, the last alternative had five *elements*. Each *element* is either a *production name* or a *token name*.

The second element (`exp`) of the single alternative of production **add**, stands for all possible alternatives of production `exp` . Note that it cannot stand for anything else. This behavior can be mimicked through inheritance in Java. SableCC could define an abstract class `PExp` denoting production `exp`, and define two classes `AExp1 extends PExp` and `AExp2 extends PExp`, denoting respectively the first and second alternatives of production `exp`. This would give us a type system that enforces a variable of type `PExp` to hold only values of type `AExp1` or `AExp2`. For our example grammar, SableCC would generate the following classes defining an *abstract syntax tree* (AST) for the grammar:

```
abstract class PExp {}
class AExp1 extends PExp {TNumber elem1;}
class AExp2 extends PExp {PAdd elem1;}
```

```
abstract class PAdd {}
class AAdd1 extends PAdd
{TLPar elem1; PExp elem2; TPlus elem3;
 PExp elem4; TRPar elem5; }
```

`PExp` and `PAnd` are abstract classes (meaning that no instance of these types can be created, only instances of derived type), to prevent the construction of meaningless ASTs.

We have implemented this inheritance and naming scheme in an early version of SableCC. But, after some usage and feedback, it was felt that looking for the fourth (or was is the fifth?) element of the third alternative of production p made the code less readable and made code maintenance more difficult. It is easy to inadvertently type `PExp2` instead of `PExp3` in a variable declaration, but it can be quite difficult to find such an error.

To resolve this problem, the current version of SableCC requires a name for every alternative, if a production has more than one. The name is added at the begining of the alternative between braces. Additionally, SableCC requires a user specified name for every element that appears more than once in a single alternative. The user specified name is added in front of the element between brackets, followed by a colon. Here's the modified grammar:

```
Productions /* grammar */
  exp = {constant} number | {add} add;
  add = l_par [left]:exp plus [right]:exp r_par;
```

When SableCC builds the set of AST classes, it does not give direct access to element variables. Instead, it provides accessor methods. Accessors are `get`*xxx* and `set`*xxx* methods, where the *xxx* is the name of the element. For our example grammar SableCC would generate the following classes:

```
abstract class Node {}
abstract class PExp extends Node{}
class AConstantExp extends PExp
{ TNumber getNumber(){...}
  void setNumber(TNumber number){...}; }
```

```
class AAddExp extends PExp
{ PAdd getAdd(){...}
  void setAdd(PAdd add){...} }
etc.
```

SableCC adds code in these accessors to further prevent the contruction of an invalid AST. For example, it enforces the "tree" property of the AST (e.g., a node cannot have more than one parent). It also provides a *parent()* method to all AST nodes. The programmer does not (and cannot!) set this reference. It is done automatically every time a `set`*xxx* method is called.

4: The visitor design pattern and SableCC

In order to build AST-walker classes that can be easily extended to add actions on specific nodes of the AST, SableCC uses an adaptation of the *visitor*[3] design pattern. In the following subsections, we revisit this design pattern, extend it, and then explain how it is used by SableCC to achieve our design goals.

4.1: The visitor design pattern revisited

A somewhat formal definition of the visitor design pattern has been given as *"a solution to the problem of adding operations on the elements of an object structure without changing the classes of the elements on which it operates"*[3]. In our view, and according to our experience in teaching it to novices in object-oriented programming, the name of this design pattern is not very intuitive. So, in reaction to this, we have developed new names for the constituents of this design pattern. We describe the design pattern as it is often used. We say that it is an object-oriented way of implementing a *switch* on the type of an element.

Here is a small example. We have three classes `Circle`, `Square` and `Rectangle`, all derived from class `Shape`. These classes are used in a program with a graphical user interface that shows circles, squares and rectangles to the user. Every time the user points an object with the mouse pointer and clicks on the mouse button, the method `Selected` is called with the selected object as a parameter.

In the method `Selected`, we would like to print a diagnostic message saying "an x was selected", where x is either circle, square or rectangle, depending on the type of the selected object. One way of doing this, in Java, would be to use the `instanceof` operator:

```
void Selected(Shape obj)
{ if(obj instanceof Circle)
    System.out.println("a circle was selected");
  else if(obj instanceof Square)
    System.out.println("a square was selected");
  else
    System.out.println("a rectangle was selected"); }
```

The problem with this approach, is that if we had 100 shapes, it could take up to 99 comparisons to find the shape of an object. Thus finding the shape of an object is $O(n)$ (worst case) where n is the number of available shape classes. We would like to do this operation in $O(1)$. One way of making this possible would be to define an abstract method `id` in class `Shape`, and override it in each shape class to return a user defined unique `ID`. Here's the code:

```
abstract class Shape
{ ...
  abstract int id(); }
class Circle extends Shape
{ ...
  static final int ID = 1;
  int id() { return ID; } }
```

```
class Square extends Shape
{ ...
  static final int ID = 2;
  int id() { return ID; } }
class Rectangle extends Shape
{ ...
  static final int ID = 3;
  int id() { return ID; } }
```

```
void Selected(Shape obj)
{ switch(obj.id())
  { case Circle.ID:
      System.out.println("a circle was selected"); break;
    case Square.ID:
      System.out.println("a square was selected"); break;
    case Rectangle.ID:
      System.out.println("a rectangle was selected"); break; } }
```

This approach has some problems. For example, it leaves the responsibility of keeping the IDs unique in the hands of the programmer and it is easy to forget to write the `break` statement. Additionally, the unique ID is redundant information. We can already identify the type of an object using the `instanceof` operator.

Fortunately, there exists an object oriented way of doing this switch statement, without defining a unique integer `ID` for each class. This method, uses inheritance and interfaces to achieve its goal.

The first step in this solution is to define a `Switch` interface as follows: for each class derived from `Shape`, we add a method called `case`*xxx* where *xxx* is the name of the class.

```
interface Switch
{ void caseCircle(Circle obj);
  void caseSquare(Square obj);
  void caseRectangle(Rectangle obj); }
```

This interface, will be implemented by every *switch statement class* used to tailor actions based on the type of a shape object.

The second step is to modify each shape class to implement an `apply` method. The apply method will call the appropriate method on a switch object passed in parameter.

```
abstract class Shape
{ ...
  abstract void apply(Switch sw); }
etc.
```

```
class Circle extends Shape
{ ...
  void apply(Switch sw)
  { sw.caseCircle(this); } }
```

Notice how the `Circle.apply` method calls the `caseCircle` method, passing a reference to `this` in parameter. This means that when `Circle`.`apply` is called, the `caseCircle` method of `sw` is called with the circle in parameter.

Finally, we can use an anonymous Java class to implement the object-oriented switch on the type of a shape. The following code demonstrates this:

```
void Selected(Shape obj)
{ obj.apply(new Switch()
  { void caseCircle(Circle obj)
    { System.out.println("a circle was selected"); }
    void caseSquare(Square obj)
    { System.out.println("a square was selected"); }
    void caseRectangle(Rectangle obj)
    { System.out.println("a rectangle was selected"); } }); }
```

This code is relatively similar to the previous implementation of the `Selected` method, but this time, we used the `apply` method on `obj`, instead of the `switch` keyword.

This is normally called the *visitor design pattern.* In the usual presentation of the visitor pattern, `apply` is called `accept`, and the `caseXxx` methods are called `visitXxx`. The anonymous class (or any class implementing `Switch`) is called a *visitor.*

4.2: Extending the visitor design pattern

The visitor pattern, as described in the previous section has some limitations. As stated in [3], the visitor pattern makes it hard to *add new element types to the visited structure, and visiting across class hierarchies is impossible.* In SableCC generated frameworks, we have made some modifications to the visitor design pattern to overcome these limitations and render the design usable in the context of evolving structures.

To do so, we redefine the `Switch` interface to be more generic.

```
interface Switch { }
```

This interface will be the ancestor of all switch interfaces in the framework. Then, we define a new `Switchable` interface:

```
interface Switchable { void apply(Switch sw); }
```

Every *switchable* class (like `Circle`, `Square`, and `Rectangle`) should implement this interface. Here's our modified code:

```
interface ShapeSwitch extends Switch
{ void caseCircle(Circle obj);
  void caseSquare(Square obj);
  void caseRectangle(Rectangle obj); }
abstract class Shape
  implements Switchable {...}
```

```
class Circle extends Shape
{ ...
  void apply(Switch sw)
  { ((ShapeSwitch)sw).caseCircle(this); } }
etc.
```

The introduction of the new `Switch` and `Switchable` interfaces allows us to add a new `Oval` shape without modifying any existing class:

```
interface ExtendedShapeSwitch
  extends Switch
{ void caseOval(Oval obj); }
```

```
class Oval extends Shape
{ ...
  void apply(Switch sw)
  { ((ExtendedShapeSwitch) sw).
       caseOval(this); } }
```

So, in short, to add a new shape (or a collection of new shapes), we define a new interface that extends `Switch` and includes a `caseXxx` method for each new shape. Then we make sure that each new shape class implements the `Switchable` interface. We can now write:

```
interface AllShapesSwitch extends ShapeSwitch, ExtendedShapeSwitch {}
void Selected(Shape obj)
{ obj.apply(new AllShapesSwitch()
  { void caseCircle(Circle obj)
    { System.out.println("a circle was selected"); }
    void caseSquare(Square obj)
    { System.out.println("a square was selected"); }
    void caseRectangle(Rectangle obj)
    { System.out.println("a rectangle was selected"); }
    void caseOval(Oval obj)
    { System.out.println("an oval was selected"); } }); }
```

4.3: SableCC and visitors

In each generated framework, SableCC defines an `Analysis` interface that extends `Switch`. This interface contains all the `caseXxx` methods for token classes (`TXxx` and `Token`) and alternative classes (`AXxx` and `Start`). Naturally, class `Node`, the ancestor of all AST classes, implements `Switchable`.

In addition, SableCC implements a utility class called `AnalysisAdapter` that implements `Analysis` and provides a default implementation for all methods. Unless a method is overridden, it will call the `defaultCase` method. This makes it possible to implement a switch by extending `AnalysisAdapter`, specifying only relevant cases and catching all unspecified cases in a *default* handler.

Here's an example of the implementation of a *switch adapter* for our (extended) shape example:

```
class ExtendedShapeSwitchAdapter implements ExtendedShapeSwitch
{ void caseCircle(Circle obj) { defaultCase(obj); }
  void caseSquare(Square obj) { defaultCase(obj); }
  void caseRectangle(Rectangle obj) { defaultCase(obj); }
  void caseOval(Oval obj) { defaultCase(obj); }
  void defaultCase(Shape obj) { } }
```

We could use it to detect circles:

```
void Selected(Shape obj)
{ obj.apply(new ExtendedShapeSwitchAdapter()
  { void caseCircle(Circle obj)
    { System.out.println("a circle was selected"); }
    void defaultCase(Shape obj)
    { System.out.println("The selected object is not a circle"); } }); }
```

4.4: AST walkers

One of the basic functionalities required to work on an AST is visiting its nodes. A tree-walker class is a class that will visit all the nodes of an AST in a predefined order. By default, SableCC provides two tree-walker classes. One that visits the nodes in a normal depth-first traversal. The second visits the AST nodes in the reverse depth-first traversal.

To implement the tree walkers, SableCC uses the extended visitor design pattern presented in earlier sections.

Here is how SableCC implements tree walkers. It uses a set of recursive methods, like this:

```
class DephtFirstAdapter extends AnalysisAdapter
{ void caseXxx(Xxx node)
  { node.getYyy.apply(this);    // first child of Xxx
    node.getZzz.apply(this); }  // second child of Xxx
  void caseAaa(Aaa node)
  { node.getBbb.apply(this); }  // first child of Aaa
etc.
```

For every *alternative* of every *production* of the compiled grammar, SableCC adds a `caseXxx` method in the walker class. Each `caseXxx` method calls the `apply` method on every *element* of the alternative.

To implement actions, a programmer defines a new class that extends one of the two provided tree-walker classes. He then overrides the `caseXxx` method for all interesting nodes.

5: A mini-BASIC interpreter

In this section, we demonstrate the simplicity of the SableCC framework by developing a mini-BASIC interpreter. A complete version of this interpreter requires less than 250 lines of documented Java code. In figure 3, we show the grammar specification of mini-BASIC. It contains a package declaration, a list of helper regular expressions to simplify the writing of more complex regular expression, a list of tokens with their definition, a list of tokens that are ignored by the parser, and a list of productions that describe the grammar of the language.

The language is a simple BASIC with integer variables. It contains a decision statement in the form of an IF..THEN..ELSE ..ENDIF construct, and a loop statement in the form of a FOR..TO..NEXT. Input and output are handled by the READ, PRINT and PRINTLN statements. And finally, the language contains an assignment statement.

In order to build an interpreter for this language, we follow the steps mentioned in section 2.1. So initially, we create a text file containing the grammar specification in of figure 3.

The next step is simply to launch SableCC on the specification file. At the shell prompt we type:

```
$> java SableCC minibasic.grammar
....
$>
```

The next step is to create working classes. In our case, we have a single class to create: Interpreter.java. To simplify writing our interpreter, we create six methods to store and retrieve values.

- int get/setVariable(String name) that gets and sets the integer value of variable *name*. (Initially, all variables default to zero.)
- int get/setIntValue(Node node) that gets and sets the integer value associated with a node.
- int get/setBoolValue(Node node) that gets and sets the boolean value associated with a node.

We create the `Interpreter.java` class inheriting from the `DepthFirstAdapter` tree-walker class, as shown in figure 3. Then, we override the appropriate method to add the code to handle each kind of statement.

For example, to interpret an IF statement, we override the `caseAIfStatement` method. An IF statement evaluation proceeds as follows:

1. Evaluate the `Condition` subtree
2. If condition is true evaluate the `Statements` subtree
3. Else evaluate the `OptionalElse` subtree.

Next, we implement the `FOR` loop interpretation. The process is again quite easy:

1. Evaluate the `FromExp` and the `ToExp` subtrees.
2. Execute a real Java loop from `FromExp` to `ToExp`.
3. In each iteration, Assign the appropriate value the loop variable. Then, evaluate the `Statements` subtree.

We continue this way until we have covered all the statements in the grammar. Some statement might require some special handling, like the READ statement that can cause a I/O ERROR.

The next step is to implement the interpretation of conditions and expressions. The difference between expressions and statements is that an expression returns a value. More specifically, evaluating an expression associates the result of the expression with the `Expression` node. This is done using the setInt/BoolValue(Node, value).

For example, to evaluate a `less_than` condition, we override the `caseALessThanCondition` method. Then we proceed with the evaluation as follows:

1. Evaluate the `left` and `right` subtrees.
2. setBoolValue(node, if getIntValue(`left`) < getIntValue(`right`) then *true* else *false*)

Figure 3 shows how to interpret the most important or difficult conditions, expressions and values. This is all what is needed in the `Interpreter` class.

The next step is to create a Main class with a main method to initiate the process. Here's a simplified version:

```
public class Main
{ public static void main(String[] arguments)
  { try
    { Lexer lexer = new Lexer(...(File(arguments[0])...);
      Parser parser = new Parser(lexer);
      Node ast = parser.parse();     // build the AST
      ast.apply(new Interpreter()); // Call the interpreter
    } catch(Exception e)
    { System.out.println(e); } } }
```

We save this code in a file called `Main.java`. Then we are ready to build our interpreter and test it:

```
$> javac Main.java
...
$> java Main file.bas
TYPE A NUMBER? 3
9/3 = 3
TYPE A NUMBER? 0
DIVIDE BY ZERO ERROR IN LINE 5
```

6: Other applications

SableCC has been used in multiple projects. We list some of them here:

- A project to build a Java 1.02 front-end compiler, handling Unicode characters and escapes.
  To realize this project, we simply took the Java grammar in [4] that was accepted directly by SableCC (after adding names to alternatives). In order to handle all cases in the lexer, we have buit two lexers with SableCC. We fed the output of the first to the second.
- SableCC has been used to generate a newer version of itself.
  The current version of SableCC has been buit using a previous version to generate its framework. This allowed us to take advantage of all the improvements that were incorporated in the framework to add new functinality.
- We have built a SIMPLE C compiler and implemented a state-of-the-art linear time points-to analysis of programs[11].
  This allowed us to assess the suitability of using SableCC to implement modern compiler analyses. The code of the analysis is almost identical to the pseudo-code of the algorithm.
- Some McGill University undergraduate students used SableCC to build a compiler that generates Internet Common Gateway Interface (CGI) programs.

7: Related work

The most widely used compiler compilers today, fall into two main families: Lex/YACC and PCCTS. We will discuss both tool families and look at their most popular Java imple-

```
Package minibasic;
Helpers
 cr = 13;                  lf = 10;
 letter = [’A’..’Z’];
 digit = [’0’..’9’];
 not_cr_lf = [[32..127] - [cr + lf]];
Tokens
 if = ’IF’;                then = ’THEN’;
 else = ’ELSE’;            endif = ’ENDIF’;
 for = ’FOR’;              to = ’TO’;
 next = ’NEXT’;            read = ’READ’;
 println = ’PRINTLN’; print = ’PRINT’;
 assign = ’:=’;            equal = ’=’;
 greater_than = ’>’;  less_than = ’<’;
 plus = ’+’;               minus = ’-’;
 mult = ’*’;               div = ’/’;
 l_par = ’(’;              r_par = ’)’;
 mod = ’MOD’;
 number = digit+;
 string = ’"’ [not_cr_lf - ’"’]* ’"’;
 identifier = letter (letter | digit)*;
 new_line = cr | lf | cr lf;
 blank = ’ ’*;
Ignored Tokens
  blank;
Productions
 statements =
  {list}  statement statements | {empty} ;
 statement =
  {if} if condition then [nl1]:new_line
       statements optional_else
       endif [nl2]:new_line |
  {for} for identifier assign
        [from_exp]:expression to
        [to_exp]:expression [nl1]:new_line
        statements
        next [nl2]:new_line |
  {read} read identifier new_line |
  {print_exp} print expression new_line |
  {print_str} print string new_line |
  {println} println new_line |
  {assignment} identifier assign expression
               new_line;
 optional_else =
  {else}  else new_line statements | {empty} ;
 condition =
  {less_than} [left]:expression less_than
              [right]:expression |
  {greater_than} [left]:expression greater_than
                 [right]:expression |
  {equal} [left]:expression equal
          [right]:expression;
 expression =
  {value} value |
  {plus} [left]:value plus [right]:value |
  {minus} [left]:value minus [right]:value |
  {mult} [left]:value mult [right]:value |
  {div} [left]:value div [right]:value |
  {mod} [left]:value mod [right]:value;
 value =
  {constant}  number |
  {identifier} identifier |
  {expression} l_par expression r_par;
```

```
public class Interpreter extends DepthFirstAdapter
{public void caseAIfStatement(//IF
                 AIfStatement node)
 {node.getCondition().apply(this);//eval
   if(getBoolValue(node.getCondition()))
    node.getStatements().apply(this);//eval
   else node.getOptionalElse().apply(this);}//eval
 public void caseAForStatement(//FOR
                 AForStatement node)
 {node.getFromExp().apply(this);//eval
  node.getToExp().apply(this);//eval
  int from = getIntValue(node.getFromExp());
  int to   = getIntValue(node.getToExp());
  for(int i = from; i <= to; i++)
  {setVariable(node.getIdentifier().getText(),i);
   node.getStatements().apply(this);}}//eval
 public void caseAReadStatement(//READ
                  AReadStatement node)
 {System.out.print("? ");//display prompt
  try
  {String s = in.readLine();//read
   int value = Integer.parseInt(s);//parse
   setVariable(node.getIdentifier().getText(),
     value);//set variable
  }catch(IOException e)//IO ERROR
  {error("I/O ERROR IN LINE "+
         node.getRead().getLine());
  }catch(NumberFormatException e)//NUMBER ERROR
  {error("NUMBER FORMAT ERROR IN LINE "+
         node.getRead().getLine());}}
 public void caseAPrintExpStatement(//PRINT
                 APrintExpStatement node)
 {node.getExpression().apply(this);//eval
  out.print(getIntValue(node.getExpression()));}
 public void caseALessThanCondition(//left<right
                 ALessThanCondition node)
 {node.getLeft().apply(this);//eval
  node.getRight().apply(this);//eval
  setBoolValue(node,getIntValue(node.getLeft())<
               getIntValue(node.getRight()));}
 public void caseADivExpression(//left/right
                 ADivExpression node)
 {try
  {node.getLeft().apply(this);//eval
   node.getRight().apply(this);//eval
   setIntValue(node,getIntValue(node.getLeft())/
               getIntValue(node.getRight()));
  }catch(ArithmeticException e)//DIVIDE BY ZERO
  {error("DIVIDE BY ZERO ERROR IN LINE "+
         node.getDiv().getLine());}}
 public void caseAConstantValue(//Number
                 AConstantValue node)
 {try
  {int value = Integer.parseInt(//parse
    node.getNumber().getText());
   setIntValue(node,value);
  }catch(NumberFormatException e) //NUMBER ERROR
  {error("NUMBER FORMAT ERROR IN LINE "+
         node.getNumber().getLine());}}
 public void caseAIdentifierValue(//Identifier
                 AIdentifierValue node)
 {setIntValue(node,
   getVariable(node.getIdentifier().getText()));}}
```

Figure 3. `minibasic.grammar` **and** `Interpreter.java` mentations.

These tools benefit from a large user base. Therefore, it can be relatively easy to find a grammar for most languages to use with these tools, that has already been tested.

7.1: Lex/YACC

Lex[9] and YACC[7] (*Yet Another Compiler Compiler*) are a pair of tools that can be used together to generate a compiler or its front-end. Many variations on these tools are in use today.

A version of Lex has been ported to Java. It is called JLex. It has been developed by Elliot Joel Berk, a student of the Department of Computer Science, Princeton University. It is quite similar in functionality to Flex.

A Java version of YACC is called CUP (*Constructor of Useful Parsers*). It has been developed by Developed by Scott E. Hudson, Graphics Visualization and Usability Center, Georgia Institute of Technology. It is very similar to YACC, but actions are written in the Java language.

The pair JLex/CUP is most suitable to build one pass compilers. This is achieved by inserting actions at different points of the grammar specification. There is no special support for building abstract syntax trees. So writing an interpreter or a multiple pass compiler can be a long and error prone process

7.2: PCCTS

PCCTS stands for *Purdue Compiler Construction Tool Set.* It has been developed mainly by Terence Parr. Originally, PCCTS has been written in the C++ language to generate compilers written in C++. Lately, PCCTS 1.33 has been ported to Java and renamed ANTLR2.xx[6].

A very similar tool has been developed in parallel by Sun Microsystems inc., called JavaCC[5]. There are only small differences between these two products. JavaCC has a better support for building abstract syntax trees (AST) than ANTLR.

These tools are used a technique to parse the input called LL(K) with semantic predicates. Not unlike the LEX/YACC family of tools, the programmer is allowed to add actions into the specification. In fact, the parsing power of semantic predicates cannot be obtained without the use of actions.

JavaCC has options to automatically build an AST. There are two flavors of automatic ASTs. In the first form, there is a single Node class. The type of a node is obtained by querying the Node.getType() method. Unlike SableCC, there are no specific types for different alternatives of a production. Child nodes are accessed using a getNode(int child_number) method.

In the second flavor, the AST contains one class per production. Again, there are no types for alternatives, and children are accessed by number.

Here are some consequences of these design decisions:

1. There is no general way of knowing the current alternative. For example, given a Statement node, it is only by looking for specific child nodes (like getNode(1) = 'FOR' or 'IF'), that we can find if we have a For or an If Statement. To minimize the difficulty of this problem, the concept of tree parsers is introduced. But this does not resolve the problem completely.

2. The integrity and the correctness of the AST is left in the hands of the programmer. There will be no warning if a transformation on the AST results in a degenerated tree. Such bugs are extremely difficult to track. They may result in a null pointer exception or some other error condition in unrelated code thousands of instructions after the transformation has occurred. This is comparable to C and C++ array out of bound problems.
3. Minor modifications to the grammar can cause problems that are hard to fix. This happens when an additional element is added to an alternative. For example, if an element is added to the beginning of an alternative, then all the code referring the getNode(1) should be changed to getNode(2), but only for this specific alternative. (And we have already seen that knowing the alternative is not always trivial).

8: Conclusion

Writing a small compiler or interpreter, or just writing a parser to read some formatted text has become a common task. Compiler compilers are tools used by programmers to accomplish these tasks. As the Java language appeared on the internet and gained popularity, existing compiler compilers have been ported to Java.

In this paper, we have introduced SableCC, a new compiler framework. We explained our main design decisions in pursuit of building maintainable compilers in the Java language. The framework automatically builds strictly typed abstract syntax trees and tree-walker classes. We presented an extended version of the visitor design pattern which enables the implementation of actions on the nodes of the abstract syntax tree.

SableCC is freely available to download at *http://www.sable.mcgill.ca.*

References

[1] ANSI. HL7 version 2.3 (ballot draft 3), 1997. World-Wide Web page URL: http://www.mcis.duke.edu/standards/HL7/pubs/version2.3/.

[2] J.W. Backus, R. J. Beeber, S. Best, R. Goldberg, L.M. Haibt, H.L. Herrick, R.A. Nelson, D. Sayre, P.B. Sheridan, H. Stern, I Ziller, R. A. Hughes, and R. Nutt. The FORTRAN automatic coding system. *Western Joint Computer Conference*, 1957.

[3] E. Gamma and R. Helm. *Design Patterns, Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[4] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison-Wesley, 1996.

[5] Sun Microsystems Inc. JavaCC, 1997. World-Wide Web page URL: http://www.suntest.com/.

[6] Magelang Institute. ANTLR 2.0, 1997. World-Wide Web page URL: http://www.antlr.org/.

[7] S. C. Johnson. YACC - yet another compiler compiler. Technical Report Computing Science Technical Report 32, AT&T Bell Laboratories, Murray Hill, N.J., 1975.

[8] D. E. Knuth. *The TeX book.* Addison Wesley, 1984.

[9] M. E. Lesk. Lex - a lexical analyzer generator. Technical Report Computing Science Technical Report 39, AT&T Bell Laboratories, Murray Hill, N.J., 1975.

[10] T. LinDholm and F.Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, 1997.

[11] Bjarne Steensgaard. Points-to analysis in almost linear time. In *In Proceedings of the Twentythird Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 32–41, 1996.

[12] W3C. HTML 4.0 specification, 1997. World-Wide Web page URL: http://www.w3.org/TR/REC-html40/.