Learning Compiler Construction by Example

José de Oliveira Guimarães
jose@dc.ufscar.br

March 26, 2003

Contents

This report explains the code of 10 compilers. It begins with a very small grammar and proceeds towards a almost complete compiler of a language similar to Pascal.

1 Compiler 1

We will build a compiler for the grammar

```
Expr ::= '(' oper Expr Expr ')' | number
oper ::= '+' | '-'
number ::= '0' | '1' | ... | '9'
```

In fact, we will code only a syntactical analyzer since there will not be code generation. Anyway, it will be called a compiler.

The compiler has two classes, `Compiler` and `Main`. The last one is the class of method `main`, where the program execution starts. In this method, the input to the compiler is given as an array of characters — see Figure 1.

The compiler itself is in class `Compiler`, which has method `compile` that compiles the input.

In `Compiler` there are:

- a method for each non-terminal of the grammar. See methods `expr`, `oper`, and `number` below;
- a method `error` that prints an error message pointing where the error occurred;
- a method `nextToken` which skips white spaces and puts the next character in instance variable `token`. `nextToken` is repeatedly called and `token` assumes `'('`, `'-'`, `'('`, `'+'`, ...

Method `nextToken` is then the lexical analyzer. Methods `expr`, `oper`, and `number` compose the syntactical analyzer.

```
public class Compiler {

    public void compile( char []p_input ) {
        input = p_input;
        tokenPos = 0;
        nextToken();
        expr();
        if ( token != '\0' )


public class Main {
    public static void main( String []args ) {
        char []input = "(-  (+ 5 4) 1)".toCharArray();

        Compiler compiler = new Compiler();

        compiler.compile(input);
    }
}
```

Figure 1: The main class of the first compiler

```
      error();
}

private void expr() {
    if ( token == '(' ) {
        nextToken();
        oper();
        expr();
        expr();
        if ( token == ')' )
          nextToken();
        else
          error();
    }
    else
      number();
}

private void number() {
    if ( token >= '0'  && token <= '9' ) {
        nextToken();
    }
    else
      error();
}

private void oper() {
    if ( token == '+' || token == '-' )
      nextToken();
    else
      error();
}

private void nextToken() {
    while (  tokenPos < input.length && input[tokenPos] == ' ' )
      tokenPos++;
    if ( tokenPos < input.length ) {
      token = input[tokenPos];
      tokenPos++;
    }
    else
      token = '\0';
}

private void error() {
    if ( tokenPos == 0 )
```

```
        tokenPos = 1;
      else
        if ( tokenPos >= input.length )
          tokenPos = input.length;

      String strInput = new String( input,
               tokenPos - 1, input.length - tokenPos + 1 );
      String strError = "Error at \"" + strInput + "\"";
      System.out.println( strError );
      throw new RuntimeException(strError);
    }

    private char token;
    private int  tokenPos;
    private char []input;

}
```

Each method of the syntactical analyzer correspond to one grammar rule and is responsible for analyzing the input according to that rule. Then, `number` has a rule

```
number ::= '0' | '1' | ... | '9'
```

and therefore method `number` should test if the input, given by variable `token`, is one of the digits. Remember `token` is extracted from the input by `nextToken` by skipping blanks. Can `token` be any thing other than a digit in the rule `number` ? The answer is no. Therefore method `number` must signal an error if `token` is not a digit:

```
    private void number() {
        if ( token >= '0'  && token <= '9' ) {
            nextToken();
        }
        else
          error();
    }
```

If a digit is recognized, `nextToken` is called to get the next piece of input. The same observations apply to rule

```
oper ::= '+' | '-'
```

and method `oper`:

```
    private void oper() {
        if ( token == '+' || token == '-' )
          nextToken();
        else
          error();
    }
```

The rule

```
Expr ::= '(' oper Expr Expr ')' | number
```

is a little bit more complex than the other two, for there is a `|` meaning choice. If we want to expand `Expr` to produce the input "`1`", we choose the second rule, `Expr ::= number`. See the derivation below.

`Expr` $\Longrightarrow$ `number` $\Longrightarrow$ `1`

The input "`1`" is recognized by this grammar because we can derive it from the start symbol `Expr`.

Suppose the input is "`(+ 5 4)`". Now we have to choose rule

`Expr ::= '(' oper Expr Expr ')'`

to derive `Expr`:

`Expr` $\Longrightarrow$ `'(' oper Expr Expr ')'`

This was the only way to obtain the `'('` that matches the first character of the input "`(+ 5 4)`". The other rule, `Expr ::= number`, would result in a number. The grammars we will work with will have the property that, whenever a rule `A ::=` α can derive an expression beginning with terminal `w`, `A ::=` β cannot derive expression beginning with `w`. If that were allowed, we would not know which rule to use in the expansion of `A` when the next terminal of the input is `w`. That is, if `token` contains `w` and we are deriving `A`, should we use `A ::=` α or `A ::=` β ? No doubt will ever exist.

In the example, the first rule `Expr ::= '(' oper Expr Expr ')'` produces an expression beginning with `'('` and the second rule `Expr ::= number` produces a digit. If the input is "`(+ 5 4)`" and we want to expand `Expr`, we test if the first rule can produce an expression that starts with `'('`. It can and we choose this rule. Then in method `expr` there will be an `if` to decide which rule to use based in the input. And the next character of the input was put at instance variable `token`:

```
private void expr() {
    if ( token == '(' ) {
        nextToken();
        oper();
        expr();
        expr();
        if ( token == ')' )
          nextToken();
        else
          error();
    }
    else
      number();
}
```

whenever a terminal appears at the grammar, there is a code of the type

```
if ( token == terminal )
   nextToken();
else
   error();
```

in the compiler. In the beginning of `expr`, there is something similar. Once `'('` was recognized, `nextToken` is called.

Whenever a non-terminal appears in the grammar, there is a call to the corresponding method at the compiler. See the rule `Expr ::= '(' oper Expr Expr ')'` and the calls to `oper`, `expr`, and `number` in method `expr`.

2 Compiler 2

This compiler generates code in language C for the expression. Method `expr` generates code for the grammar rule `Expr` and method `number` for the rule `number`. A sole number, as

```
1
```

will be analyzed by method `number`,

```
private void number() {
    if ( token >= '0'  && token <= '9' ) {
        System.out.print(token);
        nextToken();
    }
    else
      error();
}
```

which just generates the number itself, `1`. The generated code is printed in the standard output.

A composite expression

```
(+ 1 2)
```

generates

```
(1 + 2)
```

in C. Method `expr` analyzes this expression as follows. First `'('` is printed — a composite expression begins and ends with `'('` and `')'`. Then a recursive call generates code for the left expression. The operator is printed. A recursive call generates code for the right expression. `')'` is printed. See method `expr` below.

```
private void expr() {
    if ( token == '(' ) {
        nextToken();
        char op = oper();
        System.out.print("(");
        expr();
        System.out.print(op);
        expr();
        System.out.print(")");
        if ( token == ')' )
          nextToken();
        else
          error();
    }
    else
      number();
}
```

Method `oper` was modified to return the operator so `expr` can correctly generate code:

```
private char oper() {
```

```
        char op = token;
        if ( token == '+' || token == '-' ) {
          nextToken();
          return op;
        }
        else {
          error();
            // should never execute since error terminates the program
          return '\0';
        }
    }
```

3 Compiler 3

This compiler generates assembly code for the expression. An assembly of a virtual machine is used. Methods `expr` and `number` generate code that will calculate the value of an expression or a number and push this value into the machine stack. In the calculation of an expression, when the assembly code is executed, other stack positions may be used. But the result will be put at the top of the stack.

Then method `number`, which analyzes a sole number, simply pushes the number into the stack; when `number` analyzes

`1`

it generates

```
mov R0, 1
push R0
```

See the method below.

```
    private void number() {
        if ( token >= '0'  && token <= '9' ) {
            System.out.println("mov R0, " + token );
            System.out.println("push R0");
            nextToken();
        }
        else
          error();
    }
```

Method `expr` may analyze code

`(+ 1 2)`

The generated code pushes `1`, then pushes `2`, and adds both numbers, pushing the result into the stack:

```
mov R0, 1    // generated by number()
push R0
mov  R0, 2   // generated by number()
push R0
pop R1       // generated by expr()
```

```
pop R0
add R0, R1
push R0
```

Method `expr` calls itself recursively to generate code for the first and second expressions, which are `1` and `2` in this case. See method `expr` below.

```
    private void expr() {
        if ( token == '(' ) {
            nextToken();
            char op = oper();
            expr();
            expr();
            System.out.println("pop R1");
            System.out.println("pop R0");
            switch (op) {
                case '+' :
                  System.out.println("add R0, R1");
                  break;
                case '-' :
                  System.out.println("sub R0, R1");
                  break;
            }
            System.out.println("push R0");
            if ( token == ')' )
              nextToken();
            else
              error();
        }
        else
          number();
    }
```

4 Compiler 4

This compiler evaluates the value of the expression. That is, at the end of the syntactic analysis, the compiler will print the value of the expression, this value being calculated during the analysis.

Methods `expr` and `number` return the value of the input associated to them. So, when `expr` analyzes "`(+ 1 2)`", it returns `3`. When `number` analyzes "`3`", it returns the integer `3`.

Method `number` calculates the integer value associated to a digit by subtracting `'0'` from it. That is, `1`, integer, is `'1' - '0'`. See `token - '0'` in the code below.

```
    private int number() {
        int result = 0;

        if ( token >= '0'  && token <= '9' ) {
            result = token - '0';
```

```
            nextToken();
        }
        else
          error();
        return result;
    }
```

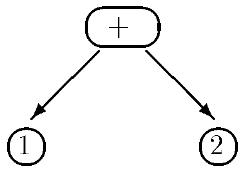
Method `expr` recursively calls itself to calculate the left and right expressions before adding the two or subtracting one from the other. See the method below.

```
    private int expr() {
        if ( token == '(' ) {
            nextToken();
            char op = oper();
            int left  = expr();
            int right = expr();
            int result = 0;
            switch (op) {
                case '+' :
                  result = left + right;
                  break;
                case '-' :
                  result = left - right;
                  break;
            }
            if ( token == ')' )
              nextToken();
            else
              error();
            return result;
        }
        else
          return number();
    }
```

5 Compiler 5

This compiler builds an Abstract Syntax Tree (AST) for the input expression and generates code through it.

An Abstract Syntax Tree is a data structure that represents the program structure in an abstract form. Here "the program" is the input to be compiled. Then the program "`(+ 1 2)`" is translated to



The balls are objects, Java objects. The balls that hold numbers (`1` or `2`) are objects of class `NumberExpr`. As the name says, this class represents a number used in an expression. This class has only one instance variable which is used to store the number:

```
public class NumberExpr extends Expr {
    public NumberExpr( char n ) {
        this.n = n;
    }

    public void genC() {
        System.out.print(n);
    }
    private char n;
}
```

In the Figure, the ball with a `+` is an object of class `CompositeExpr`. This class represents an expression with an operator, a composite expression. Clearly an object of this class needs to store the operator, a pointer to the left expression, and another to the right expression. This can be checked in the class itself:

```
public class CompositeExpr extends Expr {
    public CompositeExpr( Expr pleft, char poper, Expr pright ) {
        left = pleft;
        oper = poper;
        right = pright;
    }
    public void genC() {
        System.out.print("(");
        left.genC();
        System.out.print(" " + oper + " ");
        right.genC();
        System.out.print(")");
    }
    private Expr left, right;
    private char oper;
}
```

Classes `CompositeExpr` and `NumberExpr` inherit from abstract class `Expr`:

```
abstract public class Expr {
    abstract public void genC();
}
```

Why ? Because the `left` and `right` pointers of `CompositeExpr` can refer to both a `NumberExpr` or a `CompositeExpr`. In "`(- (+ 1 2) 3)`" the `left` variable of the topmost object (a `CompositeExpr` object) points to a `CompositeExpr` object representing "`(+ 1 2)`". And the `right` variable points to a `NumberExpr` object representing "3". Then the type of `left` and `right` of `CompositeExpr` should be such that it can point to both a `CompositeExpr` or a `NumberExpr`. The only way to achieve that is to create a common superclass. This is possible because a variable of a superclass can point to an object of a subclass. The assignments

```
Expr left;
left = new CompositeExpr(...);
left = new NumberExpr('3');
```

are legal. Then the constructor of `CompositeExpr`,

```
public CompositeExpr( Expr pleft, char poper, Expr pright )
```

can accept, in its first parameter, either a `CompositeExpr` or a `NumberExpr` object.

The AST for the expression "`(+ 1 2)`" can be created by the Java code

```
Expr expr;
expr = new CompositeExpr( new NumberExpr('1'),
                          '+',
                          new NumberExpr('2')
                          );
```

class `Expr` has an abstract method `genC` whose meaning is "generate C code for the expression". `genC` is redefined in both subclasses. In `Number`, it just prints the number. In `CompositeExpr`, it prints `'('`, calls the method `genC` of the `left` variable, prints the operator, calls the method `genC` of the `right` variable, and prints `')'`. Note `left` has type `Expr` and there is no object of class `Expr` since this class is abstract. But `Expr` has subclasses and `left` points to an object of `CompositeExpr` or `NumberExpr`. Since "`left.genC()`" was used, it was necessary to define an abstract method `genC` at `Expr`.

Method `number` and `expr` of the syntactic analyzer will now return objects of the AST they build. Method `number` builds an object of `NumberExpr`:

```
private NumberExpr number() {
    NumberExpr e = null;

    if ( token >= '0'  && token <= '9' ) {
        e = new NumberExpr(token);
        nextToken();
    }
    else
      error();
    return e;
}
```

Method `expr` has to return an `Expr` since it can return either a `NumberExpr` or a `CompositeExpr`:

```
private Expr expr() {
    if ( token == '(' ) {
        nextToken();
```

```
            char op = oper();
            Expr e1 = expr();
            Expr e2 = expr();
            CompositeExpr ce = new CompositeExpr(e1, op, e2);
            if ( token == ')' )
              nextToken();
            else
              error();
            return ce;
        }
        else
          return number();
    }
```

Method `expr` calls itself recursively to get the left and right subtrees before creating an object of `CompositeExpr`. The result will be an object of `CompositeExpr` whose variable `left` will point to an object got in the first call to `expr`. Similar comments apply to variable `right`.

Note method `compile` also returns an object of the AST. In fact, it returns the object of the AST that represents the program. See the code below.

```
    public Expr compile( char []p_input ) {
        input = p_input;
        tokenPos = 0;
        nextToken();
        Expr e = expr();
        if ( token != '\0' )
          error();
        return e;
    }
```

Method `main` calls `compile` and then calls `genC` on the object of the AST that represents the program:

```
public class Main {
    public static void main( String []args ) {
        char []input = "( + (-  (+ 5 4) 1)  7)".toCharArray();

        Compiler compiler = new Compiler();

        Expr expr = compiler.compile(input);
        expr.genC();
    }
}
```

C code for the whole program, which is an expression, is generated.

6 Compiler 6

This compiler has variable[1] declarations preceding the expression. A typical program would be

```
a = 1 b = 3 : (- (+ a 2) b)
```

The new grammar is:

```
Program ::= VarDecList ':' Expr
VarDecList ::=  | VarDec VarDecList
VarDec ::= Letter '=' Number;
Expr::=  '(' oper Expr  Expr ')' | Number | Letter
Oper ::= '+' | '-' | '*' | '/'
Number ::= '0'| '1' | ... | '9'
Letter ::= \'A' | 'B'| ... | 'Z'| \'a'| 'b' | ... | 'z'
```

Of course, there are methods for rules `Program`, `VarDecList`, `VarDec`, and `Letter`. `VarDecList` is a list of zero or more `VarDec`, which starts with a letter. Then `VarDecList` should derive using the second rule,

```
VarDecList ::= VarDec VarDecList
```

while the next token is a letter. This is reflected in the code of method `varDecList`:

```
private Vector varDecList() {
      /* See how the repetition in the grammar reflects in the
         code. Since VarDec always begin with a letter, if token
         is NOT a letter, then VarDecList is empty and null is
         returned */
    if ( ! Character.isLetter(token) )
      return null;
    else {
        Vector arrayVariable = new Vector();
        while ( Character.isLetter(token) )
          arrayVariable.addElement( varDec() );
        return arrayVariable;
        }

}
```

Method `program`,

```
private Program program() {
    Vector arrayVariable = varDecList();
    if ( token != ':' ) {
        error();
        return null;
    }
    else {
        nextToken();
        Expr e = expr();
```

[1] Since the values of the variables cannot be modified, they are in fact constants.

```
            return new Program( arrayVariable, e );
        }
    }
```

stores the variables in a `Vector` and returns an object of `Program` with the variables and the expression.

Method `varDec` returns a `Variable` object containing the name and value of the variable:

```
    private Variable varDec() {
        char name = letter();
        if ( token != '=' ) {
          error();
          return null;
        }
        else {
          nextToken();
          NumberExpr n = number();
          return new Variable( name, n.getValue() );
        }
    }
```

Method `letter` tests if `token` is a letter:

```
    private char letter() {
       if ( ! Character.isLetter(token) ) {
          error();
          return '\0';
       }
       else {
         char ch = token;
         nextToken();
         return ch;
       }
    }
```

Method `expr` now has to test if `token` is `'('`, a number, or a letter:

```
    private Expr expr() {
        if ( token == '(' ) {
            nextToken();
            char op = oper();
            Expr e1 = expr();
            Expr e2 = expr();
            CompositeExpr ce = new CompositeExpr(e1, op, e2);
            if ( token == ')' )
              nextToken();
            else
              error();
            return ce;
```

```
        }
        else
            // note we test the token to decide which production to use
          if ( Character.isDigit(token) )
            return number();
          else
            return new VariableExpr(letter());
    }
```

Note a variable inside an expression — see method `expr` above — results in an object of class `VariableExpr`, not `Variable`. Class `Variable` is used to represent declarations of variables, not use of a variable in an expression. See the code of classes `VariableExpr` and `Variable` below. Note method `genC` of both classes show `VariableExpr` generates code for a variable in an expression and `Variable` generates code for a variable declaration.

```
public class VariableExpr extends Expr {
    public VariableExpr( char name ) {
        this.name = name;
    }

    public void genC() {
        System.out.print(name);
    }

    private char name;
}

public class Variable {
    public Variable( char name, int value ) {
        this.name = name;
        this.value = value;
    }

    public void genC() {
        System.out.println( "int " + name + " = " + value + ";" );
    }
    private char name;
    private int value;
}
```

Class `Program` generates code for the main program in C, which will contain the variable declarations and the expression:

```
public class Program {
    public Program( Vector arrayVariable, Expr expr ) {
        this.arrayVariable = arrayVariable;
        this.expr = expr;
    }
```

```
    public void genC() {
        System.out.println("#include <stdio.h>\n");
        System.out.println("void main() {");
        if ( arrayVariable != null ) {
              // generate code for the declaration of variables
            Enumeration e = arrayVariable.elements();
            while ( e.hasMoreElements() )
              ((Variable ) e.nextElement()).genC();
        }

          // generate code for the expression
        System.out.print("printf(\"%d\\n\", ");
        expr.genC();
        System.out.println(" );\n}");
    }

    private Vector arrayVariable;
    private Expr expr;
}
```

7 Compiler 7

This compiler uses the same grammar as compiler 6. It evaluates the value of the expression. It uses a symbol table to keep the value of each variable. The symbol table is a simple hash table.

The symbol table is set up in method compile:

```
    public Program compile( char []p_input ) {
        input = p_input;
        tokenPos = 0;
        symbolTable = new Hashtable();
        nextToken();
        Program result = program();
        if ( token != '\0' )
          error("End of file expected");
        return result;
        }
```

In `varDec`, the name and value of the variable are inserted in the symbol table:

```
    private Variable varDec() {
        char name = letter();
        if ( token != '=' ) {
          error("= expected");
          return null;
        }
        else {
          nextToken();
```

```
        NumberExpr n = number();
          // semantic analysis
        if ( symbolTable.put( name + "", n ) != null )
          error("Variable " + name + " has already been declared");
        return new Variable( name, n.getValue() );
      }
  }
```

Later on, we will be able to retrieve the value of a variable if we give its name.

Method `expr` no more returns an object of `VariableExpr` in its last return statement. It instead calls `letterExpr`:

```
  private Expr expr() {
      if ( token == '(' ) {
          nextToken();
          char op = oper();
          Expr e1 = expr();
          Expr e2 = expr();
          CompositeExpr ce = new CompositeExpr(e1, op, e2);
          if ( token == ')' )
            nextToken();
          else
            error(") expected");
          return ce;
      }
      else
          // note we test the token to decide which production to use
        if ( Character.isDigit(token) )
          return number();
        else
          return letterExpr();
  }
```

`letterExpr` recognizes a variable. It retrieves the value of the variable from the symbol table:

```
  private Expr letterExpr() {
      NumberExpr e;
      char ch = letter();
        // semantic analysis
        // was the variable declared ?
      e = (NumberExpr ) symbolTable.get( ch + "" );
      if ( e == null ) {
        error("Variable was not declared");
        return null;
      }
      else
        return new VariableExpr( ch, e.getValue() );
  }
```

Note the value of the variable was stored as an object of `NumberExpr`. `letterExpr` creates an object of `VariableExpr` with the value of the variable, which is retrieved from `e` using method `getValue`.

Method `eval` of classes `VariableExpr`, `NumberExpr`, and `CompositeExpr` evaluates the expression:

```
public class VariableExpr extends Expr {
    public VariableExpr( char name, int value ) {
        this.name = name;
        this.value = value;
    }

    public void genC() {
        System.out.print(name);
    }

    public int eval() {
        return value;
    }

    private char name;
    private int  value;
}


public class NumberExpr extends Expr {
    public NumberExpr( char n ) {
        this.n = n;
    }

    public int getValue() {
        return n - '0';
    }
    public void genC() {
        System.out.print(n);
    }

    public int eval() {
        return n - '0';
    }

    private char n;
}


public class CompositeExpr extends Expr {
    public CompositeExpr( Expr pleft, char poper, Expr pright ) {
        left = pleft;
```

```
        oper = poper;
        right = pright;
    }
    public void genC() {
        System.out.print("(");
        left.genC();
        System.out.print(" " + oper + " ");
        right.genC();
        System.out.print(")");
    }

    public int eval() {
        int evalLeft  = left.eval();
        int evalRight = right.eval();

        switch ( oper ) {
            case '+' :
              return evalLeft + evalRight;
            case '-' :
              return evalLeft - evalRight;
            case '*' :
              return evalLeft*evalRight;
            case '/' :
              if ( evalRight == 0 )
                CompilerRuntime.error("Division by zero");
              return evalLeft/evalRight;
            default :
              CompilerRuntime.error("Unknown operator");
              return 0;
        }
    }

    private Expr left, right;
    private char oper;
}
```

8 Compiler 8

This compiler accepts variables with any number of characters and numbers with more than one digit. The language used supports comments — anything after // till the end of the line is a comment. There are comparison operators (`<`, `>=`, ...) and new grammar rules. In particular, there are keywords in the grammar, as `if`, `then`, and `begin`.

The lexical analyzer has to be modified. Each time `nextToken` finds a terminal (token), it puts in instance variable `token` a number corresponding to that token. These numbers are defined in class `Symbol`:

```
public class Symbol {
    public final static int
      EOF = 0,
      IDENT = 1,
      NUMBER = 2,
      PLUS = 3,
      MINUS = 4,
      MULT = 5,
      ...

}
```

If `token` is `Symbol.IDENT`, instance variable `stringValue` of class `Compiler` contains the identifier. If `token` is `Symbol.NUMBER`, instance variable `numberValue` contains the number. Then the code

```
if ( token == Symbol.IDENT ) {
   String name = stringValue;
   ...
}
```

puts in variable `name` the identifier found.

We add to the lexical analyzer a `keywordsTable`, which is a hashtable with all keywords:

```
public class Compiler {

      // contains the keywords
    static private Hashtable keywordsTable;

     // this code will be executed only once for each program execution
    static {
        keywordsTable = new Hashtable();
        keywordsTable.put( "var", new Integer(Symbol.VAR) );
        keywordsTable.put( "begin", new Integer(Symbol.BEGIN) );
        keywordsTable.put( "end", new Integer(Symbol.END) );
        keywordsTable.put( "if", new Integer(Symbol.IF) );
        keywordsTable.put( "then", new Integer(Symbol.THEN) );
        keywordsTable.put( "else", new Integer(Symbol.ELSE) );
        keywordsTable.put( "endif", new Integer(Symbol.ENDIF) );
        keywordsTable.put( "read", new Integer(Symbol.READ) );
        keywordsTable.put( "write", new Integer(Symbol.WRITE) );

    }
    ...
}
```

This table is initialized inside a static block of class `Compiler`. This block is only executed once. To the string of each keyword is associated an `Integer` which keeps the number of that keyword. In `nextToken`, whenever a sequence of letters is found a search is made at `keywordsTable` to discover if the sequence is a keyword:

```
if ( Character.isLetter( ch ) ) {
    // get an identifier or keyword
    StringBuffer ident = new StringBuffer();
    while ( Character.isLetter( input[tokenPos] ) ) {
        ident.append(input[tokenPos]);
        tokenPos++;
    }
    stringValue = ident.toString();
      // if stringValue is in the list of keywords, it is a keyword !
    Object value = keywordsTable.get(stringValue);
    if ( value == null )
      token = Symbol.IDENT;
    else
        // get the token
      token = ((Integer ) value).intValue();
    if ( Character.isDigit(input[tokenPos]) )
      error("Word followed by a number");
}
else if ( Character.isDigit( ch ) ) {
    // get a number
    StringBuffer number = new StringBuffer();
    while ( Character.isDigit( input[tokenPos] ) ) {
        number.append(input[tokenPos]);
        tokenPos++;
    }
    token = Symbol.NUMBER;
    numberValue = Integer.valueOf(number.toString()).intValue();
}
else {
    tokenPos++;
    switch ( ch ) {
        case '+' :
          token = Symbol.PLUS;
          break;
          ...
```

The code above also accepts a number with any number of digits.

If the search at `keywordsTable` succeeds (else part), `token` will receive the integer value corresponding to the keyword.

The `assignmentStatement` method, shown below, analyzes an assignment and is called whenever `token` is `Symbol.IDENT` when analyzing a statement. The name of the variable is put at variable `stringValue`. A search is made in the symbol table to discover if the variable was declared.

```
private AssignmentStatement assignmentStatement() {

    String name = stringValue;

      // is the variable in the symbol table ? Variables are inserted in the
      // symbol table when they are declared. It the variable is not there, it has
      // not been declared.

    Variable v = (Variable ) symbolTable.get(name);
      // was it in the symbol table ?
    if ( v == null )
        error("Variable " + name + " was not declared");
    nextToken();
    if ( token != Symbol.ASSIGN )
      error("= expected");
    nextToken();
    return new AssignmentStatement( v, expr() );
}
```

At `varDec`, which analyzes a variable declaration, the variable is put at the symbol table:

```
private Variable varDec() {
    Variable v;

    if ( token != Symbol.IDENT )
      error("Identifier expected");
      // name of the identifier
    String name = stringValue;
    nextToken();
      // semantic analysis
      // if the name is in the symbol table, the variable is been declared twice.
    if ( symbolTable.get(name) != null )
      error("Variable " + name + " has already been declared");
      // inserts the variable in the symbol table. The name is the key and an
      // object of class Variable is the value. Hash tables store a pair (key, value)
      // retrieved by the key.
    symbolTable.put( name, v = new Variable(name) );
    return v;
}
```

9 Compiler 9

In this compiler the lexical analyzer and error signalling were separated from the syntactical analyzer. Each one was put into a separate class. Instance variable `lexer` of class `Compiler` points to the lexical analyzer. Instance variable `error` points to an object of class `CompilerError` that does the error signalling. Variable `lexer` has a pointer to the error object and vice-versa.

Variables now have types, which can be boolean, integer, or char. There is a class `Type` which keeps just the type name:

```
abstract public class Type {

    public Type( String name ) {
        this.name = name;
    }

    public static Type booleanType = new BooleanType();
    public static Type integerType = new IntegerType();
    public static Type charType    = new CharType();

    public String getName() {
        return name;
    }

    abstract public void genC( PW pw );

    private String name;
}
```

Class `Expr` and its subclasses now have a method `getType` which returns an object representing the type of the expression. See classes `CompositeExpr` and `NumberExpr` below.

```
public class CompositeExpr extends Expr {

    public CompositeExpr( Expr pleft, int poper, Expr pright ) {
        left = pleft;
        oper = poper;
        right = pright;
    }
    public void genC( PW pw ) {
        pw.out.print("(");
        left.genC(pw);
        pw.out.print(" " + arrayOper[oper] + " ");
        right.genC(pw);
        pw.out.print(")");
    }

    public Type getType() {
          // left and right must be the same type
       if ( oper == Symbol.EQ || oper == Symbol.NEQ ||
            oper == Symbol.LE || oper == Symbol.LT ||
            oper == Symbol.GE || oper == Symbol.GT )
            return Type.booleanType;
       else if ( oper == Symbol.AND || oper == Symbol.OR )
```

```
            return Type.booleanType;
        else
            return Type.integerType;
    }

    private Expr left, right;
    private int oper;
    private static String []arrayOper;
    static {
        arrayOper = new String[Symbol.LastSymbol];
        int i;
        for( i = 0; i < arrayOper.length; i++ )
          arrayOper[i] = "";
        arrayOper[Symbol.PLUS] = "+";
        arrayOper[Symbol.MINUS] = "-";
        arrayOper[Symbol.MULT] = "*";
        arrayOper[Symbol.DIV] = "/";
        arrayOper[Symbol.REMAINDER] = "%";
        arrayOper[Symbol.LT] = "<";
        arrayOper[Symbol.LE] = "<=";
        arrayOper[Symbol.GT] = ">";
        arrayOper[Symbol.GE] = ">=";
        arrayOper[Symbol.NEQ] = "!=";
        arrayOper[Symbol.EQ] = "==";
        arrayOper[Symbol.ASSIGN] = "=";
        arrayOper[Symbol.AND] = "&&";
        arrayOper[Symbol.OR] = "||";
    }
}

public class NumberExpr extends Expr {

    public NumberExpr( int value ) {
        this.value = value;
    }

    public int getValue() {
        return value;
    }
    public void genC( PW pw ) {
        pw.out.print(value);
    }

    public Type getType() {
        return Type.integerType;
    }
```

```
    private int value;
}
```

Class `PW` is used to correctly indent the generated code. It is the type of parameter `pw` of all `genC` methods of the AST. The code

```
    pw.println("if");
```

will print an `if` preceded by a certain number k of spaces. This number k is kept in an instance variable of class `PW`. This number can be increased by method `add` and decreased by `sub`. If necessary, one can print a string without the k spaces using `pw.out`. An example is given below.

```
pw.print("if");
pw.out.println(" ( i > 0 ) ");
pw.add();
pw.println("a = 3;");
pw.sub();
pw.println("print(a);");
```

This code will print

```
if ( i > 0 )
   a = 3;
print(a);
```

Now there is semantic analysis when analyzing a sum or subtraction of two values. Both values should be of type `integer`. See method `addExpr` below.

```
    private Expr addExpr() {
      /*
        AddExpr ::= MultExpr { AddOp MultExpr }

      */
      int op;
      Expr left, right;
      left = multExpr();
      while ( (op = lexer.token) == Symbol.PLUS ||
              op == Symbol.MINUS ) {
        lexer.nextToken();
        right = multExpr();
          // semantic analysis
        if ( left.getType() != Type.integerType ||
             right.getType() != Type.integerType )
           error.signal("Expression of type integer expected");
        left = new CompositeExpr( left, op, right );
      }
      return left;
    }
```

10 Compiler 10

In this compiler identifiers begin with a letter which may be followed by any number of letters and/or digits. The compiler continues the compilation after signalling the first error — several errors may be shown in a single compilation. The grammar defines procedures and functions that resemble Pascal:

```
procedure print( x, y : integer; ch : char )
     var i : integer;
   begin
   write(ch);
   i = 1;
   while i <= 5 do
     begin
     write( x + y );
     i = i + 1;
     end;
   end

 function factorial( n : integer ) : integer
   begin
   if ( n <= 1 )
     return 1;
   else
     return n*fatorial(n-1);
   endif;
   end
```

The symbol table has two hash tables: one for local and other for global symbols. `x`, `y`, `ch`, `i`, and `n` are local symbols. `print` and `factorial` are global. Only procedures and functions can be global in this language. Parameters and local variables are always local. `print` is represented by an object of class `Procedure` of the AST. `factorial` is represented by an object of class `Function`. Both `Procedure` and `Function` are subclasses of class `Subroutine`:

```
abstract public class Subroutine {

    abstract public void genC( PW pw );

    public String getName() {
        return name;
    }

    public void setParamList( ParamList paramList ) {
        this.paramList = paramList;
    }

    public ParamList getParamList() {
```

```
        return paramList;
    }

    public void setLocalVarList( LocalVarList localVarList ) {
        this.localVarList = localVarList;
    }

    public void setCompositeStatement( CompositeStatement compositeStatement ) {
        this.compositeStatement = compositeStatement;
    }

      // fields should be accessible in subclasses
    protected String name;
    protected LocalVarList localVarList;
    protected CompositeStatement compositeStatement;
    protected ParamList paramList;
}



public class Procedure extends Subroutine {

    public Procedure( String name ) {
        this.name = name;
    }

    public void genC( PW pw ) {
        pw.out.print("void " + name + "(");
        if ( paramList != null )
          paramList.genC(pw);
        pw.out.println(") {");
        pw.add();
        if ( localVarList != null )
          localVarList.genC(pw);
        compositeStatement.getStatementList().genC(pw);
        pw.sub();
        pw.out.println("}");
    }
}



public class Function extends Subroutine {

    public Function( String name ) {
        this.name = name;
```

```
    }

    public Type getReturnType() {
        return returnType;
    }

    public void setReturnType( Type returnType ) {
        this.returnType = returnType;
    }

    public void genC( PW pw ) {

        returnType.genC(pw);
        pw.out.print(" " + name + "(");
        if ( paramList != null )
          paramList.genC(pw);
        pw.out.println(") {");
        pw.add();
        if ( localVarList != null )
          localVarList.genC(pw);
        pw.out.println();
        compositeStatement.getStatementList().genC(pw);
        pw.sub();
        pw.out.println("}");
    }

    private Type returnType;
}
```

Class `Parameter` of the AST inherits from class `Variable`:

```
public class Variable {

    public Variable( String name, Type type ) {
        this.name = name;
        this.type = type;
    }

    public Variable( String name ) {
        this.name = name;
    }

    public void setType( Type type ) {
        this.type = type;
    }

    public String getName() { return name; }
```

```
    public void genC( PW pw ) {
        pw.out.print(name);
    }

    public Type getType() {
        return type;
    }

    private String name;
    private Type type;
}
```

```
public class Parameter extends Variable {
    public Parameter( String name, Type type ) {
        super(name, type);
    }

    public Parameter( String name ) {
        super(name);
    }
}
```

Since there are local and global identifiers, they should be treated differently by the symbol table. In fact, insertion of identifiers is made by two different methods, one for each scope:

```
symbolTable.putInGlobal( "print", new procedure("print") );
symbolTable.putInLocal(  "x",     new Parameter("x") );
```

The local symbols do not exist after the end of the subroutine in which they are declared. So, they should be removed from the symbol table at that point. This is made through method `removeLocalIdent` of `SymbolTable`.

Each symbol is inserted at the symbol table when it is declared. Subroutines are never removed from this table. Parameters and local variables are removed at the end of the subroutine that declared them.

When an identifier is the first token found in method `statement` of class `Compiler`, it is necessary to discover what this identifier is: a variable/parameter or a procedure ? It may be both:

```
 a = 1;
 print(1, 1, \'A');
```

Then the compiler searches at the local and global table of the symbol table. This is made through method `get` of `SymbolTable`, which first searches at the local and then at the global table. When `lexer.token` is `Symbol.IDENT`, `lexer.getStringValue()` returns the string containing the identifier. See the code below.

```
private Statement statement() throws StatementException {
    /*  Statement ::= AssignmentStatement | IfStatement | ReadStatement |
           WriteStatement | ProcedureCall | ForStatement | WhileStatement |
           CompositeStatement | ReturnStatement

    */

    switch (lexer.token) {
        case Symbol.IDENT :
            // if the identifier is in the symbol table, "symbolTable.get(...)"
            // will return the corresponding object. If it is a procedure,
            // we should call procedureCall(). Otherwise we have an assignment
          if ( symbolTable.get(lexer.getStringValue()) instanceof Procedure )
              // Is the identifier a procedure ?
            return procedureCall();
          else
            return assignmentStatement();
        case Symbol.IF :
          return ifStatement();
        case Symbol.READ :
          ...
    }
}
```