Learning Compiler Construction by Examples

José de Oliveira Guimarães
jose@dc.ufscar.br

May 8, 2017

Contents

This report is a small course on compiler construction. It explains how to build compilers using ten examples in Java. The first example uses a very small grammar described in three lines. The last one is a compiler for a language similar to Pascal. Each example, in general, adds some grammar rules to its predecessor.

All the main elements of compiler construction are addressed in this report: lexical analysis, syntactical analysis, semantic analysis, abstract syntax tree (AST), code generation (to language C and assembly), symbol table, and code generation through the AST. Object-oriented programming is used extensively. In particular, the AST is composed by objects, which nicely organizes the compiler.

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 be no code generation. Anyway, it will be called a compiler. Syntactic analysis is also called parsing and a syntactic analyzer is usually called parser.

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:

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

        Compiler compiler = new Compiler();

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

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. Instead of producing numbers representing the terminals, it produces the terminals themselves. This is only possible because all terminals are one-character tokens. Methods `expr`, `oper`, and `number` compose the parser (syntactical analyzer).

```
public class Compiler {

    public void compile( char []p_input ) {
        input = p_input;
        tokenPos = 0;
        nextToken();
        expr();
        if ( token != '\0' )
          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 );
           // throws an exception - the program is terminated
        throw new RuntimeException(strError);
    }

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

}
```

Each method of the parser (syntactical analyzer) corresponds 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 (part "then" of the `if` statement above), `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` is `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 check if the first rule can produce an expression that starts with `'(`. It can and we choose this rule. Then in method `expr` there is an `if` to decide which rule to use based on 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`.

There is a conceptual error in class `Compiler`. In object-oriented programming, methods represent actions and should have verbal names such as `calculate`, `select`, `findWord`, `delete`, `addElement`, and so on. However, we used nouns in class `Compiler`, the grammar non-terminal names `expr`, `number`, and `oper`. They should be replaced by `analyzeExpr`, `analyzeNumber`, and `analyzeOper`. The new names really express the correct actions taken by each method. Nevertheless, we will use the non-terminals for method names in this report, just to make things easy.[1]

Let us see now some common patterns of grammar rules and their corresponding parsers. Lower case letters and quoted symbols (between ") represent terminals. Upper case letters represent non-terminals. The lexical analyzer for each grammar is not shown.

- Grammar:

```
A ::= B | C
B ::= b E
C ::= c F | d C
```

Analyzer:

```
void A() {
   if ( token == 'b' )
      B();
   else
      C();
}

void B() {
   // we know B is only called if the current token is 'b'
   // We assumed that B is only used in the production for A
   nextToken();
   E();
}

void C() {
   // we know C is called if the current token is any other than
   // 'b'. Therefore, we should test if token is either 'c' or 'd'
   if ( token == 'c' ) {
      nextToken();
      F();
   }
   else if ( token == 'd' ) {
      nextToken();
      C();
```

[1] This is a disease called LPS — "Lazy Programmer Syndrome".

```
   }
   else
      error("c or d expected");
}
```

Note that the method for non-terminal `F` is not shown. It can be any one. The rule for F does not interfere with the methods for rules `A`, `B`, or `C`. This situation repeats in the following grammars.

- Grammar:

  `A ::= b B | c C | d D`

  Analyzer:

```
void A() {
   switch (token) {
      case 'b' :
         nextToken();
         B();
         break;
      case 'c' :
         nextToken();
         C();
         break;
      case 'd' :
         nextToken();
         D();
         break;
      default :
         error("b, c, or d expected");
   }
}
```

- Grammar:

  `A ::= [ b B ] C`
  Anything between [and] is optional.

  Analyzer:

```
void A() {
   if ( token == 'b' ) {
      nextToken();
      B();
   }
   C();
}
```

- Grammar:

```
A ::= [ B ] C
B ::= b D | c E
```

Analyzer:

```
void A() {
   if ( token == 'b' || token == 'c' )
      B();
   C();
}

void B() {
   if ( token == 'b' ) {
      nextToken();
      D();
   }
   else if ( token == 'c' ) {
      nextToken();
      E();
   }
   else
      error("b or c expected");
}
```

- Grammar:

```
A ::= B [ C ] D
C ::= c E
```

Analyzer:

```
void A() {
   B();
   if ( token == 'c' )
      C();
   D();
}
```

- Grammar:

```
A ::= [ B ] [ C ] D
B ::= b E
C ::= c F
```

Analyzer:

```
void A() {
```

```
   if ( token == 'b' )
      B();
   if ( token == 'c' )
      C();
   D();
}

void B() {
      // B is only called if the current token is b
      // Assume B is only used in A
   nextToken();
   E();
}

void C() {
      // C is only called if the current token is c
      // Assume C is only used in A
   nextToken();
   F();
}
```

- Grammar:

  ```
  A ::= B { B }
  B ::= b C | c D
  ```
  Anything between { and } can be repeated zero or more times.

  Analyzer:

```
void A() {
   B();
   while ( token == 'b' || token == 'c' )
      B();
}

void B() {
      // note that the current token is not necessarily b or c
      // because of the first call to B in method A
   if ( token == 'b' ) {
      nextToken();
      C();
   }
   else if ( token == 'c' ) {
      nextToken();
      D();
   }
   else
        // if B is called only at A(), this will never happen.
```

```
      error("b or c expected");
}
```

- Grammar:

```
A ::= B { "," B }
B ::= b C
```

Analyzer:

```
void A() {
   B();
   while ( token == ',' ) {
      nextToken();
      B();
   }
}

void B() {
   if ( token == 'b' ) {
      nextToken();
      C();
   }
   else
      error("b expected");
}
```

- Grammar:

```
A ::= { B }
B ::= b C | c D
```

The rule for `A` can be replaced by `A ::= | B A`, in which the first right-hand side, before `|`, is empty.

Analyzer:

```
void A() {
   while ( token == 'b' || token == 'c' )
      B();
}

void B() {
      // if B is called only at production for A,
      // then we know token is either b or c. Even so
      // we have to test it.
   if ( token == 'b' ) {
      nextToken();
      C();
```

```
   }
   else if ( token == 'c' ) {
      nextToken();
      D();
   }
   else
      error("b or c expected");
}
```

- Grammar:

```
A ::= { a [ B ] }
B ::= b C | c D
```

Analyzer:

```
void A() {
   while ( token == 'a' ) {
      nextToken();
      if ( token == 'b' || token == 'c' )
         B();
   }
}

void B() {
   if ( token == 'b' ) {
      nextToken();
      C();
   }
   else if ( token == 'c' ) {
      nextToken();
      D();
   }
   else
      error("b or c expected");
}
```

- Grammar:

```
A ::= { B | C }
B ::= b f g E | e F
C ::= c G
```

Analyzer:

```
void A() {
   while ( token == 'b' || token == 'e' || token == 'c' ) {
```

```
      if ( token == 'c' )
         C();
      else
         B();
   }
}

void B() {
   if ( token == 'b' ) {
      nextToken();
      if ( token == 'f' )
         nextToken();
      else
         error("f expected");
      if ( token == 'g' )
         nextToken();
      else
         error("g expected");
      E();
   }
   else if ( token == 'e' ) {
      nextToken();
      F();
   }
   else
      error("b or e expected");
}

void C() {
   if ( token == 'c' ) {
      nextToken();
      G();
   }
   else
      error("c expected");
}
```

Exercises

1. How do you identify in a grammar which symbols are terminals? Which are the terminal symbols in the grammars shown below? The grammars are incomplete.

- (a)

```
CompositeStatement ::= "begin" StatementList "end"
```

```
StatementList ::= | Statement ";" StatementList
Statement ::= AssignmentStatement | IfStatement | ReadStatement |
              WriteStatement
AssignmentStatement ::= Variable "="  Expr
IfStatement ::= "if" Expr "then" StatementList [ "else" StatementList ] "endif"
ReadStatement ::= "read" "(" Variable ")"
WriteStatement ::= "write" "(" Expr ")"
```

- (b)

```
A ::= a A | B b
B ::= C { "," C }
```

2. Make the parsers of the following grammars. Assume a class `Compiler` is available with methods `number()`, `error()`, and `nextToken()`. Use rule `number` of language 1.

(a) `E ::= T { '+' T }`
`T ::= number | 'i' | 'j'`

(b) `E ::= T { T '*' }`
`T ::= number | 'i'`

3. The grammar of language LE is: E ::= T E′

E′ ::= + T E′ | ϵ
T ::= F T′
T′ ::= $*$ F T′ | ϵ
F ::= Numero | (E)

The terminal symbols this language must be separated by white spaces. Numbers are composed by a single digit. Make in Java a parser for this language.

4. Write a grammar for a prefix expression (**PreE**). Then, make a Java parser for this language.

5. What does a parser do?

2 Compiler 2

This compiler generates code in language C for the expression that is the compiler input. Each method generates code for the piece of the grammar it analyzes. Method `expr` generates code for the grammar rule `Expr` and method `number` for the rule `number`. A singleton 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`, through "`System.out`". 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 to `expr` 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';
        }
    }
```

The complete compiler is shown below.

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

        Compiler compiler = new Compiler();

        compiler.compile(input);
    }
}


// ###################################################################
/*
    comp2

    This program is a syntactic analyzer and code generator for the following grammar:

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

     The code is generated to C.

*/

public class Compiler {

    public void compile( char []p_input ) {
        input = p_input;
        tokenPos = 0;
        nextToken();
        expr();
```

```
    if ( token != '\0')
      error();
}

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();
}

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

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

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.print( strError );
        throw new RuntimeException(strError);
    }

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

}
```

Exercises

6. The parsers of the following grammars were made in an exercise of Section 1. Add code generation to C to these parsers.

(a)
```
E ::= T { '+' T }
T ::= number | 'i' | 'j'
```

(b)
```
E ::= T { T '*' }
T ::= number | 'i'
```

7. Add to the compiler of language LE (see exercises of Compiler 1) statements to generate code in C.

8. Add to the compiler of language PreE (see exercises of Compiler 1) statements to generate code in C. The generated code must be a infix expression with parenthesis.

3 Compiler 3

This compiler uses the same language as compilers 1 and 2. It 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 singleton number, simply pushes the number into the stack. When `number` analyzes

```
1
```

it generates

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

in which `R0` is a machine register. 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`, adds both numbers, and pushes 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();
    }
```

The complete compiler is shown below.

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

        Compiler compiler = new Compiler();

        compiler.compile(input);
    }
}


// ##################################################################
/*
    comp3

    This program is a syntactic analyzer and code generator for the following grammar:

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

     The code is generated to ASM as described in Chapter 8 of the primer
     "Construção de Compiladores". The generated code is as bad as anyone can ever design
     but it is enough to show how assembler code is produced.
```

```
*/

public class Compiler {

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

    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();
    }

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

```
        else
          error();
    }

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

    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.print( strError );
        throw new RuntimeException(strError);
    }

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

}
```

Exercises

9. The parsers of the following grammars were made in an exercise of Section 1. Add code generation to assembly to these parsers.

(a)
```
E ::= T { '+' T }
T ::= number | 'i' | 'j'
```

(b)
```
E ::= T { T '*' }
T ::= number | 'i'
```

10. Add to the compiler of language LE (see exercises of Compiler 1) statements to generate code in assembly.

11. Add to the compiler of language PreE (see exercises of Compiler 1) statements to generate code in assembly.

4 Compiler 4

This compiler evaluates the expression. That is, at the end of parsing, 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, a character, 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:

```
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();
}
```

The complete compiler is shown below.

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

        Compiler compiler = new Compiler();

        System.out.println("Resultado = " + compiler.compile(input) );
    }
}


// ###################################################################
/*
    comp4

    This program is a syntactic analyzer and evaluator for the following grammar:

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

    The evaluation of the expression is printed in the standard output
*/

public class Compiler {

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

    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();
}


private int number() {
    int result = 0;

    if ( token >= '0'  && token <= '9' ) {
        result = token - '0';
        nextToken();
    }
    else
      error();
    return result;
}

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

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.print( strError );
        throw new RuntimeException(strError);
    }

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

}
```

Exercises

12. The parsers of the following grammars were made in an exercise of Section 1. Change the parser in such a way that it now calculates the value of the expression.

(a)
```
E ::= T { '+' T }
T ::= number | 'i' | 'j'
```

(b)
```
E ::= T { T '*' }
T ::= number | 'i'
```

13. Add to the compiler of language LE (see exercises of Compiler 1) statements to calculate the value of the expression.

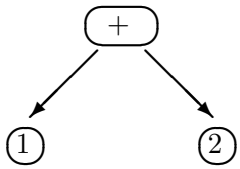
14. Add to the compiler of language PreE (see exercises of Compiler 1) statements to calculate the value of the expression.

5 Compiler 5

This compiler builds an Abstract Syntax Tree (AST) for the input expression and generates code through it. The grammar is still the same as compiler 1:

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

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 Java objects. The balls that hold numbers (`1` or `2`) are objects of class `NumberExpr`. As the name says, this class represents a number. It 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` which represents an expression with an operator, a composite expression. Clearly, an object of this class needs to store the operator and pointers to the left and right expressions:

```
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 either 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 variable `left` (`right`) of `CompositeExpr` should be such that this variable can point to either a `CompositeExpr` or a `NumberExpr` object. The only way to achieve that is to create a common superclass which, in this case, is `Expr`. 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` declares an abstract method `genC` whose meaning is "generate C code for the expression". Method `genC` is redefined in both subclasses. In `NumberExpr`, 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 `')'`.

In class `CompositeExpr`, instance variable `left` has type `Expr` and there is no object of class `Expr` since this class is abstract. But `Expr` has subclasses and `left` will point to an object of an `Expr` subclass (`CompositeExpr` or `NumberExpr`). Since "`left.genC()`" is used in method `genC`, it was necessary to define an abstract method `genC` at `Expr`, which is the common superclass of `CompositeExpr` and `NumberExpr`.

Methods `number` and `expr` of the parser 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;
}
```

The return type of method `expr` should be `Expr` since this method can return either a `NumberExpr` or a `CompositeExpr` object:

```
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;      // return a CompositeExpr object
    }
    else
      return number();  // return a NumberExpr object
}
```

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`.

Method `compile` also returns an object of the AST. In fact, it returns the object of the AST that represents the whole 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:

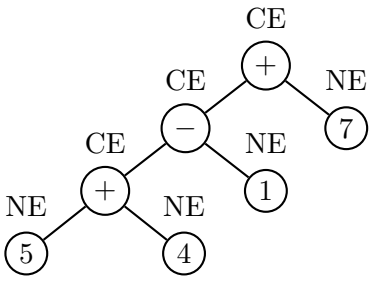
```
public class Main {
    public static void main( String []args ) {
        char []input = "( + (-  (+ 5 4) 1)  7)".toCharArray();

        Compiler compiler = new Compiler();

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

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

The whole compiler follows.

```
// comp5

import AST.*;

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();
    }
}
```

The syntatic analyzer builds the AST (Abstract Sintax Tree) below:



```
// ###############################################################
/*
    comp5

    This program is a syntactic analyzer that builds the abstract syntax tree (AST). The
    classes of the AST were put in the package AST. The classes of the AST represent
    expressions and all have a "genC" method. If exp is a variable of class Expr, then
          exp.genC()
    generates code in C for the expression.

    Grammar:

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


*/

import AST.*;

public class Compiler {


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

    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();
}

private NumberExpr number() {
    NumberExpr e = null;

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

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

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.print( strError );
        throw new RuntimeException(strError);
    }

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

}

// ###################################################################
package AST;

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


// ###################################################################
package AST;

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;
}


// ###################################################################
package AST;
```

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

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

Exercises

15. Change the compiler of language LE (see exercises of Compiler 1) in such a way it builds the AST of the program. The AST classes should have methods `genC` to generate code in C for the program.

16. Make in Java the lexical and syntactical analyzers for the language MU defined by the grammar

S	::=	Expr ExprList
ExprList	::=	ϵ | Expr ExprList
Expr	::=	"if" Expr "then" Expr "else" Expr "endif" | E > E | E == E | E
E	::=	E + T | E "or" T | T
T	::=	T * F | T "and" F | F
F	::=	Number | "(" Expr ")"

Any number of white spaces can appear between two terminals. Number represents positive integers with any number of digits.

Construct the AST during the syntactical analysis. Of course, design the AST classes.

17. Draw the AST of the expression

```
if 2 > 5 then 2*3 else 1 + (if 1 then 3 else 5 endif) endif
```

It is a sentence of language MU. In the drawing, use circles for objects and arrows for pointer references.

18. Codify a Java expression that creates the AST of the expression of the previous exercise. For example, the expression

```
if 2 > 1 then 0 else 3
```

can be created by

```
IfExpr ifExpr = new IfExpr(
    new CompositeStatement( new NumberExpr('2'), '>', new NumberExpr('1') ),
    new NumberExpr('0'),
    new NumberExpr('3')
    );
```

19. Why do the compiler build the AST if it has access to the same information through the symbol table and source program?

20. For each of the commands/declarations defined by the grammar rules given below, do the following:

- write the grammar rules that describe the command/declaration;
- define the classes of the AST. Specify the instance variables, inheritance, and a constructor. It is not necessary to define the other methods;
- write the tokens the lexical analyzer should recognize;
- codify the methods of class `Compiler` that analyzes the command/declaration.

1. `Records` as in Pascal or `structs` de C:

```
var a : record
          x, y : integer;
        end
```

2. The command

```
loop   C1; C2; ... Cn; end
```

   which is an endless repetition of the commands C1, C2, ... Cn. The loop ends when statement `break` is executed.

3. Arrays as in Pascal:

```
var a : array[S..E] of T;
```

   `S` and `E` are integer literals.

4. Enumerated types as in Pascal and other languages:

```
var cor : enum ( red, yellow, green, black, white ) ;
```

   There is no basic type, as `integer`, associated to the constants. So, a statement
   `i = white`
   would be illegal if `i` were an integer.

5. Types as in Pascal:

```
type Color = integer;
```

   `Color` could be used as a replacement for `integer`.

6. A command

```
loop C1; C2; ... Cn times E;
```

   in which C1, C2, ... Cn are commands and `E` is an expression. The sequence of statements `Ci` is repeated `N` times. `N` is the value of `E` evaluated before the execution of any statement `Ci`. If `N < 0`, `N` receives 0.

6 Compiler 6

This language supports variable[2] 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'
```

The first rule for `VarDecList` uses empty as the right-hand side. The first rule is that before `|`.

In this grammar, there is a clear need for a new AST class: `Variable`. A variable in this language not only has a name but it also keeps a value. Therefore, the AST class for `Variable` should have two instance variables:

```
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;
}
```

Method `genC` generates the code for the *declaration* of the variable.

The new class `Compiler` has methods for rules `Program`, `VarDecList`, `VarDec`, `Expr`, `Oper`, `Number`, and `Letter`. `VarDecList` is a list of zero or more `VarDec`'s, 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 method `varDecList`:

```
private ArrayList<Variable> 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 */
    ArrayList<Variable> arrayVariable = new ArrayList<Variable>();
    while ( Character.isLetter(token) )
        arrayVariable.add( varDec() );
    return arrayVariable;
```

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

```
   }
}
```

Objects of `Vector` are arrays with an undetermined number of elements. Objects of any class are sequentially added to the array by method `addElement`. Method `isLetter` is a static method of class `Character` and tests if its parameter is a letter.

Method `program` gets from `VarDecList` the vector with the variables and returns an object of `Program` with the vector and the expression:

```
private Program program() {
    ArrayList<Variable> arrayVariable = varDecList();
    if ( token != ':' ) {
        error();
        return null;
    }
    else {
        nextToken();
        Expr e = expr();
        return new Program( arrayVariable, e );
    }
}
```

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 `number` checks if the current token is a number:

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

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

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 that if the expression is a variable, as in

```
a = 1 : a
```

method `expr` (second line from bottom to top) returns an object of `VariableExpr`, not `Variable`. Class `Variable` represents *declarations* of variables, not the use of a variable in an expression. See the code of classes `Variable` and `VariableExpr` (that follows). Method `genC` of `VariableExpr` generates code for a variable in an expression, which is just the variable itself. Method `genC` of `Variable` generates code for the declaration of the variable in language C, something like "`int a = 1;`". The return type of method `expr` is `Expr` and therefore should return, in `expr`, an object of a subclass of `Expr`, which is abstract. Then we could not return an object of `Variable`. Class `Variable` does not inherit from `Expr`. Even if it did, there would be the problem of method `genC` just exposed.

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

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

    private char name;
}
```

Method `Program::genC`[3] generates code for the main program in C, which will contain the variable declarations and the expression:

```
package AST;

import java.util.*;

public class Program {
    public Program( ArrayList<Variable> arrayVariable, Expr expr ) {
        this.arrayVariable = arrayVariable;
        this.expr = expr;
    }
    public void genC() {
        System.out.println("#include <stdio.h>\n");
        System.out.println("void main() {");
        for ( Variable v : arrayVariable )
            v.genC();

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

    private ArrayList<Variable> arrayVariable;
    private Expr expr;
}
```

The generated code for the program

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

would be

```
#include <stdio.h>
void main() {
```

[3]Method `genC` of class `Program`.

```
int a = 1;
int b = 3;
printf("%d\n", ((a + 2) - b));
}
```

The complete compiler code is in Appendix A.

7 Compiler 7

This compiler uses the same grammar as compiler 6:

```
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'
```

The compiler evaluates the expression. It uses a symbol table to keep the value of each variable. The symbol table is a simple hash table, declared in class `Compiler` as

```
Hashtable symbolTable;
```

Class `Hashtable` has methods

```
Object put(Object key, Object value)
Object get(Object key)
```

Method `put` inserts `value` at the table using `key` as its name (or key!). It returns the table object that had key `key`. If there were none, it returns `null`:

```
Hashtable h = new Hashtable();
h.put( "one", new Integer(1) );
h.put( "two", new Integer(2) );
Object obj = h.put("one", "I am the one");
System.out.println(obj);   // prints 1
obj = h.get("two");
System.out.println(obj);   // prints 2
System.out.println( h.get("one") );  // prints "I am the one"
char name = 'a';
h.put( name + "", "This is an a");
System.out.println( h.get("a") );
```

In the last call to `put`, we used `name + ""` as the key. We could not have used just `name` because `name` is not an object. We concatenate this variable to the empty string to create the string `"a"`, which is an object and therefore can be a parameter to `put`. Anything concatenated with a string becomes a string:

```
String one = 1 + ""; // assigns "1" to one
```

The last line of the example above will print `"This is an a"`.

The symbol table is set up in method compile:

```
public Program compile( char []p_input ) {
    input = p_input;
    tokenPos = 0;
    symbolTable = new Hashtable<Character, Variable>();
    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 num = number();
        // semantic analysis
        // inserts the variable in the Symbol Table. It will be used to
        // retrieve the value of the variable when it is found inside the
        // expression and to check if the variable found in the expression
        // was "declared" before the ':'
        // Note we insert the name converted to a String in the table
      Variable v = new Variable( name, num.eval() );
      if ( symbolTable.put( name, v ) != null )
        error("Variable " + name + " has already been declared");
      return v;
    }
}
```

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

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();
    }
```

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

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

Note that the value of the variable was stored in the hash table as an object of `Variable` in method `varDec`. Method `letterExpr` creates an object of `VariableExpr` with the value of the variable.

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

```
public class VariableExpr extends Expr {

    public VariableExpr( Variable variable ) {
        this.variable = variable;
    }

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

    public int eval() {
        return variable.eval();
    }

    private Variable variable;

}

public class NumberExpr extends Expr {

    public NumberExpr( int n ) {
        this.n = n;
```

```
    }

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

    public int eval() {
        return n;
    }

    private int 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;
}
```

There is a class `CompilerRuntime` used to signal errors during the evaluation of the expression. It will not be explained here.

The complete compiler code is in Appendix B.

Exercises

21. Make the semantic analyzer of the compiler of language MU defined in the exercises of compiler 5. The type of the if expression should be `boolean`. The types of the expressions that follow `then` and `else` must be equal. The arithmetical expressions follow the usual rules.

22. Make the semantic analyzer of each item of the last exercise of Compiler 5.

8 Compiler 8

This compiler accepts variables with any number of characters and numbers with more than one digit. The language 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 grammar is

```
Program ::= [ "var" VarDecList ";" ] CompositeStatement
CompositeStatement ::= "begin" StatementList "end"
StatementList ::= | Statement ";" StatementList
Statement ::= AssignmentStatement | IfStatement | ReadStatement |
              WriteStatement
AssignmentStatement ::= Variable "="  Expr
IfStatement ::= "if" Expr "then" StatementList [ "else" StatementList ] "endif"
ReadStatement ::= "read" "(" Variable ")"
WriteStatement ::= "write" "(" Expr ")"
Variable ::= Letter { Letter }
VarDecList ::= Variable | Variable "," VarDecList
Expr:=  '(' oper Expr  Expr ')' | Number | Variable
Oper ::= '+' | '-' | '*' | '/' | '<' | '<=' | '>' | '>='| '==' | '<>'
Number ::= Digit { Digit }
Digit ::= '0'| '1' | ... | '9'
Letter ::= 'A' | 'B'| ... | 'Z'| 'a'| 'b' | ... | 'z'
```

An example of code in this language is

```
var num, i, max ;
begin
num = 0;
i = (+ num 2);
if (< i 1)
then
   max = 0;
else
   max = 3;
endif;
write(max);
write(i);
read(i);
write( (+ i i) );
end
```

Of course, `write` prints its expression in the standard output and `read` reads a value from the standard input. The other commands explain themselves.

The lexical analyzer has to be modified. Now, 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`, which is a `enum` type:

```
public enum Symbol {
      EOF("eof"),
      IDENT("Identifier"),
      NUMBER("Number"),
      PLUS("+"),
      MINUS("-"),
        ...
      Symbol(String name) {
          this.name = name;
      }
      public String toString() { return name; }
      public String name;
}

}
```

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<String, Symbol> keywordsTable;

     // this code will be executed only once for each program execution
    static {
        keywordsTable = new Hashtable<String, Symbol>();
        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:

```
private void nextToken() {
    char ch;

    while ( ( ch = input[tokenPos]) == ' ' || ch == '\r' ||
              ch == '\t' || ch == '\n' )  {
        // count the number of lines
        if ( ch == '\n')
            lineNumber++;
            tokenPos++;
    }
    if ( ch == '\0')
        token = Symbol.EOF;
    else
         // skip comments
        if ( input[tokenPos] == '/' && input[tokenPos + 1] == '/' ) {
            // comment found
            while ( input[tokenPos] != '\0'&& input[tokenPos] != '\n' )
                tokenPos++;
            nextToken();
        }
        else {
            if ( Character.isLetter( ch ) ) {
                  // get an identifier or keyword
                  // StringBuffer represents a string that can grow
                StringBuffer ident = new StringBuffer();
                  // is input[tokenPos] a letter ?
                  // isLetter is a static method of class Character
                while ( Character.isLetter( input[tokenPos] ) ) {
                      // add a character to ident
                    ident.append(input[tokenPos]);
                    tokenPos++;
                }
                  // convert a StringBuffer object into a
                  // String object
                stringValue = ident.toString();
                  // if identStr is in the list of keywords, it is a keyword !
                Symbol value = keywordsTable.get(stringValue);
                if ( value == null )
                   token = Symbol.IDENT;
                else
                   token = value;
```

```
    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;
    try {
         /*
            number.toString() converts a StringBuffer
            into a String object.
            valueOf converts a String object into an
            Integer object. intValue gets the int
            inside the Integer object.
         */
       numberValue = Integer.valueOf(number.toString()).intValue();
    } catch ( NumberFormatException e ) {
       error("Number out of limits");
    }
    if ( numberValue >= MaxValueInteger )
       error("Number out of limits");
    if ( Character.isLetter(input[tokenPos]) )
      error("Number followed by a letter");
}
else {
    tokenPos++;
    switch ( ch ) {
        case '+' :
          token = Symbol.PLUS;
          break;
        case '-' :
          token = Symbol.MINUS;
          break;
        case '*' :
          token = Symbol.MULT;
          break;
        case '/' :
          token = Symbol.DIV;
          break;
        case '<' :
          if ( input[tokenPos] == '=' ) {
              tokenPos++;
```

```
                    token = Symbol.LE;
                  }
                  else if ( input[tokenPos] == '>' ) {
                    tokenPos++;
                    token = Symbol.NEQ;
                  }
                  else
                    token = Symbol.LT;
                  break;
                case '>' :
                  if ( input[tokenPos] == '=' ) {
                    tokenPos++;
                    token = Symbol.GE;
                  }
                  else
                    token = Symbol.GT;
                  break;
                case '=' :
                  if ( input[tokenPos] == '=' ) {
                    tokenPos++;
                    token = Symbol.EQ;
                  }
                  else
                    token = Symbol.ASSIGN;
                  break;
                case '(' :
                  token = Symbol.LEFTPAR;
                  break;
                case ')' :
                  token = Symbol.RIGHTPAR;
                  break;
                case ',' :
                  token = Symbol.COMMA;
                  break;
                case ';' :
                  token = Symbol.SEMICOLON;
                  break;
                default :
                  error("Invalid Character: '" + ch + "'");
            }
        }
    }
}
```

Note that there are several novelties in `nextToken`:

- the first `while` skips all white spaces (`' '`, `'\r'`, `'\t'`, and `'\n'`) and counts the number of

lines. Variable `lineNumber` is used by method `error` which now prints the number of the line with the compiler error;

- the second `while` skips comments. Method `nextToken` is recursively called after the `while` to skip aother comment that may follow (there may be a comment after a comment — so, it is not sufficient to skip *one* comment);

- if a letter is found, a sequence of letters is gathered in `ident`;

- if a digit is found, a sequence of digits is gathered in `number`, whose integer value is retrieved by the line

  ```
  numberValue = Integer.valueOf(number.toString()).intValue();
  ```

  If there is any error in the conversion, an exception is thrown.

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

Let us study some of the classes of the AST.

- By the grammar, a program is composed by a list of variables and a composite statement:

  ```
  Program ::= [ "var" VarDecList ";" ] CompositeStatement
  ```

  Therefore, class `Program` declares two instance variables:

  ```
  private Vector arrayVariable;
  private StatementList statementList;
  ```

  `statementList` represents `CompositeStatement`. Note that the lexical elements "`begin`" and "`end`" of `CompositeStatements` are not represented by `statementList`. Rule `StatementList` of the grammar (see below) does not use any keywords.

- A `CompositeStatement`, whose rule is

  ```
  CompositeStatement ::= "begin" StatementList "end"
  ```

  does not need an AST class. A composite statement can be represented by a statement list.

- A `StatementList` is represented by class `StatementList` which declares an instance variable of class Vector. The grammar rule for this non-terminal is

  ```
  StatementList ::= | Statement ";" StatementList
  ```

  There may be zero or more statements in an object of `StatementList`.

- The rule

  ```
  Statement ::= AssignmentStatement | IfStatement |
                ReadStatement | WriteStatement
  ```

  is represented in the AST by

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

  Rule `Statement` does not define anything by itself. So it should be an abstract class. Each of the non-terminals of the right-hand side should have its own class, which must be subclass of `Statement`.

- Rule
  ```
  AssignmentStatement ::= Variable "="  Expr
  ```
  describes an assignment, which is clearly composed by a variable and an expression. So, class `AssignmentStatement` declares the following instance variables:
  ```
  private Variable v;
  private Expr expr;
  ```
  This class inherits from `Statement`.

- The rule
  ```
  IfStatement ::= "if" Expr "then" StatementList
                  [ "else" StatementList ] "endif"
  ```
  describes an if statement, which needs an expression and two statement lists. Class `IfStatement` of the AST inherits from `Statement` and declares the following instance variables:
  ```
  private Expr expr;
  private StatementList thenPart, elsePart;
  ```

- Rule
  ```
  ReadStatement ::= "read" "(" Variable ")"
  ```
  describes a read statement. Clearly, class `ReadStatement` should have only one instance variable:
  ```
  private Variable v;
  ```
  This class inherits from `Statement`.

- Rule
  ```
  WriteStatement ::= "write" "(" Expr ")"
  ```
  describes a write statement. It is represented in the AST by the `WriteStatement` class which declares the instance variable
  ```
  private Expr expr;
  ```

- Class `Variable` of the AST represents a variable and has only one instance variable:
  ```
  private String name;
  ```
  In the grammar, a variable is describe by the rule
  ```
  Variable ::= Letter { Letter }
  ```

- Rule
  ```
  VarDecList ::= Variable | Variable "," VarDecList
  ```
  describes a list of variables. There is no AST class for it. The non-terminal `VarDecList` is only used in rule `Program`:
  ```
  Program ::= [ "var" VarDecList ";" ] CompositeStatement
  ```
  In class `Program` of the AST, a list of variables is represented by a vector:
  ```
  private Vector arrayVariable;
  ```

- Rule
  ```
  Expr::=  '(' oper Expr  Expr ')' | Number | Variable
  ```
  describes an expression. There is an abstract class `Expr` in the AST:

```
package AST;

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

The first item of the right-hand side of the rule,

```
'(' oper Expr  Expr ')'
```

describes an expression with an operator. It is represented in the AST by class `CompositeExpr`, which has three instance variables:

```
private Expr left, right;
private Symbol oper;
```

`CompositeExpr` inherits from `Expr`.

The second item of the right-hand side of the rule,

```
Number
```

describes a number and is represented in the AST by class `NumberExpr`. This class inherits from `Expr`. The third item, `Variable`, is represented by class `VariableExpr`. It could not be `Variable` because class `Variable` represents a variable in a declaration — methods `genC` of `Variable` and `VariableExpr` are different.

Classes `CompositeExpr`, `NumberExpr`, and `VariableExpr` inherit from `Expr`. This is necessary because:

- method `Compiler::expr` returns an object of `CompositeExpr`, `NumberExpr`, or `VariableExpr`. The return type of `expr` should be of a superclass of all of these classes;
- class `CompositeExpr` has instance variables `left` and `right` that can refer to objects of any of the three classes already cited. The type of `left` and `right` should be of a superclass of all of these classes. The chosen superclass is `Expr`.

- Rule

```
Oper ::= '+' | '-' | '*' | '/' | '<' | '<=' | '>' |
         '>='| '==' | '<>'
```

is not represented by any AST class. It is represented by instance variable `oper` of class `CompositeExpr`.

- Rule

```
Number ::= Digit { Digit }
```

is represented in the AST by class `NumberExpr`, which inherits from class `Expr`. Rule `Number` only appears in rule `Expr` and it is not necessary to create a class `Number`. Class `NumberExpr` defines one instance variable:

```
private int value;
```

- The rules

```
Digit ::= '0'| '1' | ... | '9'
Letter ::= 'A' | 'B'| ... | 'Z'| 'a'| 'b' | ... | 'z'
```

are not represented by any AST class. A sequence of `Digits` compose a number which is

represented by instance variable `value` of class `NumberExpr`. A number is a token returned by the lexical analyzer (`Symbol.NUMBER`). A sequence of letters is an identifier, a token returned by the lexical analyzer (`Symbol.IDENT`). The identifier is stored in the AST in instance variable `name` of class `Variable`. Note that although `Digit` and `Letter` are grammar rules, they do not appear in the parser — it is the lexical analyzer that takes care of them.

Method `statement` decides which method to call based on the current token:

```
private Statement statement() {
    /*  Statement ::= AssignmentStatement | IfStatement | ReadStatement |
                      WriteStatement
    */

    switch (token) {
        case IDENT :
          return assignmentStatement();
        case IF :
          return ifStatement();
        case READ :
          return readStatement();
        case WRITE :
          return writeStatement();
        default :
            // will never be executed
          error("Statement expected");
    }
    return null;
}
```

In the `assignmentStatement` method, a search is made at the symbol table to discover if the variable of the left-hand side was declared. The variable name is got from variable `stringValue`. Note that when this method is called, the current token is `Symbol.IDENT`.

```
private AssignmentStatement assignmentStatement() {

      // the current token is Symbol.IDENT and stringValue
      // contains the identifier
    String name = stringValue;

      // is the variable in the symbol table ? Variables are inserted in the
      // symbol table when they are declared. If 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");
```

```
      // eat token Symbol.IDENT
    nextToken();
    if ( token != Symbol.ASSIGN )
      error("= expected");
    nextToken();
    return new AssignmentStatement( v, expr() );
}
```

At method `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 has 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;
}
```

The complete compiler code is in Appendix C.

Exercises

23. Change the lexical analyzer of compiler 8 in such a way that at the end of the compilation a statistics is made on the use of keywords in the program. The output of the compiler could be

```
and        : 2
begin      : 3
boolean    : 1
...
write      : 10
```

meaning that in the compiled program there was two `and`'s, three `begin`'s, and so on.

24. Change compiler 8 to output the same program as the input but with each comment replaced by a single white space.

25. Change method `nextToken` in such a way that the compiler now considers upper and lower case letters equivalent in identifiers and keywords.

26. Change method `nextToken` in such a way that the compiler now considers upper and lower case letters equivalent in identifiers only.

27. Suppose the language permits numbers in identifiers. The only requirement is that the identifiers should have at least one letter or underscore (_). Then the following identifiers are valid:

```
0X      0_      0123456789L
```

Modify method `nextToken` to implement these changes.

28. The symbol table can be made using

1. a stack implemented as a linked list of dynamically allocated objects;
2. a stack implemented as an array. Each array element points to an object of a newly created class `Token` or its subclasses `Identifier` and `Keyword`. Assume that the array length is enough for any compilation;
3. a hash table implemented as an array in which each entry points to a stack composed by a linked list of dynamically-allocated objects;
4. the same hash table as item 3 but with objects of the same lexical level linked by a linked list. There would be a pointer `level[0]` pointing to a linked list with all objects of lexical level 0. This list would be independent of the list used in the hash table. Of course, there would be lists `level[1]`, `level[2]`, etc. The lexical level `0` corresponds to the global level. Level `1` to local variables, level 2 to variables declared inside a block of a subroutine, and so on;
5. a hash table as in item 3 but with a set `level[0]` containing the indices of the hash table of all lists that contain the symbols of lexical level 0.

Compare these implementations with relation to

- (a) the insertion of an element in the table;
- (b) the search for an element;
- (c) the elimination of a lexical level;
- (d) the amount of memory used.

This comparison cannot be completely precise because we did not supply all the implementation details.

29. Draw the AST of the following program.

```
 var i, n : integer;
     escreveu : boolean;
begin
```

```
escreveu = false;
i = 1;
read(n);
if not ( n <= 1 )
then
   escreveu = true;
   while i <= n do
      begin
      write( i );
      i = i + 1;
      end;
endif;
end
```

It is not necessary to supply the grammar or language definition. Just imagine the missing information and use your good sense.

30. When the parser finds a sequence of letters, it should discover whether these letters are a keyword or identifier. There are two ways to do that:

1. by searching a data structure containing all keywords. If nothing is found, a new search is made in the symbol table;

2. by searching the symbol table directly. In this case, the keywords should have been inserted in this table before the start of compilation.

Discuss the advantages of 1 over 2 and vice-versa. In 1, the data structure may be a hashtable, a binary tree, or anything else.

31. Explique porque várias classes herdam de `Statement`. A sua resposta deve justificar porque é melhor utilizar esta herança do que não utilizá-la. Em que o compilador deveria ser modificado se as classes `AssignmentStatement`, `IfStatement`, `ReadStatement` e `WriteStatement` não possuissem superclasses? Naturalmente, classes que não herdam de ninguém herdam de `Object`.

32. O método abaixo pertence à classe `StatementList`.

```
public void genC() {
  if ( v != null ) {
     Enumeration e = v.elements();
     while ( e.hasMoreElements() )
       ((Statement ) e.nextElement()).genC();
  }
```

Explique como ele funciona. Isto é:

(a) qual é o tipo de retorno declarado do método `nextElement` de `Enumeration`? Este é o tipo obtido em compilação;

(b) em tempo de execução, "`e.nextElement()`" retornará objetos de que classes? Alguma delas é "`Statement`"? Por que é feita uma conversão do tipo de `nextElement()` para `Statement` ?

(c) o que fazem os métodos `hasMoreElements` e `nextElement`?

33. Que conferências semânticas são feitas no compilador 8?

34. Quais são os possíveis erros na análise léxica no compilador 8?

35. Ao inicializar a tabela de palavras-chave da linguagem, inserimos objetos da classe `Integer` na tabela:

```
keywordsTable = new Hashtable();
keywordsTable.put( "var", new Integer(Symbol.VAR) );
keywordsTable.put( "begin", new Integer(Symbol.BEGIN) );
...
```

Explique porque não inserimos inteiros como `Symbol.VAR` e `Symbol.BEGIN`.

36. Por que representamos a declaração de uma variável por objeto da classe `Variable` e uso da variável em uma expressão por um objeto de `VariableExpr`?

37. Quando token é `Symbol.IDENT`, sabemos que o token corrente é um identificador (variável). Como sabemos qual a string correspondente, isto é, qual o nome da variável?

38. Quando token é `Symbol.NUMBER`, como descobrimos qual é o valor do número?

39. Por que depois de encontrado um comentário o analisador léxico chama a si mesmo recursivamente?

40. No analisador léxico, porque utilizamos um objeto de `StringBuffer` e não de `String` para coletar os caracteres de um identificador (ou número)?

41. Quando uma seqüência de letras é encontrada pelo analisador léxico, como descobrimos se ela é uma palavra-chave? E como descobrimos o número correspondente a esta palavra-chave?

42. Por que no método `ReadStatement::genC` utilizamos `gets` seguido de `sscanf` ao invés de `scanf`?

43. Por que a classe `Statement` possui um método `genC` abstrato?

9 Compiler 9

This compiler differs from the previous one in several important points:

- the lexical analyzer was separated from the syntactical analyzer. It was put in class Lexer. Method `nextToken` is now called as "`lexer.nextToken()`" in which `lexer` is an instance variable of `Compiler`, initialized from a parameter of the constructor. The token is got by "`lexer.token`";
- the error treatment was also removed from the syntactical analyzer. It is in class `CompilerError`. Object `lexer` has a pointer to the error object and vice-versa;
- variables now have types. They can be `integer`, `boolean`, and `char`;
- the output of the program in C is correctly indented;
- the output of the program is made to a file and it can be made to any stream;
- the error messages can be directed to any stream;
- the variable declaration is now more Pascal-like;
- the arithmetical and logical operators are infix now:

```
a = 2*b + 3;
if a > b then a = 1; endif;
```

The new grammar is:

```
Program ::= [ "var" VarDecList ] CompositeStatement
CompositeStatement ::= "begin" StatementList "end"
StatementList ::= | Statement ";" StatementList
Statement ::= AssignmentStatement | IfStatement | ReadStatement |
              WriteStatement
AssignmentStatement ::= Ident "="  OrExpr
IfStatement ::= "if" OrExpr "then" StatementList [ "else" StatementList ] "endif"
ReadStatement ::= "read" "(" Ident ")"
WriteStatement ::= "write" "(" OrExpr ")"
VarDecList ::= VarDecList2  { VarDecList2  }
VarDecList2 ::= Ident { ',' Ident } ':' Type ';'
Ident ::= Letter { Letter }
Type ::= "integer" | "boolean" | "char"
OrExpr ::= AndExpr [ "or" AndExpr ]
AndExpr ::= RelExpr [ "and" RelExpr ]
RelExpr ::= AddExpr [ RelOp AddExpr ]
AddExpr ::= MultExpr { AddOp MultExpr }
MultExpr ::= SimpleExpr { MultOp SimpleExpr }
SimpleExpr ::= Number | Ident | "true" | "false" | Character
            | '(' OrExpr ')' | "not" SimpleExpr | AddOp SimpleExpr
RelOp ::= '<' | '<=' | '>' | '>='| '==' | '<>'
AddOp ::= '+'| '-'
```

```
MultOp ::= '*' | '/' | '%'
Number ::= ['+'|'-'] Digit { Digit }
Digit ::= '0'| '1' | ... | '9'
Letter ::= 'A' | 'B'| ... | 'Z'| 'a'| 'b' | ... | 'z'
```

An example of code in this language is

```
 var
   i, j : integer;
   achou : boolean;
   ch : char;
begin
i = 1;
j = i*3 - 4%i + 3*2*i/2;
if i + 1 > j - 3 and i <= j + 5 or 4 < i
then
   write(i);
   write(j);
endif;
ch = 'a';
achou = false;
if ch >= 'b' and not achou
then
 read(ch);
endif; end
```

There is a class `Type` which just keeps 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 String getCname();

    private String name;
}
```

The subclasses `BooleanType`, `IntegerType`, and `CharType` of `Type` represent the basic types `boolean`, `integer`, and `char`. Each time an `integer` type is found, an object of class `IntegerType` representing `integer` should be used. Using the example

```
var n : integer;
    p : integer;
    t : integer;
```

a naive approach would create three objets of class `IntegerType`. However, these objects would be equal to each other. A smarter approach uses the static variable `integerType` declared in class `Type`. Only one object representing type `integer` needs to be created. See method `Compiler::type` which is responsible for returning an object representing a type:

```
    private Type type() {
        Type result;

        switch ( lexer.token ) {
            case INTEGER :
              result = Type.integerType;
              break;
            case BOOLEAN :
              result = Type.booleanType;
              break;
            case CHAR :
              result = Type.charType;
              break;
            default :
              error.signal("Type expected");
              result = null;
        }
        lexer.nextToken();
        return result;
    }
```

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

```
public class CompositeExpr extends Expr {

    public CompositeExpr( Expr pleft, Symbol poper, Expr pright ) {
        left = pleft;
        oper = poper;
        right = pright;
    }
    public void genC( PW pw ) {
        pw.out.print("(");
        left.genC(pw);
        pw.out.print(" " + oper.toString() + " ");
```

```
        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 ||
            oper == Symbol.AND || oper == Symbol.OR )
            return Type.booleanType;
       else
            return Type.integerType;
    }

    private Expr left, right;
    private Symbol oper;
}

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 arithmetical expressions. For example, two values should have type `integer` if they are added or subtracted. See method `addExpr` below.

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

   */
   Symbol  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;
 }
```

The complete compiler code is in Appendix D.

Exercises

44. Answer the following questions:

- when objects representing variables are removed from the symbol table?
- is it possible that an object is referenced by the symbol table and the AST simultaneously?
- are there objects that will never be part of the AST although these objects are referenced by the symbol table? And the opposite?
- are objects representing types `integer` and `boolean` and constants `true` and `false` inserted in the symbol table? Are they referenced by the AST?

45. Quais são as conferências semânticas feitas neste compilador?

46. Na declaração de uma variável, o compilador insere um objeto de `Variable` na tabela de símbolos. Este objeto representa a variável. Veja o código abaixo.

```
Variable v = new Variable(name);
symbolTable.put( name, v );
```

Por que não inserir simplesmente o nome da variável? Este objeto de `Variable` que foi colocado na tabela é utilizado mais tarde em algum lugar? Quando, no código acima, o objeto apontado por v é inserido em `symbolTable`, ele não possui tipo. Este tipo é fornecido ao objeto em algum lugar? Depois de fornecido o tipo, o objecto da tabela de símbolos não continuará sem um tipo?

47. O método `Compiler::type` reconhece o tipo de uma variável :

```
private Type type() {
    Type result;

    switch ( lexer.token ) {
        case INTEGER :
          result = Type.integerType;
          break;
        case BOOLEAN :
           ...
    }
}
```

Por que este método não cria e retorna um objeto de `IntegerType`, `BooleanType` ou `CharType`? Algo como

```
private Type type() {
    Type result;

    switch ( lexer.token ) {
        case INTEGER :
          result = new IntegerType();
          break;
        case BOOLEAN :
           ...
    }
}
```

48. Explique porque é necessária uma classe `Type` superclasse de `IntegerType`, `BooleanType` e `CharType`.

49. Por que a classe `CompilerError` possui uma variável de instância da classe `Lexer`? Quando esta variável é necessária?

50. Por que `Expr` e suas subclasses precisam ter um método `getType`?

51. Faça o método `getType` de `CompositeExpr`.

52. Quais são os novos erros léxicos que este compilador pode sinalizar?

53. Por que a classe `Type` declara as variáveis `booleanType`, `integerType` e `charType` como `static`?

54. A classe `UnaryExpr` possui um `switch` com três "cases" no método `genC`. Isto indica que esta classe está desempenhando três papéis diferentes. Não seria melhor desdobrá-la em três classes diferentes?

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. There are new statements: `for`, `while`, `return`, composite statement, and procedure call. The program must have a parameterless procedure called `main`. The new Grammar is

```
Program ::= ProcFunc { ProcFunc }
ProcFunc ::= Procedure | Function
Procedure ::= "procedure" Ident '(' ParamList ')'
    [ LocalVarDec ] CompositeStatement
Function ::= "function" Ident '(' ParamList ')' ':' Type
    [ LocalVarDec ] CompositeStatement
ParamList ::= | ParamDec { ';' ParamDec }
ParamDec ::= Ident { ',' Ident } ':' Type
LocalVarDec ::= "var" VarDecList
CompositeStatement ::= "begin" StatementList "end"
StatementList ::= | Statement ";" StatementList
Statement ::= AssignmentStatement | IfStatement | ReadStatement |
   WriteStatement | ProcedureCall | ForStatement | WhileStatement |
   CompositeStatement | ReturnStatement
AssignmentStatement ::= Ident "="  OrExpr
IfStatement ::= "if" OrExpr "then" StatementList [ "else" StatementList ] "endif"
ReadStatement ::= "read" "(" Ident ")"
WriteStatement ::= "write" "(" OrExpr ")"
ProcedureCall ::= Ident '(' ExprList ')'
ExprList ::=  | OrExpr { ',' OrExpr }
ForStatement ::= "for" Ident '=' OrExpr "to" OrExpr "do" Statement
WhileStatement ::= "while" OrExpr "do" Statement
ReturnStatement ::= "return" OrExpr

VarDecList ::= VarDecList2  { VarDecList2  }
VarDecList2 ::= Ident { ',' Ident } ':' Type ';'
Ident ::= Letter { Letter }
Type ::= "integer" | "boolean" | "char"
OrExpr ::= AndExpr [ "or" AndExpr ]
AndExpr ::= RelExpr [ "and" RelExpr ]
RelExpr ::= AddExpr [ RelOp AddExpr ]
AddExpr ::= MultExpr { AddOp MultExpr }
MultExpr ::= SimpleExpr { MultOp SimpleExpr }
SimpleExpr ::= Number | "true" | "false" | Character
    | '(' OrExpr ')' | "not" SimpleExpr | AddOp SimpleExpr
    | Ident [ '(' ExprList ')' ]
RelOp ::= '<' | '<=' | '>' | '>='| '==' | '<>'
AddOp ::= '+'| '-'
MultOp ::= '*' | '/' | '%'
```

```
Number ::= ['+'|'-'] Digit { Digit }
Digit ::= '0'| '1' | ... | '9'
Letter ::= 'A' | 'B'| ... | 'Z'| 'a'| 'b' | ... | 'z'
```

An example of code in this language is given below.

```
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;
  while i <= 50 do
    i = i + 1;
  for i = 1 to 10 do
    x = x + 1;
  for i = 1 to 10 do
     begin
     x = x + 1;
     y = y + 1;
     end;
  end

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


procedure main()
  var n : integer;
  begin
  write( fatorial(5) );
  print(0, 1, 'A');
  for n = 1 to 10 do
    write(n);
  end
```

The symbol table has two hash tables: one for local and other for global symbols. Variables and

parameters `x`, `y`, `ch`, `i`, and `n` are local symbols. `print`, `factorial`, and `main` are global. Only procedures and functions can be global in this language. Parameters and local variables are always local. Procedure `print` is represented by an object of class `Procedure` of the AST. Function `factorial` is represented by an object of class `Function`. Both `Procedure` and `Function` are subclasses of class `Subroutine` — see the source code in Appendix E. Class `Parameter` of the AST inherits from class `Variable`. We need to define new constructors for `Parameter` since constructors are not inherited.

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`. This method is called at the end of method `procFunc` of class `Compiler`.

Each symbol is inserted at the symbol table when 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. It may be a variable/parameter or a procedure:

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

The compiler searches at the local and global table for the identifier. 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 method `statement` that follows. This method throws exception `StatementException` in error.

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

        */

        switch (lexer.token) {
            case 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 IF :
```

```
          return ifStatement();
        case READ :
          return readStatement();
        case WRITE :
          return writeStatement();
        case FOR :
          return forStatement();
        case WHILE :
          return whileStatement();
        case BEGIN :
          return compositeStatement();
        case RETURN :
          return returnStatement();
        default :
          error.show("Statement expected");
          throw new StatementException();
    }
}
```

Each program in language 10 must have a `main` procedure, which is checked by method `Compiler::program`:

```
private Program program() {
   // Program ::= ProcFunc { ProcFunc }

    ArrayList<Subroutine> procfuncList = new ArrayList<Subroutine>();

   while ( lexer.token == Symbol.PROCEDURE ||
           lexer.token == Symbol.FUNCTION )
       procfuncList.add( procFunc() );

   Program program = new Program( procfuncList );
   if ( lexer.token != Symbol.EOF )
     error.signal("EOF expected");
     // semantics analysis
     // there must be a procedure called main
   Subroutine mainProc;
   if ( (mainProc = (Subroutine ) symbolTable.getInGlobal("main")) == null )
      error.show("Source code must have a procedure called main");
   return program;
}
```

Exercises

55. A tabela de símbolos possui agora símbolos locais e globais. Diga quais símbols são locais e globais. Quando cada símbolo é inserido na tabela? Quando é retirado?

56. O programa deve ter um procedimento chamado `main` sem parâmetros. Como o compilador confere se o programa tem este procedimento?

57. Teria algum problema se o compilador permitisse um nome de procedimento igual ao nome de uma função?

58. Pode uma variável ter o mesmo nome que o procedimento/função onde ela está?

59. Que conferências semânticas são feitas neste compilador?

60. Por que precisamos da classe `Function` e também da classe `FunctionCall`?

61. Por que a classe `FunctionCall` deve herdar de `Expr`?

62. Por que a classe `Parameter` herda de `Variable`? Cite um lugar onde isto é necessário.

63. Por que a classe `ProcedureCall` deve herdar de `Statement`?

64. Onde é usado a classe `UndefinedType`?

A The Complete Code of Compiler 6

This Appendix gives the complete code of compiler 6.

```
// comp 6

import AST.*;

public class Main {
    public static void main( String []args ) {
        char []input = "a = 1  b = 3 : (- (+ a 2) b)".toCharArray();

        Compiler compiler = new Compiler();

        Program program  = compiler.compile(input);
        program.genC();
    }
}


// ##############################################################
/*
    comp6


    We added to the grammar more operators and a declaration of variables so that
    the grammar now accepts a program like
          a = 1  b = 3  : (- (+ a 2) 3)
    The result of the evaluation would be 0. Another program would be
          g = 3  t = 9  : (* (- t 7) g)

    Of course, the AST was extended to cope with the new rules. New classes
    were created:
         Program - represents the program
         Variable - a variable in the declaration
         VariableExpr - a variable inside an expression

    Grammar:
       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'
```

```
*/

import AST.*;
import java.util.*;


public class Compiler {

    public Program compile( char []p_input ) {
        input = p_input;
        tokenPos = 0;
        nextToken();
        Program result = program();
        if ( token != '\0' )
          error();
        return result;
        }

    private Program program() {
        ArrayList<Variable> arrayVariable = varDecList();

        if ( token != ':' ) {
            error();
            return null;
        }
        else {
            nextToken();
            Expr e = expr();
            return new Program( arrayVariable, e );
        }
    }

    private ArrayList<Variable> 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 */
         ArrayList<Variable> arrayVariable = new ArrayList<Variable>();
         while ( Character.isLetter(token) )
           arrayVariable.add( varDec() );
         return arrayVariable;
    }


    private Variable varDec() {
        char name = letter();
```

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

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

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());
}

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

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

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

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 - 1;

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

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

}


// ###############################################################
package AST;

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;
}


// ###############################################################
package AST;

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


// ###############################################################
package AST;

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);
    }
    private char n;
}


// ###################################################################
package AST;

import java.util.*;

public class Program {
    public Program( ArrayList<Variable> arrayVariable, Expr expr ) {
        this.arrayVariable = arrayVariable;
        this.expr = expr;
    }
    public void genC() {
        System.out.println("#include <stdio.h>\n");
        System.out.println("void main() {");
        for ( Variable v : arrayVariable )
            v.genC();


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

    private ArrayList<Variable> arrayVariable;
    private Expr expr;
}


// ###################################################################
package AST;

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;
}


// ###################################################################
package AST;

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

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

    private char name;
}
```

B The Complete Code of Compiler 7

This Appendix gives the complete code of compiler 7.

```
// comp7

import AST.*;

public class Main {
    public static void main( String []args ) {
        char []input = "a = 1  b = 3 : (- (+ a 2) b)".toCharArray();

        Compiler compiler = new Compiler();

        Program program  = compiler.compile(input);
        program.genC();
        System.out.println("\nValue = " + program.eval() );
    }
}


// ###################################################################
/*
    comp7

    Class CompilerRuntime models the runtime system. Its method "error"
    signals a error at runtime and terminates the program.

    Class Expr and its subclasses now have a method "eval" to evaluate the
    expression. An expression can be a letter whose value was previously
    assigned before the ':'  --- see the grammar. Then it is necessary to use
    a symbol table to keep the variables with its values. The Symbol Table is
    pointed by variable symbolTable.

    Grammar:
       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'


*/
```

```
import AST.*;
import java.util.*;

public class Compiler {

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

    private Program program() {
        ArrayList<Variable> arrayVariable = varDecList();
        if ( token != ':' ) {
            error(": expected");
            return null;
        }
        else {
            nextToken();
            Expr e = expr();
            return new Program( arrayVariable, e );
        }
    }

    private ArrayList<Variable> 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 */
         ArrayList<Variable> arrayVariable = new ArrayList<Variable>();
         while ( Character.isLetter(token) )
           arrayVariable.add( varDec() );
         return arrayVariable;
    }


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

```
    else {
      nextToken();
      NumberExpr num = number();
        // semantic analysis
        // inserts the variable in the Symbol Table. It will be used to
        // retrieve the value of the variable when it is found inside the
        // expression and to check if the variable found in the expression
        // was "declared" before the ':'
        // Note we insert the name converted to a String in the table
      Variable v = new Variable( name, num.eval() );
      if ( symbolTable.put( name, v ) != null )
        error("Variable " + name + " has already been declared");
      return v;
    }
}

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

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();
}


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

private NumberExpr number() {
    NumberExpr e = null;

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

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

private void nextToken() {
    while (  tokenPos < input.length && input[tokenPos] == ' ' )
      tokenPos++;
    if ( tokenPos < input.length ) {
```

```
          token = input[tokenPos];
          tokenPos++;
        }
        else
          token = '\0';
    }

    private void error( String strMessage ) {
        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 = "At \"" + strInput + "\"";
        System.out.print( strError );
        System.out.println( strMessage );
        throw new RuntimeException(strError);
    }

      // current token
    private char token;
    private int  tokenPos;
      // program given as input - source code
    private char []input;

    private Hashtable<Character, Variable> symbolTable;


}


// ###################################################################

public class CompilerRuntime {
    public static void error( String s ) {
        System.out.println("Runtime error: " + s);
        throw new RuntimeException(s);
    }
}


// ###################################################################
package AST;
```

```
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;
}
// ###############################################################
package AST;

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

```
// ##################################################################
package AST;

public class NumberExpr extends Expr {

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

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

    public int eval() {
        return n;
    }

    private int n;
}

// ##################################################################
package AST;

import java.util.*;

public class Program {
    public Program( ArrayList<Variable> arrayVariable, Expr expr ) {
        this.arrayVariable = arrayVariable;
        this.expr = expr;
    }
    public void genC() {
        System.out.println("#include <stdio.h>\n");
        System.out.println("void main() {");

          // generate code for the declaration of variables
        for ( Variable v : arrayVariable )
            v.genC();

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

    public int eval() {
```

```
        return expr.eval();
    }

    private ArrayList<Variable> arrayVariable;
    private Expr expr;
}

// ###################################################################
package AST;

public class Variable {

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

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

    public int eval() { return value; }

    public String getName() { return name + ""; }

    private char name;
    private int value;
}

// ###################################################################
package AST;

public class VariableExpr extends Expr {

    public VariableExpr( Variable variable ) {
        this.variable = variable;
    }

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

    public int eval() {
        return variable.eval();
    }
```

```
    private Variable variable;

}
```

C The Complete Code of Compiler 8

This Appendix gives the complete code of compiler 8.

```
import AST.*;
import java.io.*;

public class Main {
    public static void main( String []args ) {

        File file;
        FileReader stream;
        int numChRead;

        if ( args.length != 1 )
            System.out.println("Use only one parameter, the file to be compiled");
        else {
           file = new File(args[0]);
           if ( ! file.exists() || ! file.canRead() ) {
             System.out.println("Either the file " + args[0] + " does not exist or it cannot be
             throw new RuntimeException();
           }
           try {
             stream = new FileReader(file);
           } catch ( FileNotFoundException e ) {
               System.out.println("Something wrong: file does not exist anymore");
               throw new RuntimeException();
           }
               // one more character for '\0' at the end that will be added by the
               // compiler
           char []input = new char[ (int ) file.length() + 1 ];

           try {
             numChRead = stream.read( input, 0, (int ) file.length() );
           } catch ( IOException e ) {
               System.out.println("Error reading file " + args[0]);
               throw new RuntimeException();
           }

           if ( numChRead != file.length() ) {
               System.out.println("Read error");
               throw new RuntimeException();
           }
           try {
             stream.close();
           } catch ( IOException e ) {
```

```
                System.out.println("Error in handling the file " + args[0]);
                throw new RuntimeException();
            }


            Compiler compiler = new Compiler();

            Program program  = compiler.compile(input);
            program.genC();
            }
    }
}


// ####################################################################
/*
    comp8

    Variables now can have any number of characters and numbers any number
    of digits. There are new keywords and new non-terminals. The operator
    set includes the comparison operators. There are a few statements.
    Anything after // till the end of the line is a comment.
    Note that VarDecList was modified.

    The input is now taken from a file.
    Method error now prints the line in which the error occurred.


    Grammar:
       Program ::= [ "var" VarDecList ";" ] CompositeStatement
       CompositeStatement ::= "begin" StatementList "end"
       StatementList ::= | Statement ";" StatementList
       Statement ::= AssignmentStatement | IfStatement | ReadStatement |
          WriteStatement
       AssignmentStatement ::= Variable "="  Expr
       IfStatement ::= "if" Expr "then" StatementList [ "else" StatementList ] "endif"
       ReadStatement ::= "read" "(" Variable ")"
       WriteStatement ::= "write" "(" Expr ")"

       Variable ::= Letter { Letter }
       VarDecList ::= Variable | Variable "," VarDecList
       Expr::=  '(' oper Expr  Expr ')' | Number | Variable
       Oper ::= '+' | '-' | '*' | '/' | '<' | '<=' | '>' | '>='| '==' | '<>'
       Number ::= Digit { Digit }
       Digit ::= '0'| '1' | ... | '9'
       Letter ::= 'A' | 'B'| ... | 'Z'| 'a'| 'b' | ... | 'z'
```

```
   Anything between [] is optional. Anything between { e } can be
   repeated zero or more times.
*/

import AST.*;
import java.util.*;
import Lexer.*;


public class Compiler {

      // contains the keywords
    static private Hashtable<String, Symbol> keywordsTable;

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

     }



      // compile must receive an input with an character less than
      // p_input.lenght
    public Program compile( char []p_input ) {
        input = p_input;
          // add an end-of-file label to make it easy to do the lexer

        input[input.length - 1] = '\0';
          // number of the current line
        lineNumber = 1;
        tokenPos = 0;
          // symbol table. Will contain the declared variable
        symbolTable = new Hashtable<String, Variable>();
        nextToken();
        return program();
```

```
    }

private Program program() {
    // Program ::= [ "var" VarDecList ] CompositeStatement

    ArrayList<Variable> arrayVariable = null;

    if ( token == Symbol.VAR )  {
      nextToken();
      arrayVariable = varDecList();
      if ( token != Symbol.SEMICOLON )
        error("; expected");
      nextToken();
    }
    Program program = new Program( arrayVariable, compositeStatement() );
    if ( token != Symbol.EOF )
      error("EOF expected");
    return program;
}


private StatementList compositeStatement() {
      //  CompositeStatement ::= "begin" StatementList "end"
      //  StatementList ::= | Statement ";" StatementList

    if ( token != Symbol.BEGIN )
        error("BEGIN expected");
    nextToken();
    StatementList sl = statementList();
    if ( token != Symbol.END )
      error("\"end\" expected");
    nextToken();
    return sl;
}

private StatementList statementList() {
    ArrayList<Statement> v = new ArrayList<Statement>();
      // statements always begin with an identifier, if, read or write
    while ( token == Symbol.IDENT ||
            token == Symbol.IF ||
            token == Symbol.READ ||
            token == Symbol.WRITE ) {
        v.add( statement() );
        if ( token != Symbol.SEMICOLON )
          error("; expected");
        nextToken();
```

```
        }
    return new StatementList(v);
}

private Statement statement() {
    /*  Statement ::= AssignmentStatement | IfStatement | ReadStatement |
                      WriteStatement
    */

    switch (token) {
        case IDENT :
          return assignmentStatement();
        case IF :
          return ifStatement();
        case READ :
          return readStatement();
        case WRITE :
          return writeStatement();
        default :
            // will never be executed
          error("Statement expected");
    }
    return null;
}

private AssignmentStatement assignmentStatement() {

      // the current token is Symbol.IDENT and stringValue
      // contains the identifier
    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");
      // eat token Symbol.IDENT
    nextToken();
    if ( token != Symbol.ASSIGN )
      error("= expected");
    nextToken();
    return new AssignmentStatement( v, expr() );
}
```

```
private IfStatement ifStatement() {
    nextToken();
    Expr e = expr();
    if ( token != Symbol.THEN )
      error("then expected");
    nextToken();
    StatementList thenPart = statementList();
    StatementList elsePart = null;
    if ( token == Symbol.ELSE ) {
        nextToken();
        elsePart = statementList();
    }
    if ( token != Symbol.ENDIF )
      error("\"endif\" expected");
    nextToken();
    return new IfStatement( e, thenPart, elsePart );
}



private ReadStatement readStatement() {
    nextToken();
    if ( token != Symbol.LEFTPAR )
      error("( expected");
    nextToken();
    if ( token != Symbol.IDENT )
      error("Identifier expected");
      // check if the variable was declared
    String name = stringValue;
    Variable v = (Variable ) symbolTable.get(name);
    if ( v == null )
        error("Variable " + name + " was not declared");
    nextToken();
    if ( token != Symbol.RIGHTPAR )
      error(") expected");
    nextToken();
    return new ReadStatement( v );
}


private WriteStatement writeStatement() {
    nextToken();
    if ( token != Symbol.LEFTPAR )
      error("( expected");
```

```
    nextToken();
    Expr e = expr();
    if ( token != Symbol.RIGHTPAR )
      error(") expected");
    nextToken();
    return new WriteStatement( e );
}



private ArrayList<Variable> varDecList() {
    //   VarDecList ::= Variable | Variable "," VarDecList ";"
    ArrayList<Variable> v = new ArrayList<Variable>();
    Variable variable = varDec();
    v.add(variable);
    while ( token == Symbol.COMMA ) {
      nextToken();
      v.add( varDec() );
    }
    return v;
}



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 has 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;
}
```

```
private Expr expr() {
    if ( token == Symbol.LEFTPAR ) {
        nextToken();
        Symbol op = token;
        if ( op == Symbol.EQ || op == Symbol.NEQ || op == Symbol.LE || op == Symbol.LT ||
             op == Symbol.GE || op == Symbol.GT  || op == Symbol.PLUS ||
             op == Symbol.MINUS || op == Symbol.MULT || op == Symbol.DIV )
             nextToken();
        else
          error("operator expected");
        Expr e1 = expr();
        Expr e2 = expr();
        CompositeExpr ce = new CompositeExpr(e1, op, e2);
        if ( token == Symbol.RIGHTPAR )
          nextToken();
        else
          error(") expected");
        return ce;
    }
    else
        // note we test the token to decide which production to use
      if ( token == Symbol.NUMBER )
        return number();
      else {
        if ( token != Symbol.IDENT )
          error("Identifier expected");
        String name = stringValue;
        nextToken();
        Variable v = (Variable ) symbolTable.get( name );
          // semantic analysis
          // was the variable declared ?
        if ( v == null )
          error("Variable " + name + " was not declared");
        return new VariableExpr(v);
      }

}



private NumberExpr number() {
    NumberExpr e = null;

    if ( token != Symbol.NUMBER )
      error("Number expected");  // in the current version, never occurs
```

```
      // the number value is stored in token.value as an object of Integer.
      // Method intValue returns that value as an value of type int.
    int value = numberValue;
    nextToken();
    return new NumberExpr(value);
}


private void nextToken() {
    char ch;

    while (  (ch = input[tokenPos]) == ' ' || ch == '\r' ||
             ch == '\t' || ch == '\n')  {
        // count the number of lines
      if ( ch == '\n')
        lineNumber++;
      tokenPos++;
      }
    if ( ch == '\0')
      token = Symbol.EOF;
    else
         // skip comments
      if ( input[tokenPos] == '/' && input[tokenPos + 1] == '/' ) {
            // comment found
           while ( input[tokenPos] != '\0'&& input[tokenPos] != '\n' )
             tokenPos++;
           nextToken();
           }
      else {
        if ( Character.isLetter( ch ) ) {
              // get an identifier or keyword
              // StringBuffer represents a string that can grow
            StringBuffer ident = new StringBuffer();
              // is input[tokenPos] a letter ?
              // isLetter is a static method of class Character
            while ( Character.isLetter( input[tokenPos] ) ) {
                  // add a character to ident
                ident.append(input[tokenPos]);
                tokenPos++;
            }
              // convert a StringBuffer object into a
              // String object
            stringValue = ident.toString();
              // if identStr is in the list of keywords, it is a keyword !
            Symbol value = keywordsTable.get(stringValue);
            if ( value == null )
```

```
       token = Symbol.IDENT;
    else
       token = value;
    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;
    try {
         /*
            number.toString() converts a StringBuffer
            into a String object.
            valueOf converts a String object into an
            Integer object. intValue gets the int
            inside the Integer object.
         */
       numberValue = Integer.valueOf(number.toString()).intValue();
    } catch ( NumberFormatException e ) {
       error("Number out of limits");
    }
    if ( numberValue >= MaxValueInteger )
       error("Number out of limits");
    if ( Character.isLetter(input[tokenPos]) )
      error("Number followed by a letter");
}
else {
    tokenPos++;
    switch ( ch ) {
        case '+' :
          token = Symbol.PLUS;
          break;
        case '-' :
          token = Symbol.MINUS;
          break;
        case '*' :
          token = Symbol.MULT;
          break;
        case '/' :
          token = Symbol.DIV;
          break;
```

```
                case '<' :
                  if ( input[tokenPos] == '=' ) {
                    tokenPos++;
                    token = Symbol.LE;
                  }
                  else if ( input[tokenPos] == '>' ) {
                    tokenPos++;
                    token = Symbol.NEQ;
                  }
                  else
                    token = Symbol.LT;
                  break;
                case '>' :
                  if ( input[tokenPos] == '=' ) {
                    tokenPos++;
                    token = Symbol.GE;
                  }
                  else
                    token = Symbol.GT;
                  break;
                case '=' :
                  if ( input[tokenPos] == '=' ) {
                    tokenPos++;
                    token = Symbol.EQ;
                  }
                  else
                    token = Symbol.ASSIGN;
                  break;
                case '(' :
                  token = Symbol.LEFTPAR;
                  break;
                case ')' :
                  token = Symbol.RIGHTPAR;
                  break;
                case ',' :
                  token = Symbol.COMMA;
                  break;
                case ';' :
                  token = Symbol.SEMICOLON;
                  break;
                default :
                  error("Invalid Character: '" + ch + "'");
            }
          }
        }
}
```

```
    private void error( String strMessage ) {
        if ( tokenPos == 0 )
          tokenPos = 1;
        else
          if ( tokenPos >= input.length )
            tokenPos = input.length;

        StringBuffer line = new StringBuffer();
          // go to the beginning of the line
        int i = tokenPos;
        while ( i >= 1 && input[i] != '\n' )
          i--;
        if ( input[i] == '\n' )
          i++;
          // go to the end of the line putting it in variable line
        while ( input[i] != '\0' && input[i] != '\n' && input[i] != '\r' ) {
            line.append( input[i] );
            i++;
        }

        System.out.println("Error at line " + lineNumber + ": ");
        System.out.println(line);
        System.out.println( strMessage );
        throw new RuntimeException(strMessage);
    }

      // current token
    private Symbol token;
    private String stringValue;
    private int numberValue;
    private int  tokenPos;
      // program given as input - source code
    private char []input;

    private Hashtable<String, Variable> symbolTable;

    // number of current line. Starts with 1
    private int lineNumber;
    private static final int MaxValueInteger = 32768;

}

// ##################################################################
package Lexer;
```

```
public enum Symbol {
      EOF("eof"),
      IDENT("Identifier"),
      NUMBER("Number"),
      PLUS("+"),
      MINUS("-"),
      MULT("*"),
      DIV("/"),
      LT("<"),
      LE("<="),
      GT(">"),
      GE(">="),
      NEQ("!="),
      EQ("=="),
      ASSIGN("="),
      LEFTPAR("("),
      RIGHTPAR(")"),
      SEMICOLON(";"),
      VAR("var"),
      BEGIN("begin"),
      END("end"),
      IF("if"),
      THEN("then"),
      ELSE("else"),
      ENDIF("endif"),
      COMMA(","),
      READ("read"),
      WRITE("write");

      Symbol(String name) {
          this.name = name;
      }
      public String toString() { return name; }
      public String name;
}


// ###############################################################
package AST;

public class AssignmentStatement extends Statement {

    public AssignmentStatement( Variable v, Expr expr ) {
        this.v = v;
        this.expr = expr;
    }
```

```
    public void genC() {
        System.out.print( v.getName() + " = " );
        expr.genC();
        System.out.println(";");
    }

    private Variable v;
    private Expr expr;
}

// ##############################################################
package AST;

import Lexer.*;

public class CompositeExpr extends Expr {

    public CompositeExpr( Expr pleft, Symbol poper, Expr pright ) {
        left = pleft;
        oper = poper;
        right = pright;
    }
    public void genC() {
        System.out.print("(");
        left.genC();
        System.out.print(" " + oper.toString() + " ");
        right.genC();
        System.out.print(")");
    }

    private Expr left, right;
    private Symbol oper;
}


// ##############################################################
package AST;

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


// ##############################################################
package AST;
```

```
public class IfStatement extends Statement {

    public IfStatement( Expr expr, StatementList thenPart, StatementList elsePart ) {
        this.expr = expr;
        this.thenPart = thenPart;
        this.elsePart = elsePart;
    }

    public void genC() {
        System.out.print("if ( ");
        expr.genC();
        System.out.println(" ) { ");
        if ( thenPart != null )
          thenPart.genC();
        System.out.println("}");
        if ( elsePart != null ) {
          System.out.println("else {");
          elsePart.genC();
          System.out.println("}");
        }
    }

    private Expr expr;
    private StatementList thenPart, elsePart;
}

// ###############################################################

package AST;

public class NumberExpr extends Expr {

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

    public int getValue() {
        return value;
    }

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

    private int value;
```

```
}

// ############################################################
package AST;

import java.util.*;

public class Program {

    public Program( ArrayList<Variable> arrayVariable, StatementList statementList ) {
        this.arrayVariable = arrayVariable;
        this.statementList = statementList;
    }

    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
            for ( Variable v : arrayVariable )
                v.genC();
        }

        statementList.genC();
        System.out.println("}");
    }

    private ArrayList<Variable> arrayVariable;
    private StatementList statementList;
}

// ############################################################
package AST;

public class ReadStatement extends Statement {

    public ReadStatement( Variable v ) {
        this.v = v;
    }

    public void genC() {
        System.out.println("{ char s[256]; gets(s); sscanf(s, \"%d\", &" + v.getName() + "); }"
    }

    private Variable v;
}
```

```
// ##############################################################
package AST;

abstract public class Statement {
    abstract public void genC();
}


// ##############################################################
package AST;

import java.util.*;

public class StatementList {

    public StatementList(ArrayList<Statement> v) {
        this.v = v;
    }

    public void genC() {
      if ( v != null ) {
         for ( Statement s : v )
             s.genC();
      }
    }

    private  ArrayList<Statement> v;
}

// ##############################################################
package AST;

public class Variable {

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

    public String getName() { return name; }

    public void genC() {
        System.out.println( "int " + name + ";" );
    }

    private String name;
```

```
}

// ############################################################
package AST;

public class VariableExpr extends Expr {

    public VariableExpr( Variable v ) {
        this.v = v;
    }

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

    private Variable v;
}

// ############################################################
package AST;

public class WriteStatement extends Statement {

    public WriteStatement( Expr expr ) {
        this.expr = expr;
    }

    public void genC() {
        System.out.print("printf(\"%d\", ");
        expr.genC();
        System.out.println(" );");
    }

    private Expr expr;
}
```

D The Complete Code of Compiler 9

This Appendix gives the complete code of compiler 9.

```
import AST.*; import java.io.*;

public class Main {

    public static void main( String []args ) {

        File file;
        FileReader stream;
        int numChRead;
        Program program;

        if ( args.length != 2 )  {
            System.out.println("Usage:\n   Main input output");
            System.out.println("input is the file to be compiled");
            System.out.println("output is the file where the generated code will be stored");
        }
        else {
           file = new File(args[0]);
           if ( ! file.exists() || ! file.canRead() ) {
             System.out.println("Either the file " + args[0] + " does not exist or it cannot be
             throw new RuntimeException();
           }
           try {
             stream = new FileReader(file);
            } catch ( FileNotFoundException e ) {
                System.out.println("Something wrong: file does not exist anymore");
                throw new RuntimeException();
            }
                // one more character for '\0' at the end that will be added by the
                // compiler
            char []input = new char[ (int ) file.length() + 1 ];

            try {
              numChRead = stream.read( input, 0, (int ) file.length() );
            } catch ( IOException e ) {
                System.out.println("Error reading file " + args[0]);
                throw new RuntimeException();
            }

            if ( numChRead != file.length() ) {
                System.out.println("Read error");
                throw new RuntimeException();
```

```
            }
            try {
              stream.close();
            } catch ( IOException e ) {
                System.out.println("Error in handling the file " + args[0]);
                throw new RuntimeException();
            }


            Compiler compiler = new Compiler();
            FileOutputStream  outputStream;
            try {
               outputStream = new FileOutputStream(args[1]);
            } catch ( IOException e ) {
                System.out.println("File " + args[1] + " could not be opened for writing");
                throw new RuntimeException();
            }
            PrintWriter printWriter = new PrintWriter(outputStream);
            program = null;
              // the generated code goes to a file and so are the errors
            try {
              program  = compiler.compile(input, printWriter );
            } catch ( RuntimeException e ) {
                System.out.println(e);
            }
            if ( program != null ) {
              PW pw = new PW();
              pw.set(printWriter);
              program.genC(pw);
              if ( printWriter.checkError() ) {
                  System.out.println("There was an error in the output");
              }
            }
        }
    }
}


// ##############################################################

import Lexer.*; import java.io.*;

public class CompilerError {

    public CompilerError( PrintWriter out ) {
          // output of an error is done in out
```

```
        this.out = out;
    }

    public void setLexer( Lexer lexer ) {
        this.lexer = lexer;
    }

    public void signal( String strMessage ) {
        out.println("Error at line " + lexer.getLineNumber() + ": ");
        out.println(lexer.getCurrentLine());
        out.println( strMessage );
        if ( out.checkError() )
          System.out.println("Error in signaling an error");
        throw new RuntimeException(strMessage);
    }

    private Lexer lexer;
    PrintWriter out;
}


// ############################################################
/*
    comp9

    Main changes from the previous compiler:
      - the lexical analyzer was separated from the syntactical analyzer.
        It was put in class Lexer. nextToken is now called by
           lexer.nextToken()
        where lexer is an instance variable of Compiler, initialized from a parameter
        of the constructor. The token is got by
          lexer.token
      - the error treatment was also removed from the syntactical analyzer. It is
        in class CompilerError;
      - variables now have types. They can be integer, boolean, and char. The rules for
        them are similar to Java;
      - the output of the program in C is correctly indented;
      - the output of the program is made to a file and it can be made to any stream;
      - the error messages can be directed to any stream;
      - the variable declaration is now more Pascal like;
      - the expressions resemble Java/C/C++


    Grammar:
 Program ::= [ "var" VarDecList ] CompositeStatement
 CompositeStatement ::= "begin" StatementList "end"
```

```
 StatementList ::= | Statement ";" StatementList
 Statement ::= AssignmentStatement | IfStatement | ReadStatement |
               WriteStatement
 AssignmentStatement ::= Ident "="  OrExpr
 IfStatement ::= "if" OrExpr "then" StatementList [ "else" StatementList ] "endif"
 ReadStatement ::= "read" "(" Ident ")"
 WriteStatement ::= "write" "(" OrExpr ")"
 VarDecList ::= VarDecList2  { VarDecList2  }
 VarDecList2 ::= Ident { ',' Ident } ':' Type ';'
 Ident ::= Letter { Letter }
 Type ::= "integer" | "boolean" | "char"
 OrExpr ::= AndExpr [ "or" AndExpr ]
 AndExpr ::= RelExpr [ "and" RelExpr ]
 RelExpr ::= AddExpr [ RelOp AddExpr ]
 AddExpr ::= MultExpr { AddOp MultExpr }
 MultExpr ::= SimpleExpr { MultOp SimpleExpr }
 SimpleExpr ::= Number | Ident | "true" | "false" | Character
             | '(' OrExpr ')' | "not" SimpleExpr | AddOp SimpleExpr
 RelOp ::= '<' | '<=' | '>' | '>='| '==' | '<>'
 AddOp ::= '+'| '-'
 MultOp ::= '*' | '/' | '%'
 Number ::= ['+'|'-'] Digit { Digit }
 Digit ::= '0'| '1' | ... | '9'
 Letter ::= 'A' | 'B'| ... | 'Z'| 'a'| 'b' | ... | 'z'

   Character is a Letter enclosed between ' and ', like 'A', 'e' as in Java, C++, etc.
   Anything between [] is optional. Anything between { e } can be
   repeated zero or more times.
*/

import AST.*; import java.util.*; import java.lang.Character;
import Lexer.*; import java.io.*;

public class Compiler {

      // compile must receive an input with an character less than
      // p_input.lenght
    public Program compile( char []input, PrintWriter outError ) {

        symbolTable = new Hashtable<String, Variable>();
        error = new CompilerError( outError );
        lexer = new Lexer(input, error);
        error.setLexer(lexer);

        lexer.nextToken();
        return program();
```

```
    }

private Program program() {
    // Program ::= [ "var" VarDecList ] CompositeStatement

    ArrayList<Variable> arrayVariable = null;

    if ( lexer.token == Symbol.VAR )  {
      lexer.nextToken();
      arrayVariable = varDecList();
    }
    Program program = new Program( arrayVariable, compositeStatement() );
    if ( lexer.token != Symbol.EOF )
      error.signal("EOF expected");
    return program;
}


private StatementList compositeStatement() {
      // CompositeStatement ::= "begin" StatementList "end"
      //  StatementList ::= | Statement ";" StatementList

    if ( lexer.token != Symbol.BEGIN )
        error.signal("BEGIN expected");
    lexer.nextToken();
    StatementList sl = statementList();
    if ( lexer.token != Symbol.END )
      error.signal("\"end\" expected");
    lexer.nextToken();
    return sl;
}

private StatementList statementList() {
    ArrayList<Statement> v = new ArrayList<Statement>();
      // statements always begin with an identifier, if, read or write
    while ( lexer.token == Symbol.IDENT ||
            lexer.token == Symbol.IF ||
            lexer.token == Symbol.READ ||
            lexer.token == Symbol.WRITE ) {
        v.add( statement() );
        if ( lexer.token != Symbol.SEMICOLON )
          error.signal("; expected");
        lexer.nextToken();
        }
    return new StatementList(v);
}
```

```
private Statement statement() {
    /*  Statement ::= AssignmentStatement | IfStatement | ReadStatement |
                      WriteStatement
    */

    switch (lexer.token) {
        case IDENT :
          return assignmentStatement();
        case IF :
          return ifStatement();
        case READ :
          return readStatement();
        case WRITE :
          return writeStatement();
        default :
            // will never be executed
          error.signal("Statement expected");
    }
    return null;
}

private AssignmentStatement assignmentStatement() {

    String name = lexer.getStringValue();

      // 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.signal("Variable " + name + " was not declared");
    lexer.nextToken();
    if ( lexer.token != Symbol.ASSIGN )
      error.signal("= expected");
    lexer.nextToken();
    Expr right = orExpr();
      // semantic analysis
      // check if expression has the same type as variable
    if ( v.getType() != right.getType() )
      error.signal("Type error in assignment");

    return new AssignmentStatement( v, right );
}
```

```
private IfStatement ifStatement() {

    lexer.nextToken();
    Expr e = orExpr();
      // semantic analysis
      // check if expression has type boolean
    if ( e.getType() != Type.booleanType )
      error.signal("Boolean type expected in if expression");

    if ( lexer.token != Symbol.THEN )
      error.signal("then expected");
    lexer.nextToken();
    StatementList thenPart = statementList();
    StatementList elsePart = null;
    if ( lexer.token == Symbol.ELSE ) {
        lexer.nextToken();
        elsePart = statementList();
    }
    if ( lexer.token != Symbol.ENDIF )
      error.signal("\"endif\" expected");
    lexer.nextToken();
    return new IfStatement( e, thenPart, elsePart );
}



private ReadStatement readStatement() {
    lexer.nextToken();
    if ( lexer.token != Symbol.LEFTPAR )
      error.signal("( expected");
    lexer.nextToken();
    if ( lexer.token != Symbol.IDENT )
      error.signal("Identifier expected");
      // semantic analysis
      // check if the variable was declared
    String name = lexer.getStringValue();
    Variable v = (Variable ) symbolTable.get(name);
    if ( v == null )
      error.signal("Variable " + name + " was not declared");
      // semantic analysis
      // check if variable has type char or integer
    if ( v.getType() != Type.charType && v.getType() != Type.integerType )
      error.signal("Variable should have type char or integer");
```

```
    lexer.nextToken();
    if ( lexer.token != Symbol.RIGHTPAR )
      error.signal(") expected");
    lexer.nextToken();
    return new ReadStatement( v );
}


private WriteStatement writeStatement() {
    lexer.nextToken();
    if ( lexer.token != Symbol.LEFTPAR )
      error.signal("( expected");
    lexer.nextToken();
      // expression may be of any type
    Expr e = orExpr();
    if ( lexer.token != Symbol.RIGHTPAR )
      error.signal(") expected");
    lexer.nextToken();
    return new WriteStatement( e );
}




private ArrayList<Variable> varDecList() {
    //  VarDecList ::= VarDecList2  { VarDecList2  }


    ArrayList<Variable> varList = new ArrayList<Variable>();

    varDecList2(varList);
    while ( lexer.token == Symbol.IDENT )
      varDecList2(varList);
    return varList;
}

private void varDecList2( ArrayList<Variable> varList ) {
    //  VarDecList2 ::= Ident { ',' Ident } ':' Type ';'

    ArrayList<Variable> lastVarList = new ArrayList<Variable>();

    while ( true ) {
      if ( lexer.token != Symbol.IDENT )
        error.signal("Identifier expected");
        // name of the identifier
      String name = lexer.getStringValue();
```

```
      lexer.nextToken();
        // semantic analysis
        // if the name is in the symbol table, the variable is been declared twice.
      if ( symbolTable.get(name) != null )
        error.signal("Variable " + name + " has already been declared");

        // variable does not have a type yet
      Variable v = new Variable(name);
        // 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 );
        // list of the last variables declared. They don't have types yet
      lastVarList.add(v);

      if ( lexer.token == Symbol.COMMA )
        lexer.nextToken();
      else
        break;
    }

    if ( lexer.token != Symbol.COLON ) //  :
      error.signal(": expected");
    lexer.nextToken();
      // get the type
    Type typeVar = type();

    for ( Variable v : lastVarList ) {
        // add type to the variable
      v.setType(typeVar);
        // add variable to the list of variable
      varList.add(v);
    }

    if ( lexer.token != Symbol.SEMICOLON )
      error.signal("; expected");
    lexer.nextToken();
}



private Type type() {
    Type result;

    switch ( lexer.token ) {
        case INTEGER :
```

```
          result = Type.integerType;
          break;
        case BOOLEAN :
          result = Type.booleanType;
          break;
        case CHAR :
          result = Type.charType;
          break;
        default :
          error.signal("Type expected");
          result = null;
    }
    lexer.nextToken();
    return result;
}


private Expr orExpr() {
  /*
    OrExpr ::= AndExpr [ "or" AndExpr ]
  */

  Expr left, right;
  left = andExpr();
  if ( lexer.token == Symbol.OR ) {
    lexer.nextToken();
    right = andExpr();
      // semantic analysis
    if ( left.getType() != Type.booleanType ||
         right.getType() != Type.booleanType )
       error.signal("Expression of boolean type expected");
    left = new CompositeExpr(left, Symbol.OR, right);
  }
  return left;
}

private Expr andExpr() {
  /*
    AndExpr ::= RelExpr [ "and" RelExpr ]
  */
  Expr left, right;
  left = relExpr();
  if ( lexer.token == Symbol.AND ) {
    lexer.nextToken();
    right = relExpr();
      // semantic analysis
```

```
    if ( left.getType() != Type.booleanType ||
         right.getType() != Type.booleanType )
       error.signal("Expression of boolean type expected");
    left = new CompositeExpr( left, Symbol.AND, right );
  }
  return left;
}

private Expr relExpr() {
  /*
    RelExpr ::= AddExpr [ RelOp AddExpr ]
  */
  Expr left, right;
  left = addExpr();
  Symbol op = lexer.token;
  if ( op == Symbol.EQ || op == Symbol.NEQ || op == Symbol.LE || op == Symbol.LT ||
       op == Symbol.GE || op == Symbol.GT  ) {
     lexer.nextToken();
     right = addExpr();
       // semantic analysis
     if ( left.getType() != right.getType() )
       error.signal("Type error in expression");
     left = new CompositeExpr( left, op, right );
   }
   return left;
}

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

  */
  Symbol  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;
```

```
}

private Expr multExpr() {
 /*
    MultExpr ::= SimpleExpr { MultOp SimpleExpr }
 */
   Expr left, right;
   left = simpleExpr();
   Symbol op;
   while ( (op = lexer.token) == Symbol.MULT ||
            op == Symbol.DIV || op == Symbol.REMAINDER ) {
      lexer.nextToken();
      right = simpleExpr();
        // 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;
}

private Expr simpleExpr() {
  /*
    SimpleExpr ::= Number |  "true" | "false" | Character
       | '(' Expr ')' | "not" SimpleExpr | Variable
  */

  Expr e;

    // note we test the lexer.getToken() to decide which production to use
  switch ( lexer.token ) {
     case NUMBER :
        return number();
     case TRUE :
       lexer.nextToken();
       return BooleanExpr.True;
     case FALSE :
       lexer.nextToken();
       return BooleanExpr.False;
     case CHARACTER :
         // get the token with getToken.
         // then get the value of it, with has the type Object
         // convert the object to type Character using a cast
         // call method charValue to get the character inside the object
       char ch = lexer.getCharValue();
```

```
        lexer.nextToken();
        return new CharExpr(ch);
      case LEFTPAR :
        lexer.nextToken();
        e = orExpr();
        if ( lexer.token != Symbol.RIGHTPAR )
          error.signal(") expected");
        lexer.nextToken();
        return e;
      case NOT :
        lexer.nextToken();
        e = orExpr();
          // semantic analysis
        if ( e.getType() != Type.booleanType )
          error.signal("Expression of type boolean expected");
        return new UnaryExpr( e, Symbol.NOT );
      case PLUS :
        lexer.nextToken();
        e = orExpr();
          // semantic analysis
        if ( e.getType() != Type.integerType )
          error.signal("Expression of type integer expected");
        return new UnaryExpr( e, Symbol.PLUS );
      case MINUS :
        lexer.nextToken();
        e = orExpr();
          // semantic analysis
        if ( e.getType() != Type.integerType )
          error.signal("Expression of type integer expected");
        return new UnaryExpr( e, Symbol.MINUS );
      default :
          // an identifier
        if ( lexer.token != Symbol.IDENT )
          error.signal("Identifier expected");
        String name = lexer.getStringValue();
        lexer.nextToken();
        Variable v = (Variable ) symbolTable.get( name );
          // semantic analysis
          // was the variable declared ?
        if ( v == null )
          error.signal("Variable " + name + " was not declared");
        return new VariableExpr(v);
    }

}
```

```
    private NumberExpr number() {

        NumberExpr e = null;

          // the number value is stored in lexer.getToken().value as an object of Integer.
          // Method intValue returns that value as an value of type int.
        int value = lexer.getNumberValue();
        lexer.nextToken();
        return new NumberExpr( value );
    }


    private Hashtable<String, Variable> symbolTable;
    private Lexer lexer;
    private CompilerError error;


}



// ###############################################################
package Lexer;

import java.util.*; import CompilerError;

public class Lexer {

    public Lexer( char []input, CompilerError error ) {
        this.input = input;
          // add an end-of-file label to make it easy to do the lexer
        input[input.length - 1] = '\0';
          // number of the current line
        lineNumber = 1;
        tokenPos = 0;
        this.error = error;
    }

      // contains the keywords
    static private Hashtable<String, Symbol> keywordsTable;

     // this code will be executed only once for each program execution
     static {
        keywordsTable = new Hashtable<String, Symbol>();
        keywordsTable.put( "var", Symbol.VAR );
        keywordsTable.put( "begin", Symbol.BEGIN );
```

```
    keywordsTable.put( "end", Symbol.END );
    keywordsTable.put( "if", Symbol.IF );
    keywordsTable.put( "then", Symbol.THEN );
    keywordsTable.put( "else", Symbol.ELSE );
    keywordsTable.put( "endif", Symbol.ENDIF );
    keywordsTable.put( "read", Symbol.READ );
    keywordsTable.put( "write", Symbol.WRITE );
    keywordsTable.put( "integer", Symbol.INTEGER );
    keywordsTable.put( "boolean", Symbol.BOOLEAN );
    keywordsTable.put( "char", Symbol.CHAR );
    keywordsTable.put( "true", Symbol.TRUE );
    keywordsTable.put( "false", Symbol.FALSE );
    keywordsTable.put( "and", Symbol.AND );
    keywordsTable.put( "or", Symbol.OR );
    keywordsTable.put( "not", Symbol.NOT );


 }



public void nextToken() {
    char ch;

    while (  (ch = input[tokenPos]) == ' ' || ch == '\r' ||
             ch == '\t' || ch == '\n')  {
        // count the number of lines
      if ( ch == '\n')
        lineNumber++;
      tokenPos++;
      }
    if ( ch == '\0')
      token = Symbol.EOF;
    else
      if ( input[tokenPos] == '/' && input[tokenPos + 1] == '/' ) {
            // comment found
           while ( input[tokenPos] != '\0'&& input[tokenPos] != '\n' )
             tokenPos++;
           nextToken();
           }
      else {
        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 identStr is in the list of keywords, it is a keyword !
    Symbol value = keywordsTable.get(stringValue);
    if ( value == null )
      token = Symbol.IDENT;
    else
      token = value;
    if ( Character.isDigit(input[tokenPos]) )
      error.signal("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;
    try {
       numberValue = Integer.valueOf(number.toString()).intValue();
    } catch ( NumberFormatException e ) {
       error.signal("Number out of limits");
    }
    if ( numberValue >= MaxValueInteger )
       error.signal("Number out of limits");

} else {
    tokenPos++;
    switch ( ch ) {
        case '+' :
          token = Symbol.PLUS;
          break;
        case '-' :
          token = Symbol.MINUS;
          break;
        case '*' :
          token = Symbol.MULT;
          break;
        case '/' :
          token = Symbol.DIV;
          break;
        case '%' :
          token = Symbol.REMAINDER;
```

```
  break;
case '<' :
  if ( input[tokenPos] == '=' ) {
    tokenPos++;
    token = Symbol.LE;
  }
  else if ( input[tokenPos] == '>' ) {
    tokenPos++;
    token = Symbol.NEQ;
  }
  else
    token = Symbol.LT;
  break;
case '>' :
  if ( input[tokenPos] == '=' ) {
    tokenPos++;
    token = Symbol.GE;
  }
  else
    token = Symbol.GT;
  break;
case '=' :
  if ( input[tokenPos] == '=' ) {
    tokenPos++;
    token = Symbol.EQ;
  }
  else
    token = Symbol.ASSIGN;
  break;
case '(' :
  token = Symbol.LEFTPAR;
  break;
case ')' :
  token = Symbol.RIGHTPAR;
  break;
case ',' :
  token = Symbol.COMMA;
  break;
case ';' :
  token = Symbol.SEMICOLON;
  break;
case ':' :
  token = Symbol.COLON;
  break;
case '\'' :
  token = Symbol.CHARACTER;
```

```
                  charValue = input[tokenPos];
                  tokenPos++;
                  if ( input[tokenPos] != '\'' )
                    error.signal("Illegal literal character" + input[tokenPos-1] );
                  tokenPos++;
                  break;
                default :
                  error.signal("Invalid Character: '" + ch + "'");
              }
          }
        }
    lastTokenPos = tokenPos - 1;
}

  // return the line number of the last token got with getToken()
public int getLineNumber() {
    return lineNumber;
}

public String getCurrentLine() {
    int i = lastTokenPos;
    if ( i == 0 )
      i = 1;
    else
      if ( i >= input.length )
        i = input.length;

    StringBuffer line = new StringBuffer();
      // go to the beginning of the line
    while ( i >= 1 && input[i] != '\n' )
      i--;
    if ( input[i] == '\n' )
      i++;
      // go to the end of the line putting it in variable line
    while ( input[i] != '\0' && input[i] != '\n' && input[i] != '\r' ) {
        line.append( input[i] );
        i++;
    }
    return line.toString();
}

public String getStringValue() {
   return stringValue;
}

public int getNumberValue() {
```

```
       return numberValue;
    }

    public char getCharValue() {
       return charValue;
    }
          // current token
    public Symbol token;
    private String stringValue;
    private int numberValue;
    private char charValue;

    private int  tokenPos;
      //  input[lastTokenPos] is the last character of the last token
    private int lastTokenPos;
      // program given as input - source code
    private char []input;

    // number of current line. Starts with 1
    private int lineNumber;

    private CompilerError error;
    private static final int MaxValueInteger = 32768;
}


// ###############################################################
package Lexer;

public enum Symbol {

      EOF("eof"),
      IDENT("Ident"),
      NUMBER("Number"),
      PLUS("+"),
      MINUS("-"),
      MULT("*"),
      DIV("/"),
      LT("<"),
      LE("<="),
      GT(">"),
      GE(">="),
      NEQ("!="),
      EQ("=="),
      ASSIGN("="),
      LEFTPAR("("),
```

```
      RIGHTPAR(")"),
      SEMICOLON(";"),
      VAR("var"),
      BEGIN("begin"),
      END("end"),
      IF("if"),
      THEN("then"),
      ELSE("else"),
      ENDIF("endif"),
      COMMA(","),
      READ("read"),
      WRITE("write"),
      COLON(":"),
      INTEGER("integer"),
      BOOLEAN("boolean"),
      CHAR("char"),
      CHARACTER("character"),
      TRUE("true"),
      FALSE("false"),
      OR   ("||"),
      AND  ("&&"),
      REMAINDER("%"),
      NOT("!");

      Symbol(String name) {
          this.name = name;
      }

      public String toString() {
          return name;
      }

      private String name;

}

// ###############################################################
package AST;

public class AssignmentStatement extends Statement {

    public AssignmentStatement( Variable v, Expr expr ) {
        this.v = v;
        this.expr = expr;
    }
```

```
    public void genC( PW pw ) {
        pw.print(v.getName() + " = " );
        expr.genC(pw);
        pw.out.println(";");
    }
    private Variable v;
    private Expr expr;
}

// ###############################################################
package AST;

public class BooleanExpr extends Expr {

    public BooleanExpr( boolean value ) {
        this.value = value;
    }

    public void genC( PW pw ) {
       pw.out.print( value ? "1" : "0" );
    }

    public Type getType() {
        return Type.booleanType;
    }

    public static BooleanExpr True  = new BooleanExpr(true);
    public static BooleanExpr False = new BooleanExpr(false);

    private boolean value;
}


// ###############################################################
package AST;

public class BooleanType extends Type {

   public BooleanType() { super("boolean"); }

   public String getCname() {
      return "int";
   }
}
```

```
// ##############################################################
package AST;

public class CharExpr extends Expr {

    public CharExpr( char value ) {
        this.value = value;
    }

    public void genC( PW pw ) {
        pw.out.print("'" + value + "'");
    }

    public char getValue() {
        return value;
    }

    public Type getType() {
        return Type.charType;
    }

    private char value;
}

// ##############################################################
package AST;

public class CharType extends Type {

    public CharType() {
        super("char");
    }

   public String getCname() {
      return "char";
   }

}

// ##############################################################
package AST;

import Lexer.*;

public class CompositeExpr extends Expr {
```

```
    public CompositeExpr( Expr pleft, Symbol poper, Expr pright ) {
        left = pleft;
        oper = poper;
        right = pright;
    }
    public void genC( PW pw ) {
        pw.out.print("(");
        left.genC(pw);
        pw.out.print(" " + oper.toString() + " ");
        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 ||
            oper == Symbol.AND || oper == Symbol.OR )
            return Type.booleanType;
       else
            return Type.integerType;
    }

    private Expr left, right;
    private Symbol oper;
}


// ##############################################################
package AST;

abstract public class Expr {
    abstract public void genC( PW pw );
      // new method: the type of the expression
    abstract public Type getType();
}

// ##############################################################
package AST;

public class IfStatement extends Statement {

    public IfStatement( Expr expr, StatementList thenPart, StatementList elsePart ) {
        this.expr = expr;
        this.thenPart = thenPart;
        this.elsePart = elsePart;
```

```
    }

    public void genC( PW pw ) {

        pw.print("if ( ");
        expr.genC(pw);
        pw.out.println(" ) { ");
        if ( thenPart != null ) {
          pw.add();
          thenPart.genC(pw);
          pw.sub();
          pw.println("}");
        }
        if ( elsePart != null ) {
          pw.println("else {");
          pw.add();
          elsePart.genC(pw);
          pw.sub();
          pw.println("}");
        }
    }

    private Expr expr;
    private StatementList thenPart, elsePart;
}

// ###############################################################
package AST;

public class IntegerType extends Type {

    public IntegerType() {
        super("integer");
    }

   public String getCname() {
      return "int";
   }

}

// ###############################################################
package AST;

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;
}


// ###############################################################
package AST;

import java.io.*;

public class PW {

   public void add() {
      currentIndent += step;
   }
   public void sub() {
      currentIndent -= step;
   }

   public void set( PrintWriter out ) {
      this.out = out;
      currentIndent = 0;
   }

   public void set( int indent ) {
      currentIndent = indent;
   }

   public void print( String s ) {
      out.print( space.substring(0, currentIndent) );
      out.print(s);
   }
```

```
   public void println( String s ) {
      out.print( space.substring(0, currentIndent) );
      out.println(s);
   }

   int currentIndent = 0;
   /* there is a Java and a Green mode.
      indent in Java mode:
      3 6 9 12 15 ...
      indent in Green mode:
      3 6 9 12 15 ...
   */
   static public final int green = 0, java = 1;
   int mode = green;
   public int step = 3;
   public PrintWriter out;


   static final private String space = "

}


// ###############################################################
package AST;

public class ParenthesisExpr extends Expr {

    public ParenthesisExpr( Expr expr ) {
        this.expr = expr;
    }

    public void genC( PW pw ) {
        pw.out.print("(");
        expr.genC(pw);
        pw.out.print(")");
    }

    public Type getType() {
        return expr.getType();
    }

    private Expr expr;
}
```

```
// ############################################################
package AST;

import java.util.*;

public class Program {

    public Program( ArrayList<Variable> arrayVariable,
                    StatementList statementList ) {
        this.arrayVariable = arrayVariable;
        this.statementList = statementList;
    }

    public void genC( PW pw ) {

        pw.out.println("#include <stdio.h>");
        pw.out.println();
        pw.println("void main() {");

        pw.add();
          // generate code for the declaration of variables
        for( Variable v : arrayVariable )
            pw.println(v.getType().getCname() +
                " " + v.getName() + ";" );


        pw.out.println("");
        statementList.genC(pw);
        pw.sub();
        pw.out.println("}");
    }

    private ArrayList<Variable> arrayVariable;
    private StatementList statementList;
}

// ############################################################
package AST;

public class ReadStatement extends Statement {
    public ReadStatement( Variable v ) {
        this.v = v;
    }

    public void genC( PW pw ) {
        if ( v.getType() == Type.charType )
```

```
          pw.print("{ char s[256]; gets(s); sscanf(s, \"%c\", &"  );
        else // should only be an integer
          pw.print("{ char s[256]; gets(s); sscanf(s, \"%d\", &"  );
        pw.out.println(  v.getName() + "); }" );
    }
    private Variable v;
}

// ###############################################################
package AST;

abstract public class Statement {
    abstract public void genC( PW pw );
}


// ###############################################################
package AST;

import java.util.*;

public class StatementList {

    public StatementList(ArrayList<Statement> v) {
        this.v = v;
    }

    public void genC( PW pw ) {

      for( Statement s : v )
          s.genC(pw);
    }

    private ArrayList<Statement> v;
}

// ###############################################################
package AST;

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 String getCname();

    private String name;
}


// ###############################################################
package AST;

import Lexer.Symbol;

public class UnaryExpr extends Expr {

    public UnaryExpr( Expr expr, Symbol op ) {
        this.expr = expr;
        this.op = op;
    }

    public void genC( PW pw ) {
        switch ( op ) {
            case PLUS :
              pw.out.print("+");
              break;
            case MINUS :
              pw.out.print("-");
              break;
            case NOT :
              pw.out.print("!");
              break;
        }
        expr.genC(pw);
    }

    public Type getType() {
        return expr.getType();
    }

    private Expr expr;
    private Symbol op;
```

```
}

// ##############################################################
package AST;

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 Type getType() {
        return type;
    }

    private String name;
    private Type type;
}

// ##############################################################
package AST;

public class VariableExpr extends Expr {

    public VariableExpr( Variable v ) {
        this.v = v;
    }

    public void genC( PW pw ) {
        pw.out.print( v.getName() );
    }

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

```
    private Variable v;
}

// ##############################################################
package AST;

public class WriteStatement extends Statement {

    public WriteStatement( Expr expr ) {
        this.expr = expr;
    }

    public void genC( PW pw ) {

        if ( expr.getType() == Type.booleanType ) {
          pw.print("printf(\"%s\\n\", ");
          expr.genC(pw);
          pw.out.print(" ? \"True\" : \"False\"");
        }
        else {
            if ( expr.getType() == Type.charType )
              pw.print("printf(\"%c\\n\", ");
            else
              pw.print("printf(\"%d\\n\", ");
            expr.genC(pw);
        }
        pw.out.println(" );");
    }


    private Expr expr;
}
```

E The Complete Code of Compiler 10

This Appendix gives the complete code of compiler 10.

```
import AST.*; import java.io.*;

public class Main {

    public static void main( String []args ) {

        File file;
        FileReader stream;
        int numChRead;
        Program program;

        if ( args.length != 1 &&  args.length != 2 )  {
            System.out.println("Usage:\n   Main input [output]");
            System.out.println("input is the file to be compiled");
            System.out.println("output is the file where the generated code will be stored");
        }
        else {
           file = new File(args[0]);
           if ( ! file.exists() || ! file.canRead() ) {
             System.out.println("Either the file " + args[0] + " does not exist or it cannot be
             return ;
           }
           try {
             stream = new FileReader(file);
            } catch ( FileNotFoundException e ) {
                System.out.println("Something wrong: file does not exist anymore");
                throw new RuntimeException();
            }
                // one more character for '\0' at the end that will be added by the
                // compiler
            char []input = new char[ (int ) file.length() + 1 ];

            try {
              numChRead = stream.read( input, 0, (int ) file.length() );
            } catch ( IOException e ) {
                System.out.println("Error reading file " + args[0]);
                return ;
            }

            if ( numChRead != file.length() ) {
                System.out.println("Read error");
                return ;
```

```
        }
        try {
          stream.close();
        } catch ( IOException e ) {
            System.out.println("Error in handling the file " + args[0]);
            return ;
        }


        Compiler compiler = new Compiler();

        String outputFileName;
        if ( args.length == 2 )
          outputFileName = args[1];
        else {
            outputFileName = args[0];
            int lastIndex;
            if ( (lastIndex = outputFileName.lastIndexOf('.')) == -1 )
              lastIndex = outputFileName.length();
            StringBuffer sb = new StringBuffer(outputFileName.substring(0, lastIndex));
            sb.append(".c");
            outputFileName = sb.toString();
        }

        FileOutputStream  outputStream;
        try {
           outputStream = new FileOutputStream(outputFileName);
        } catch ( IOException e ) {
            System.out.println("File " + args[1] + " was not found");
            return ;
        }
        PrintWriter printWriter = new PrintWriter(outputStream);
        program = null;
          // the generated code goes to a file and so are the errors
        program  = compiler.compile(input, new PrintWriter(System.out) );

        if ( program != null ) {
           PW pw = new PW();
           pw.set(printWriter);
           program.genC( pw );
           if ( printWriter.checkError() ) {
              System.out.println("There was an error in the output");
           }
        }
    }
}
```

```
}


// ##############################################################

import java.util.*;

public class SymbolTable {

    public SymbolTable() {
        globalTable = new Hashtable();
        localTable  = new Hashtable();
    }

    public Object putInGlobal( String key, Object value ) {
       return globalTable.put(key, value);
    }

    public Object putInLocal( String key, Object value ) {
       return localTable.put(key, value);
    }

    public Object getInLocal( Object key ) {
       return localTable.get(key);
    }

    public Object getInGlobal( Object key ) {
       return globalTable.get(key);
    }

    public Object get( String key ) {
        // returns the object corresponding to the key.
        Object result;
        if ( (result = localTable.get(key)) != null ) {
              // found local identifier
            return result;
        }
        else {
              // global identifier, if it is in globalTable
            return globalTable.get(key);
        }
    }

    public void removeLocalIdent() {
           // remove all local identifiers from the table
         localTable.clear();
```

```
    }


    private Hashtable globalTable, localTable;
}


// ############################################################

public class StatementException extends Exception { }


// ############################################################
import Lexer.*; import java.io.*;

public class CompilerError {

    public CompilerError( Lexer lexer, PrintWriter out ) {
          // output of an error is done in out
        this.lexer = lexer;
        this.out = out;
        thereWasAnError = false;
    }

    public void setLexer( Lexer lexer ) {
        this.lexer = lexer;
    }

    public boolean wasAnErrorSignalled() {
        return thereWasAnError;
    }

    public void show( String strMessage ) {
        show( strMessage, false );
    }

    public void show( String strMessage, boolean goPreviousToken ) {
        // is goPreviousToken is true, the error is signalled at the line of the
        // previous token, not the last one.
        if ( goPreviousToken ) {
          out.println("Error at line " + lexer.getLineNumberBeforeLastToken() + ": ");
          out.println( lexer.getLineBeforeLastToken() );
        }
        else {
          out.println("Error at line " + lexer.getLineNumber() + ": ");
          out.println(lexer.getCurrentLine());
```

```
        }

        out.println( strMessage );
        out.flush();
        if ( out.checkError() )
          System.out.println("Error in signaling an error");
        thereWasAnError = true;
    }

    public void signal( String strMessage ) {
        show( strMessage );
        out.flush();
        thereWasAnError = true;
        throw new RuntimeException();
    }

    private Lexer lexer;
    private PrintWriter out;
    private boolean thereWasAnError;
}


// ############################################################
/*
    comp10

    Main changes from the previous compiler:

      - user can supply only the file to be compiled. The output file is made
        to a ".c" file with the same name as the input;
      - identifiers should start with a letter but can be composed by letters and
        digits;
      - there is a while and a for statement;
      - some error messages do not cause the compiler to terminate the
        compilation. Then the compiler can signal more than one error;
      - previous compilers put parenthesis around any composite expression. Now they are
        put only in some situations in which they are probably necessary;
      - the program has now 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*factorial(n-1);
          endif;
          end
```

- there must be a procedure called main without parameters.

Grammar:

```
   Program ::= ProcFunc { ProcFunc }
   ProcFunc ::= Procedure | Function
   Procedure ::= "procedure" Ident '(' ParamList ')'
       [ LocalVarDec ] CompositeStatement
   Function ::= "function" Ident '(' ParamList ')' ':' Type
       [ LocalVarDec ] CompositeStatement
   ParamList ::= | ParamDec { ';' ParamDec }
   ParamDec ::= Ident { ',' Ident } ':' Type
   LocalVarDec ::= "var" VarDecList
   CompositeStatement ::= "begin" StatementList "end"
   StatementList ::= | Statement ";" StatementList
   Statement ::= AssignmentStatement | IfStatement | ReadStatement |
      WriteStatement | ProcedureCall | ForStatement | WhileStatement |
      CompositeStatement | ReturnStatement
   AssignmentStatement ::= Ident "="  OrExpr
   IfStatement ::= "if" OrExpr "then" StatementList [ "else" StatementList ] "endif"
   ReadStatement ::= "read" "(" Ident ")"
   WriteStatement ::= "write" "(" OrExpr ")"
   ProcedureCall ::= Ident '(' ExprList ')'
   ExprList ::=  | OrExpr { ',' OrExpr }
   ForStatement ::= "for" Ident '=' OrExpr "to" OrExpr "do" Statement
   WhileStatement ::= "while" OrExpr "do" Statement
   ReturnStatement ::= "return" OrExpr

   VarDecList ::= VarDecList2  { VarDecList2  }
   VarDecList2 ::= Ident { ',' Ident } ':' Type ';'
   Ident ::= Letter { Letter }
   Type ::= "integer" | "boolean" | "char"
   OrExpr ::= AndExpr [ "or" AndExpr ]
```

```
        AndExpr ::= RelExpr [ "and" RelExpr ]
        RelExpr ::= AddExpr [ RelOp AddExpr ]
        AddExpr ::= MultExpr { AddOp MultExpr }
        MultExpr ::= SimpleExpr { MultOp SimpleExpr }
        SimpleExpr ::= Number | "true" | "false" | Character
            | '(' OrExpr ')' | "not" SimpleExpr | AddOp SimpleExpr
            | Ident [ '(' ExprList ')' ]
        RelOp ::= '<' | '<=' | '>' | '>='| '==' | '<>'
        AddOp ::= '+'| '-'
        MultOp ::= '*' | '/' | '%'
        Number ::= ['+'|'-'] Digit { Digit }
        Digit ::= '0'| '1' | ... | '9'
        Letter ::= 'A' | 'B'| ... | 'Z'| 'a'| 'b' | ... | 'z'

   Character is a Letter enclosed between ' and ', like 'A', 'e' as in Java, C++, etc.
   Anything between [] is optional. Anything between { e } can be
   repeated zero or more times.
*/

import AuxComp.StatementException;
import AuxComp.CompilerError;
import AuxComp.SymbolTable;
import AST.*;
import java.util.*;
import Lexer.*;
import java.io.*;


public class Compiler {

      // compile must receive an input with an character less than
      // p_input.lenght
    public Program compile( char []input, PrintWriter outError ) {

        symbolTable = new SymbolTable();
        error = new CompilerError( lexer, new PrintWriter(outError) );
        lexer = new Lexer(input, error);
        error.setLexer(lexer);

        lexer.nextToken();

        Program p = null;
        try {
          p = program();
        } catch ( Exception e ) {
              // the below statement prints the stack of called methods.
```

```
          // of course, it should be removed if the compiler were
          // a production compiler.
         e.printStackTrace();
    }
    if ( error.wasAnErrorSignalled() )
      return null;
    else
      return p;
    }

private Program program() {
   // Program ::= ProcFunc { ProcFunc }

   ArrayList<Subroutine> procfuncList = new ArrayList<Subroutine>();

   while ( lexer.token == Symbol.PROCEDURE ||
           lexer.token == Symbol.FUNCTION )
       procfuncList.add( procFunc() );

   Program program = new Program( procfuncList );
   if ( lexer.token != Symbol.EOF )
     error.signal("EOF expected");
     // semantics analysis
     // there must be a procedure called main
   Subroutine mainProc;
   if ( (mainProc = (Subroutine ) symbolTable.getInGlobal("main")) == null )
      error.show("Source code must have a procedure called main");
   return program;
}

private Subroutine procFunc() {
   // Procedure ::= "procedure" Ident '(' ParamList ')'
   //     [ LocalVarDec ] CompositeStatement
   // Function ::= "function" Ident '(' ParamList ')' ':' Type
   //     [ LocalVarDec ] CompositeStatement

    boolean isFunction;


    if ( lexer.token == Symbol.PROCEDURE )
      isFunction = false;
    else if ( lexer.token == Symbol.FUNCTION )
      isFunction = true;
    else {
      // should never occur
      error.signal("Internal compiler error");
```

```
   return null;
 }

lexer.nextToken();
if ( lexer.token != Symbol.IDENT )
  error.signal("Identifier expected");
String name = (String ) lexer.getStringValue();
  // Symbol table now searches for an identifier in the scope order. First
  // the local variables and parameters and then the procedures and functions.
  // at this point, there should not be any local variables/parameters in the
  // symbol table.
Subroutine s = (Subroutine ) symbolTable.getInGlobal(name);
  // semantic analysis
  // identifier is in the symbol table
if ( s != null )
  error.show("Subroutine " + name + " has already been declared");
lexer.nextToken();

if ( isFunction ) {
    // currentFunction is used to store the function being compiled or null if it is a p
  s = currentFunction = new Function(name);
}
else {
  s = new Procedure(name);
  currentFunction = null;
}

  // insert s in the symbol table
symbolTable.putInGlobal( name, s );

if ( lexer.token != Symbol.LEFTPAR ) {
  error.show("( expected");
  lexer.skipBraces();
}
else
  lexer.nextToken();

  // semantic analysis
  // if the subroutine is "main", it must be a parameterless procedure
if ( name.compareTo("main") == 0 && ( lexer.token != Symbol.RIGHTPAR
     || isFunction ) )
     error.show("main must be a parameterless procedure");

s.setParamList( paramList() );
if ( lexer.token != Symbol.RIGHTPAR ) {
  error.show(") expected");
```

```
      lexer.skipBraces();
    }
    else
      lexer.nextToken();

    if ( isFunction ) {
      if ( lexer.token != Symbol.COLON ) {
        error.show(": expected");
        lexer.skipPunctuation();
      }
      else
        lexer.nextToken();
      ((Function ) s).setReturnType( type() );
    }

    if ( lexer.token == Symbol.VAR )
      s.setLocalVarList( localVarDec() );

    s.setCompositeStatement( compositeStatement() );

    symbolTable.removeLocalIdent();
    return s;
}


private LocalVarList localVarDec() {
    // LocalVarDec ::= "var" VarDecList
    // VarDecList ::= VarDecList2  { VarDecList2  }

    LocalVarList localVarList = new LocalVarList();
      // eat token "var"
    lexer.nextToken();

    varDecList2(localVarList);
    while ( lexer.token == Symbol.IDENT )
      varDecList2(localVarList);
    return localVarList;
}

private void varDecList2( LocalVarList localVarList ) {
    //  VarDecList2 ::= Ident { ',' Ident } ':' Type ';'

    ArrayList<Variable> lastVarList = new ArrayList<Variable>();

    while ( true ) {
      if ( lexer.token != Symbol.IDENT )
```

```
    error.signal("Identifier expected");
    // name of the identifier
  String name = (String ) lexer.getStringValue();
  lexer.nextToken();
    // semantic analysis
    // if the name is in the symbol table and the scope of the name is local,
    // the variable is been declared twice.
  if ( symbolTable.getInLocal(name) != null )
     error.show("Variable " + name + " has already been declared");

    // variable does not have a type yet
  Variable v = new Variable(name);
    // 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.putInLocal( name, v );
    // list of the last variables declared. They don't have types yet
  lastVarList.add(v);

  if ( lexer.token == Symbol.COMMA )
    lexer.nextToken();
  else
    break;
}

if ( lexer.token != Symbol.COLON ) {
  error.show(": expected");
  lexer.skipPunctuation();
}
else
  lexer.nextToken();
  // get the type
Type typeVar = type();

for ( Variable v : lastVarList ) {
  v.setType(typeVar);
    // add variable to the list of variable declarations
  localVarList.addElement( v );
}

if ( lexer.token != Symbol.SEMICOLON ) {
  error.show("; expected");
  lexer.skipPunctuation();
}
else
  lexer.nextToken();
```

```
}


private ParamList paramList() {
   // ParamList ::= | ParamDec { ';' ParamDec }

    ParamList paramList = null;
    if ( lexer.token == Symbol.IDENT ) {
        paramList = new ParamList();
        paramDec(paramList);
        while ( lexer.token == Symbol.SEMICOLON ) {
            lexer.nextToken();
            paramDec(paramList);
        }
    }
    return paramList;
}

private void paramDec( ParamList paramList ) {
   // ParamDec ::= Ident { ',' Ident } ':' Type

    ArrayList<Parameter> lastVarList = new ArrayList<Parameter>();

    while ( true ) {
      if ( lexer.token != Symbol.IDENT )
        error.signal("Identifier expected");
        // name of the identifier
      String name = (String ) lexer.getStringValue();
      lexer.nextToken();
        // semantic analysis
        // if the name is in the symbol table and the scope of the name is local,
        // the variable is been declared twice.
      if ( symbolTable.getInLocal(name) != null )
         error.show("Parameter " + name + " has already been declared");

        // variable does not have a type yet
      Parameter v = new Parameter(name);
        // 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.putInLocal( name, v );
        // list of the last variables declared. They don't have types yet
      lastVarList.add(v);

      if ( lexer.token == Symbol.COMMA )
        lexer.nextToken();
```

```
      else
        break;
    }

    if ( lexer.token != Symbol.COLON ) { //  :
      error.show(": expected");
      lexer.skipPunctuation();
    }
    else
      lexer.nextToken();
      // get the type
    Type typeVar = type();

    for ( Parameter v : lastVarList )  {
        // add type to the variable
      v.setType(typeVar);
        // add v to the list of parameter declarations
      paramList.addElement(v);

    }

}



private Type type() {
    Type result;

    switch ( lexer.token ) {
        case INTEGER :
          result = Type.integerType;
          break;
        case BOOLEAN :
          result = Type.booleanType;
          break;
        case CHAR :
          result = Type.charType;
          break;
        default :
          error.show("Type expected");
          result = Type.integerType;
    }
    lexer.nextToken();
    return result;
}
```

```
private CompositeStatement compositeStatement() {
      // CompositeStatement ::= "begin" StatementList "end"
      //  StatementList ::= | Statement ";" StatementList

    if ( lexer.token != Symbol.BEGIN )
        error.show("begin expected");
    else
      lexer.nextToken();
    StatementList sl = statementList();
    if ( lexer.token != Symbol.END )
      error.show("end expected");
    else
      lexer.nextToken();
    return new CompositeStatement(sl);
}



private StatementList statementList() {

    Symbol tk;
    Statement astatement;
    ArrayList<Statement> v = new ArrayList<Statement>();

      // statements always begin with an identifier, if, read or write, ...
    while ( (tk = lexer.token) != Symbol.END &&
            tk != Symbol.ELSE &&
            tk != Symbol.ENDIF ) {
        astatement = null;
        try {
            // statement() should return null in a serious error
          astatement = statement();
        } catch (  StatementException e ) {
            lexer.skipToNextStatement();
        }
        if ( astatement != null ) {
          v.add(astatement);
          if ( lexer.token != Symbol.SEMICOLON ) {
            error.show("; expected", true);
            lexer.skipPunctuation();
          }
          else
            lexer.nextToken();
```

```
        }
    }
    return new StatementList(v);
}

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

    */

    switch (lexer.token) {
        case 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 IF :
          return ifStatement();
        case READ :
          return readStatement();
        case WRITE :
          return writeStatement();
        case FOR :
          return forStatement();
        case WHILE :
          return whileStatement();
        case BEGIN :
          return compositeStatement();
        case RETURN :
          return returnStatement();
        default :
          error.show("Statement expected");
          throw new StatementException();

    }
}

private AssignmentStatement assignmentStatement() {

    Variable v = checkVariable();
```

```
    String name = v.getName();

    if ( lexer.token != Symbol.ASSIGN ) {
      error.show("= expected");
      lexer.skipToNextStatement();
      return null;
    }
    else
      lexer.nextToken();
    Expr right = orExpr();
      // semantic analysis
      // check if expression has the same type as variable
    if ( ! checkAssignment( v.getType(), right.getType() ) )
      error.show("Type error in assignment");

    return new AssignmentStatement( v, right );
}

private boolean checkAssignment( Type varType, Type exprType ) {
    if ( varType == Type.undefinedType || exprType == Type.undefinedType )
      return true;
    else
      return varType == exprType;
}

private IfStatement ifStatement() {

    lexer.nextToken();
    Expr e = orExpr();
      // semantic analysis
      // check if expression has type boolean
    if ( e.getType() != Type.booleanType )
      error.show("Boolean type expected in if expression");

    if ( lexer.token != Symbol.THEN )
      error.show("then expected");
    else
      lexer.nextToken();
    StatementList thenPart = statementList();
    StatementList elsePart = null;
    if ( lexer.token == Symbol.ELSE ) {
        lexer.nextToken();
        elsePart = statementList();
    }
    if ( lexer.token != Symbol.ENDIF )
      error.signal("endif expected");
```

```
    lexer.nextToken();
    return new IfStatement( e, thenPart, elsePart );
}



private ReadStatement readStatement() {
    lexer.nextToken();
    if ( lexer.token != Symbol.LEFTPAR ) {
      error.show("( expected");
      lexer.skipBraces();
    }
    else
      lexer.nextToken();
    if ( lexer.token != Symbol.IDENT )
      error.signal("Identifier expected");
      // semantic analysis
      // check if the variable was declared
    String name = (String ) lexer.getStringValue();
    Variable v = (Variable ) symbolTable.getInLocal(name);
    if ( v == null ) {
      error.show("Variable " + name + " was not declared");
      symbolTable.putInLocal( name, new Variable(name, Type.undefinedType) );
    }
      // semantic analysis
      // check if variable has type char or integer or undefined
    if ( v.getType() != Type.charType && v.getType() != Type.integerType &&
         v.getType() != Type.undefinedType )
      error.show("Variable should have type char or integer");

    lexer.nextToken();
    if ( lexer.token != Symbol.RIGHTPAR ) {
      error.show(") expected");
      lexer.skipBraces();
      lexer.skipToNextStatement();
      return null;
    }
    else {
      lexer.nextToken();
      return new ReadStatement( v );
    }
}


private WriteStatement writeStatement() {
```

```
    lexer.nextToken();
    if ( lexer.token != Symbol.LEFTPAR ) {
      error.show("( expected");
      lexer.skipBraces();
    }
    else
      lexer.nextToken();
      // expression may be of any type
    Expr e = orExpr();
    if ( lexer.token != Symbol.RIGHTPAR ) {
      error.show(") expected");
      lexer.skipBraces();
      lexer.skipToNextStatement();
      return null;
    }
    else {
      lexer.nextToken();
      return new WriteStatement( e );
    }
}



private ProcedureCall procedureCall() {
      // we already know the identifier is a procedure. So we need not to check it
      // again.
    ExprList anExprList = null;

    String name = (String ) lexer.getStringValue();
    lexer.nextToken();
    Procedure p = (Procedure ) symbolTable.getInGlobal(name);
    if ( lexer.token != Symbol.LEFTPAR ) {
      error.show("( expected");
      lexer.skipBraces();
    }
    else
      lexer.nextToken();

    if ( lexer.token != Symbol.RIGHTPAR ) {
        // The parameter list is used to check if the arguments to the
        // procedure have the correct types
      anExprList = exprList( p.getParamList() );
      if ( lexer.token != Symbol.RIGHTPAR )
        error.show("Error in expression");
      else
        lexer.nextToken();
```

```
    }
    else {
        // semantic analysis
        // does the procedure has no parameter ?
      if ( p.getParamList() != null && p.getParamList().getSize() != 0 )
        error.signal("Parameter expected");
      lexer.nextToken();
    }

    return new ProcedureCall(p,anExprList);
}



ExprList exprList( ParamList paramList )
{
   ExprList anExprList;
   boolean firstErrorMessage = true;

   if ( lexer.token == Symbol.RIGHTPAR )
      return null;
   else {
      Parameter parameter;
      int sizeParamList = paramList.getSize();
      Iterator e = paramList.getParamList().iterator();
      anExprList = new ExprList();
      while ( true ) {
        parameter = (Parameter ) e.next();
          // semantic analysis
          // does the procedure has one more parameter ?
        if ( sizeParamList < 1 && firstErrorMessage ) {
          error.show("Wrong number of parameters in call");
          firstErrorMessage = false;
        }
        sizeParamList--;
        Expr anExpr = orExpr();
        if ( ! checkAssignment(parameter.getType(), anExpr.getType()) )
          error.show("Type error in parameter passing");
        anExprList.addElement( anExpr );
        if ( lexer.token == Symbol.COMMA )
          lexer.nextToken();
        else
          break;
      }
        // semantic analysis
        // the procedure may need more parameters
```

```
      if ( sizeParamList > 0 && firstErrorMessage )
        error.show("Wrong number of parameters");
      return anExprList;
   }
}

private Statement forStatement() throws StatementException {

    Variable v = null;

    lexer.nextToken();
    if ( lexer.token != Symbol.IDENT )
        error.signal("Variable expected");
    v = checkVariable();
      // variable can be of any type
    if ( lexer.token != Symbol.ASSIGN )
      error.show("= expected");
    else
      lexer.nextToken();
    Expr exprLower = orExpr();
    if ( lexer.token != Symbol.TO )
      error.show("to expected");
    else
      lexer.nextToken();
    Expr exprUpper = orExpr();
      // semantic analysis
    if ( ! checkForExpr(exprLower.getType(), exprUpper.getType()) )
      error.show("both for expressions should have the same type");
    if ( lexer.token != Symbol.DO )
      error.show("do expected");
    else
      lexer.nextToken();
    return new ForStatement(v, exprLower, exprUpper, statement() );
}


private Statement whileStatement() throws StatementException {

    lexer.nextToken();
    Expr expr = orExpr();
    if ( !  checkWhileExpr(expr.getType()) )
      error.show("Boolean expression expected");
    if ( lexer.token != Symbol.DO )
      error.show("do expected");
    else
      lexer.nextToken();
```

```
    return new WhileStatement(expr, statement() );
}


private ReturnStatement returnStatement() {

    lexer.nextToken();
    Expr e = orExpr();
      // semantic analysis
      // Are we inside a function ?
    if ( currentFunction == null )
      error.show("return statement inside a procedure");
    else if ( ! checkAssignment( currentFunction.getReturnType(),
                            e.getType() ) )
        error.show("Return type does not match function type");
    return new ReturnStatement(e);
}


private boolean checkWhileExpr( Type exprType ) {
    if ( exprType == Type.undefinedType || exprType == Type.booleanType )
      return true;
    else
      return false;
}

private boolean checkForExpr( Type lowerExprType, Type upperExprType ) {
    if ( lowerExprType == Type.undefinedType || upperExprType == Type.integerType )
      return true;
    else
      return lowerExprType == upperExprType;
}


private Variable checkVariable() {
    // tests if the current identifier is a declared variable. If not,
    // declares it with the type Type.undefinedType.
    // assume lexer.token == Symbol.IDENT

    Variable v = null;

    String name = (String ) lexer.getStringValue();
    try {
      v = (Variable ) symbolTable.getInLocal( name );
    } catch ( Exception e ) {
    }
```

```
            // semantic analysis
            // was the variable declared ?
    if ( v == null ) {
        error.show("Variable " + name + " was not declared");
        v = new Variable(name, Type.undefinedType);
        symbolTable.putInLocal( name, v );
    }
    lexer.nextToken();
    return v;
}


private Expr orExpr() {
  /*
    OrExpr ::= AndExpr [ "or" AndExpr ]
  */

  Expr left, right;
  left = andExpr();
  if ( lexer.token == Symbol.OR ) {
    lexer.nextToken();
    right = andExpr();
      // semantic analysis
    if ( ! checkBooleanExpr( left.getType(), right.getType() ) )
       error.show("Expression of boolean type expected");
    left = new CompositeExpr(left, Symbol.OR, right);
  }
  return left;
}

private boolean checkBooleanExpr( Type left, Type right ) {

    if ( left == Type.undefinedType || right == Type.undefinedType )
      return true;
    else
      return left == Type.booleanType && right == Type.booleanType;
}

private Expr andExpr() {
  /*
    AndExpr ::= RelExpr [ "and" RelExpr ]
  */
  Expr left, right;
  left = relExpr();
  if ( lexer.token == Symbol.AND ) {
    lexer.nextToken();
```

```
    right = relExpr();
      // semantic analysis
    if ( ! checkBooleanExpr( left.getType(), right.getType() )  )
       error.show("Expression of boolean type expected");
    left = new CompositeExpr( left, Symbol.AND, right );
  }
  return left;
}

private Expr relExpr() {
  /*
    RelExpr ::= AddExpr [ RelOp AddExpr ]
  */
  Expr left, right;
  left = addExpr();
  Symbol op = lexer.token;
  if ( op == Symbol.EQ || op == Symbol.NEQ || op == Symbol.LE || op == Symbol.LT ||
       op == Symbol.GE || op == Symbol.GT  ) {
     lexer.nextToken();
     right = addExpr();
       // semantic analysis
     if ( ! checkRelExpr(left.getType(), right.getType() ) )
       error.show("Type error in expression");
     left = new CompositeExpr( left, op, right );
   }
   return left;
}

private boolean checkRelExpr( Type left, Type right ) {

    if ( left == Type.undefinedType || right == Type.undefinedType )
      return true;
    else
      return left == right;
}


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

  */
  Symbol op;
  Expr left, right;
  left = multExpr();
  while ( (op = lexer.token) == Symbol.PLUS ||
```

```
          op == Symbol.MINUS ) {
    lexer.nextToken();
    right = multExpr();
      // semantic analysis
    if ( ! checkMathExpr( left.getType(), right.getType() ) )
       error.show("Expression of type integer expected");
    left = new CompositeExpr( left, op, right );
  }
  return left;
}

private boolean checkMathExpr( Type left, Type right ) {
    boolean orLeft  = left  == Type.integerType ||
                      left  == Type.undefinedType;
    boolean orRight = right == Type.integerType ||
                      right == Type.undefinedType;
    return orLeft && orRight;
}

private Expr multExpr() {
 /*
    MultExpr ::= SimpleExpr { MultOp SimpleExpr }
 */
   Expr left, right;
   left = simpleExpr();
   Symbol op;
   while ( (op = lexer.token) == Symbol.MULT ||
           op == Symbol.DIV || op == Symbol.REMAINDER ) {
      lexer.nextToken();
      right = simpleExpr();
        // semantic analysis
      if ( ! checkMathExpr(left.getType(), right.getType())  )
        error.show("Expression of type integer expected");
      left = new CompositeExpr( left, op, right );
   }
   return left;
}

private Expr simpleExpr() {
  /*
    SimpleExpr ::= Number |  "true" | "false" | Character
       | '(' Expr ')' | "not" SimpleExpr | Variable
  */

  Expr e;
```

```
  // note we test the lexer.getToken() to decide which production to use
switch ( lexer.token ) {
   case NUMBER :
      return number();
   case TRUE :
     lexer.nextToken();
     return BooleanExpr.True;
   case FALSE :
     lexer.nextToken();
     return BooleanExpr.False;
   case CHARACTER :
       // get the token with getToken.
       // then get the value of it, with has the type Object
       // convert the object to type Character using a cast
       // call method charValue to get the character inside the object
     char ch = lexer.getCharValue();
     lexer.nextToken();
     return new CharExpr(ch);
   case LEFTPAR :
     lexer.nextToken();
     e = orExpr();
     if ( lexer.token != Symbol.RIGHTPAR ) {
       error.show(") expected");
       lexer.skipBraces();
     }
     else
       lexer.nextToken();
     return new ParenthesisExpr(e);
   case NOT :
     lexer.nextToken();
     e = orExpr();
       // semantic analysis
     if ( e.getType() != Type.booleanType )
       error.show("Expression of type boolean expected");
     return new UnaryExpr( e, Symbol.NOT );
   case PLUS :
     lexer.nextToken();
     e = orExpr();
       // semantic analysis
     if ( e.getType() != Type.integerType )
       error.show("Expression of type integer expected");
     return new UnaryExpr( e, Symbol.PLUS );
   case MINUS :
     lexer.nextToken();
     e = orExpr();
       // semantic analysis
```

```
  if ( e.getType() != Type.integerType )
    error.show("Expression of type integer expected");
  return new UnaryExpr( e, Symbol.MINUS );
default :
    // an identifier
  if ( lexer.token != Symbol.IDENT ) {
    error.show("Identifier expected");
    lexer.nextToken();
    return new VariableExpr(new Variable("nameless", Type.undefinedType ));
  }
  else {
       // this part needs to be improved. If the compiler finds
       // a call to a function that was not declared, it will sign an
       // error  "Identifier was not declared" and signal errors because of the
       // parentheses following the function name. This can be corrected.
    String name = (String ) lexer.getStringValue();
      // is it a function ?
    Object objIdent = symbolTable.get(name);
    if ( objIdent == null ) {
      error.show("Identifier was not declared");
      lexer.nextToken();
      if ( lexer.token != Symbol.LEFTPAR ) {
         Variable newVariable = new Variable(name, Type.undefinedType);
         symbolTable.putInLocal( name, newVariable );
         return new VariableExpr(newVariable);
      }
      else {
         Function falseFunction = new Function(name);
         falseFunction.setReturnType(Type.undefinedType);
         falseFunction.setCompositeStatement(null);
         symbolTable.putInGlobal(name, falseFunction);
         objIdent = falseFunction;
      }
    }

    if ( objIdent instanceof Subroutine ) {
       if ( objIdent instanceof Function )
          return functionCall();
       else {
         error.show("Attempt to call a procedure in an expression");
         procedureCall();
         return new VariableExpr(new Variable("nameless", Type.undefinedType));
       }
    }
    else {
       // it is a variable
```

```
            Variable v = (Variable ) objIdent;
            lexer.nextToken();
            return new VariableExpr(v);
          }
        }
  }

}


private FunctionCall functionCall() {
      // we already know the identifier is a function. So we
      // need not to check it again.
    ExprList anExprList = null;

    String name = (String ) lexer.getStringValue();
    lexer.nextToken();
    Function p = (Function ) symbolTable.getInGlobal(name);
    if ( lexer.token != Symbol.LEFTPAR ) {
      error.show("( expected");
      lexer.skipBraces();
    }
    else
      lexer.nextToken();

    if ( lexer.token != Symbol.RIGHTPAR ) {
        // The parameter list is used to check if the arguments to the
        // procedure have the correct types
      anExprList = exprList( p.getParamList() );
      if ( lexer.token != Symbol.RIGHTPAR )
        error.show("Error in expression");
      else
        lexer.nextToken();
    }
    else {
        // semantic analysis
        // does the procedure has no parameter ?
      if ( p.getParamList() != null && p.getParamList().getSize() != 0 )
        error.show("Parameter expected");
      lexer.nextToken();
    }

    return new FunctionCall(p,anExprList);
}
```

```
private NumberExpr number() {

    NumberExpr e = null;

      // the number value is stored in lexer.getToken().value as an object of Integer.
      // Method intValue returns that value as an value of type int.
    int value = lexer.getNumberValue();
    lexer.nextToken();
    return new NumberExpr( value );
}


private boolean checkTypes( Type left, Symbol op, Type right ) {
    // return true if  left and right can be the types of a composite
    // expression with operator op

   if ( op == Symbol.EQ || op == Symbol.NEQ || op == Symbol.LE || op == Symbol.LT ||
        op == Symbol.GE || op == Symbol.GT )
      return left == right;
   else if ( op == Symbol.PLUS || op == Symbol.MINUS || op == Symbol.DIV ||
             op == Symbol.MULT || op == Symbol.REMAINDER ) {
      if ( left != right || left != Type.integerType )
         return false;
      else
         return true;
   }
   else if ( op == Symbol.AND || op == Symbol.OR ) {
      if ( left != Type.booleanType ||
           right != Type.booleanType )
         return false;
      else
         return true;
   }
   else {
      error.signal("Compiler internal error: unknown operator");
      return true;
   }
}




private SymbolTable symbolTable;
private Lexer lexer;
```

```
    private CompilerError error;

      // keeps a pointer to the current function being compiled
    private Function currentFunction;
}

// ############################################################
package Lexer;

public enum Symbol {
      EOF("eof"),
      IDENT("Ident"),
      NUMBER("Number"),
      PLUS("+"),
      MINUS("-"),
      MULT("*"),
      DIV("/"),
      LT("<"),
      LE("<="),
      GT(">"),
      GE(">="),
      NEQ("!="),
      EQ("=="),
      ASSIGN("="),
      LEFTPAR("("),
      RIGHTPAR(")"),
      SEMICOLON(";"),
      VAR("var"),
      BEGIN("begin"),
      END("end"),
      IF("if"),
      THEN("then"),
      ELSE("else"),
      ENDIF("endif"),
      COMMA(","),
      READ("read"),
      WRITE("write"),
      COLON(":"),
      INTEGER("integer"),
      BOOLEAN("boolean"),
      CHAR("char"),
      CHARACTER("character"),
      TRUE("true"),
      FALSE("false"),
      OR   ("||"),
      AND  ("&&"),
```

```
      REMAINDER("%"),
      NOT("!"),
      PROCEDURE("procedure"),
      FUNCTION("function"),
        // the following symbols are used only at error treatment
      CURLYLEFTBRACE("{"),
      CURLYRIGHTBRACE("}"),
      LEFTSQBRACKET("["),
      RIGHTSQBRACKET("]"),
        // other symbols
      FOR("for"),
      WHILE("while"),
      TO("to"),
      DO("do"),
      RETURN("return");


      Symbol(String name) {
          this.name = name;
      }

      public String toString() {
          return name;
      }

      private String name;

}

// ###############################################################
 package Lexer;

import java.util.*;
import AuxComp.*;

public class Lexer {

    public Lexer( char []input, CompilerError error ) {
        this.input = input;
          // add an end-of-file label to make it easy to do the lexer
        input[input.length - 1] = '\0';
          // number of the current line
        lineNumber = 1;
        tokenPos = 0;
        lastTokenPos = 0;
        beforeLastTokenPos = 0;
```

```
    this.error = error;
}


public void skipBraces() {
      // skip any of the symbols [ ] { } ( )
    if ( token == Symbol.CURLYLEFTBRACE || token == Symbol.CURLYRIGHTBRACE ||
         token == Symbol.LEFTSQBRACKET  || token == Symbol.RIGHTSQBRACKET )
         nextToken();
    if ( token == Symbol.EOF )
      error.signal("Unexpected EOF");
}

public void skipPunctuation() {
      // skip any punctuation symbols
    while ( token != Symbol.EOF &&
            ( token == Symbol.COLON ||
              token == Symbol.COMMA ||
              token == Symbol.SEMICOLON) )
       nextToken();
    if ( token == Symbol.EOF )
      error.signal("Unexpected EOF");
}

public void skipTo( Symbol []arraySymbol )  {
    // skip till one of the characters of arraySymbol appears in the input
    while ( token != Symbol.EOF ) {
        int i = 0;
        while ( i < arraySymbol.length )
            if ( token == arraySymbol[i] )
              return;
            else
              i++;
        nextToken();
    }
    if ( token == Symbol.EOF )
      error.signal("Unexpected EOF");
}

public void skipToNextStatement() {

    while ( token != Symbol.EOF &&
            token != Symbol.ELSE  && token != Symbol.ENDIF &&
            token != Symbol.END && token != Symbol.SEMICOLON )
       nextToken();
    if ( token == Symbol.SEMICOLON )
```

```
      nextToken();
}


  // contains the keywords
static private Hashtable<String, Symbol> keywordsTable;

 // this code will be executed only once for each program execution
 static {
    keywordsTable = new Hashtable<String, Symbol>();
    keywordsTable.put( "var", Symbol.VAR );
    keywordsTable.put( "begin", Symbol.BEGIN );
    keywordsTable.put( "end", Symbol.END );
    keywordsTable.put( "if", Symbol.IF );
    keywordsTable.put( "then", Symbol.THEN );
    keywordsTable.put( "else", Symbol.ELSE );
    keywordsTable.put( "endif", Symbol.ENDIF );
    keywordsTable.put( "read", Symbol.READ );
    keywordsTable.put( "write", Symbol.WRITE );
    keywordsTable.put( "integer", Symbol.INTEGER );
    keywordsTable.put( "boolean", Symbol.BOOLEAN );
    keywordsTable.put( "char", Symbol.CHAR );
    keywordsTable.put( "true", Symbol.TRUE );
    keywordsTable.put( "false", Symbol.FALSE );
    keywordsTable.put( "and", Symbol.AND );
    keywordsTable.put( "or", Symbol.OR );
    keywordsTable.put( "not", Symbol.NOT );
    keywordsTable.put( "procedure", Symbol.PROCEDURE );
    keywordsTable.put( "function", Symbol.FUNCTION );
    keywordsTable.put( "for", Symbol.FOR );
    keywordsTable.put( "while", Symbol.WHILE );
    keywordsTable.put( "to", Symbol.TO );
    keywordsTable.put( "do", Symbol.DO );
    keywordsTable.put( "return", Symbol.RETURN );


 }




public void nextToken() {
    char ch;

    while (  (ch = input[tokenPos]) == ' ' || ch == '\r' ||
             ch == '\t' || ch == '\n')  {
```

```
    // count the number of lines
  if ( ch == '\n')
    lineNumber++;
  tokenPos++;
  }
if ( ch == '\0')
  token = Symbol.EOF;
else
  if ( input[tokenPos] == '/' && input[tokenPos + 1] == '/' ) {
        // comment found
       while ( input[tokenPos] != '\0'&& input[tokenPos] != '\n' )
         tokenPos++;
       nextToken();
       }
  else {
    if ( Character.isLetter( ch ) ) {
        // get an identifier or keyword
        StringBuffer ident = new StringBuffer();
        while ( Character.isLetter( input[tokenPos] ) ||
                Character.isDigit( input[tokenPos] ) ) {
            ident.append(input[tokenPos]);
            tokenPos++;
        }
        stringValue = ident.toString();
          // if identStr is in the list of keywords, it is a keyword !
        Symbol value = keywordsTable.get(stringValue);
        if ( value == null )
          token = Symbol.IDENT;
        else
          token = value;
    }
    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;
        try {
           numberValue = Integer.valueOf(number.toString()).intValue();
        } catch ( NumberFormatException e ) {
           error.signal("Number out of limits");
        }
        if ( numberValue >= MaxValueInteger )
           error.signal("Number out of limits");
```

```
}
else {
    tokenPos++;
    switch ( ch ) {
        case '+' :
          token = Symbol.PLUS;
          break;
        case '-' :
          token = Symbol.MINUS;
          break;
        case '*' :
          token = Symbol.MULT;
          break;
        case '/' :
          token = Symbol.DIV;
          break;
        case '%' :
          token = Symbol.REMAINDER;
          break;
        case '<' :
          if ( input[tokenPos] == '=' ) {
            tokenPos++;
            token = Symbol.LE;
          }
          else if ( input[tokenPos] == '>' ) {
            tokenPos++;
            token = Symbol.NEQ;
          }
          else
            token = Symbol.LT;
          break;
        case '>' :
          if ( input[tokenPos] == '=' ) {
            tokenPos++;
            token = Symbol.GE;
          }
          else
            token = Symbol.GT;
          break;
        case '=' :
          if ( input[tokenPos] == '=' ) {
            tokenPos++;
            token = Symbol.EQ;
          }
          else
            token = Symbol.ASSIGN;
```

```
                break;
              case '(' :
                token = Symbol.LEFTPAR;
                break;
              case ')' :
                token = Symbol.RIGHTPAR;
                break;
              case ',' :
                token = Symbol.COMMA;
                break;
              case ';' :
                token = Symbol.SEMICOLON;
                break;
              case ':' :
                token = Symbol.COLON;
                break;
              case '\'' :
                token = Symbol.CHARACTER;
                charValue = input[tokenPos];
                tokenPos++;
                if ( input[tokenPos] != '\'' )
                  error.signal("Illegal literal character" + input[tokenPos-1] );
                tokenPos++;
                break;
                // the next four symbols are not used by the language
                // but are returned to help the error treatment
              case '{' :
                token = Symbol.CURLYLEFTBRACE;
                break;
              case '}' :
                token = Symbol.CURLYRIGHTBRACE;
                break;
              case '[' :
                token = Symbol.LEFTSQBRACKET;
                break;
              case ']' :
                token = Symbol.RIGHTSQBRACKET;
                break;
              default :
                error.signal("Invalid Character: '" + ch + "'");
            }
        }
      }
    beforeLastTokenPos = lastTokenPos;
    lastTokenPos = tokenPos - 1;
}
```

```
  // return the line number of the last token got with getToken()
public int getLineNumber() {
    return lineNumber;
}

public int getLineNumberBeforeLastToken() {
    return getLineNumber( beforeLastTokenPos );
}

private int getLineNumber( int index ) {
    // return the line number in which the character input[index] is
    int i, n, size;
    n = 1;
    i = 0;
    size = input.length;
    while ( i < size && i < index ) {
      if ( input[i] == '\n' )
        n++;
      i++;
    }
    return n;
}


public String getCurrentLine() {
    return getLine(lastTokenPos);
}

public String getLineBeforeLastToken() {
    return getLine(beforeLastTokenPos);
}

private String getLine( int index ) {
    // get the line that contains input[index]. Assume input[index] is at a token, not
    // a white space or newline

    int i = index;
    if ( i == 0 )
      i = 1;
    else
      if ( i >= input.length )
        i = input.length;

    StringBuffer line = new StringBuffer();
      // go to the beginning of the line
```

```
        while ( i >= 1 && input[i] != '\n' )
          i--;
        if ( input[i] == '\n' )
          i++;
          // go to the end of the line putting it in variable line
        while ( input[i] != '\0' && input[i] != '\n' && input[i] != '\r' ) {
            line.append( input[i] );
            i++;
        }
        return line.toString();
    }

    public String getStringValue() {
       return stringValue;
    }

    public int getNumberValue() {
       return numberValue;
    }

    public char getCharValue() {
       return charValue;
    }
          // current token
    public Symbol token;
    private String stringValue;
    private int numberValue;
    private char charValue;

    private int tokenPos;
      //  input[lastTokenPos] is the last character of the last token found
    private int lastTokenPos;
      //  input[beforeLastTokenPos] is the last character of the token before the last
      // token found
    private int beforeLastTokenPos;
      // program given as input - source code
    private char []input;

    // number of current line. Starts with 1
    private int lineNumber;

    private CompilerError error;
    private static final int MaxValueInteger = 32768;
}
```

```
// ############################################################
package AST;

import java.io.*;

public class AssignmentStatement extends Statement {

    public AssignmentStatement( Variable v, Expr expr ) {
        this.v = v;
        this.expr = expr;
    }

    public void genC( PW pw ) {
        pw.print(v.getName() + " = " );
        expr.genC(pw);
        pw.out.println(";");
    }
    private Variable v;
    private Expr expr;
}

// ############################################################
package AST;

import java.io.*;

public class BooleanExpr extends Expr {

    public BooleanExpr( boolean value ) {
        this.value = value;
    }

    public void genC( PW pw ) {
       pw.out.print( value ? "1" : "0" );
    }

    public Type getType() {
        return Type.booleanType;
    }

    public static BooleanExpr True  = new BooleanExpr(true);
    public static BooleanExpr False = new BooleanExpr(false);

    private boolean value;
}
```

```
// ##############################################################
package AST;

import java.io.*;

public class BooleanType extends Type {

   public BooleanType() { super("boolean"); }

   public String getCname() {
      return "int";
   }

}


// ##############################################################
package AST;

import java.io.*;

public class CharExpr extends Expr {
    public CharExpr( char value ) {
        this.value = value;
    }

    public void genC( PW pw ) {
        pw.out.print("'" + value + "'");
    }

    public char getValue() {
        return value;
    }

    public Type getType() {
        return Type.charType;
    }

    private char value;
}

// ##############################################################
package AST;

import java.io.*;
```

```
public class CharType extends Type {

    public CharType() {
        super("char");
    }

   public String getCname() {
      return "char";
   }

}

// ##############################################################
package AST;

import Lexer.*; import java.io.*;

public class CompositeExpr extends Expr {

    public CompositeExpr( Expr pleft, Symbol poper, Expr pright ) {
        left = pleft;
        oper = poper;
        right = pright;
    }

    public void genC( PW pw ) {
        left.genC(pw);
        pw.out.print(" " + oper.toString() + " ");
        right.genC(pw);
    }

    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 ||
            oper == Symbol.AND || oper == Symbol.OR )
            return Type.booleanType;
       else
            return Type.integerType;
    }

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

```
// ##############################################################
package AST;

import java.util.*; import java.io.*;

public class CompositeStatement extends Statement {

    public CompositeStatement( StatementList statementList ) {
        this.statementList = statementList;
    }

    public void genC( PW pw ) {
        pw.println("{");
        if ( statementList != null ) {
           pw.add();
           statementList.genC(pw);
           pw.sub();
        }
        pw.println("}");
    }

    public StatementList getStatementList() { return statementList; }

    private StatementList statementList;
}


// ##############################################################
package AST;

import java.io.PrintWriter;

abstract public class Expr {
    abstract public void genC( PW pw );
      // new method: the type of the expression
    abstract public Type getType();
}

// ##############################################################
package AST;

import java.io.*; import java.util.*;

public class ExprList {

    public ExprList() {
```

```
        v = new ArrayList<Expr>();
    }

    public void addElement( Expr expr ) {
        v.add(expr);
    }

    public void genC( PW pw ) {

        int size = v.size();
        Iterator e = v.iterator();
        while ( e.hasNext() ) {
            ((Expr ) e.next()).genC(pw);
            if ( --size > 0 )
              pw.out.print(", ");
        }



    }

    private ArrayList<Expr> v;

}

// ##############################################################
package AST;

import java.io.*;

public class ForStatement extends Statement {

    public ForStatement( Variable forVariable,
                         Expr exprStart,
                         Expr exprEnd,
                         Statement statement ) {
        this.forVariable = forVariable;
        this.exprStart = exprStart;
        this.exprEnd = exprEnd;
        this.statement= statement;
    }

    public void genC( PW pw ) {
        pw.print("for ( " + forVariable.getName() + " = ");
        exprStart.genC(pw);
        pw.out.print("; " + forVariable.getName() + " <= ");
```

```
        exprEnd.genC(pw);
        pw.out.println("; " + forVariable.getName() + "++ )");
        if ( statement != null ) {
           if ( statement instanceof CompositeStatement )
              statement.genC(pw);
           else {
              pw.add();
              statement.genC(pw);
              pw.sub();
           }
        }
    }

    private Variable forVariable;
    private Expr exprStart, exprEnd;
    private Statement statement;
}

// ############################################################
package AST;

import java.io.*;

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 ) {

        pw.out.print(returnType.getCname() + " " + 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;
}


// ############################################################
package AST;

import java.io.*;

public class FunctionCall extends Expr {

    public FunctionCall( Function function, ExprList exprList ) {
        this.function = function;
        this.exprList = exprList;
    }

    public void genC( PW pw, boolean putParenthesis ) {
        pw.out.print( function.getName() + "(" );
        if ( exprList != null )
          exprList.genC(pw);
        pw.out.print( ")");
    }

    public Type getType() {
        return function.getReturnType();
    }

    Function function;
    ExprList exprList;

}


// ############################################################
package AST;

import java.io.*;

public class IfStatement extends Statement {
```

```
    public IfStatement( Expr expr, StatementList thenPart, StatementList elsePart ) {
        this.expr = expr;
        this.thenPart = thenPart;
        this.elsePart = elsePart;
    }

    public void genC( PW pw ) {

        pw.print("if ( ");
        expr.genC(pw);
        pw.out.println(" ) { ");
        if ( thenPart != null ) {
          pw.add();
          thenPart.genC(pw);
          pw.sub();
          pw.println("}");
        }
        if ( elsePart != null ) {
          pw.println("else {");
          pw.add();
          elsePart.genC(pw);
          pw.sub();
          pw.println("}");
        }
    }

    private Expr expr;
    private StatementList thenPart, elsePart;
}

// ###############################################################
package AST;

import java.io.PrintWriter;

public class IntegerType extends Type {

    public IntegerType() {
        super("integer");
    }

   public String getCname() {
      return "int";
   }
```

```
}

// ###############################################################
package AST;

import java.io.*; import java.util.*;

public class LocalVarList {

    public LocalVarList() {
        v = new ArrayList<Variable>();
    }

    public void addElement( Variable variable ) {
        v.add(variable);
    }

    public void genC( PW pw ) {

        for( Variable variable : v )
             pw.println( variable.getType().getCname() + " " +
                variable.getName() + ";" );
    }

    ArrayList<Variable> v;
}

// ###############################################################
package AST;

import java.io.*;

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;
}


// ##############################################################
package AST;

import java.lang.System;

import java.io.*;


public class PW {

   public void add() {
      currentIndent += step;
   }
   public void sub() {
      currentIndent -= step;
   }

   public void set( PrintWriter out ) {
      this.out = out;
      currentIndent = 0;
   }

   public void set( int indent ) {
      currentIndent = indent;
   }

   public void print( String s ) {
      out.print( space.substring(0, currentIndent) );
      out.print(s);
   }

   public void println( String s ) {
      out.print( space.substring(0, currentIndent) );
      out.println(s);
   }

   int currentIndent = 0;
   /* there is a Java and a Green mode.
      indent in Java mode:
```

```
      3 6 9 12 15 ...
      indent in Green mode:
      3 6 9 12 15 ...
   */
   static public final int green = 0, java = 1;
   int mode = green;
   public int step = 3;
   public PrintWriter out;


   static final private String space = "

}


// ############################################################
package AST;

import java.io.*;
import java.util.*;

public class ParamList {

    public ParamList() {
        v = new ArrayList<Parameter>();
    }

    public void addElement( Parameter parameter) {
        v.add(parameter);
    }

    public int getSize() {
        return v.size();
    }

/*    public Enumeration elements() {
        return v.elements();
    }
 *
 */
    public ArrayList<Parameter> getParamList() {
        return v;
    }

    public void genC( PW pw ) {
        Parameter p;
```

```
        Iterator e = v.iterator();
        int size = v.size();
        while ( e.hasNext() ) {
          p = (Parameter ) e.next();
          pw.out.print( p.getType().getCname() + " " + p.getName() );

          if ( --size > 0 )
            pw.out.print(", ");
        }
    }

    ArrayList<Parameter> v;
}


// ###########################################################
package AST;

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

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

}

// ###########################################################
package AST;

import java.io.PrintWriter;

public class ParenthesisExpr extends Expr {

    public ParenthesisExpr( Expr expr ) {
        this.expr = expr;
    }

    public void genC( PW pw ) {
        pw.out.print("(");
        expr.genC(pw, false);
        pw.print(")");
    }
```

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

    private Expr expr;
}

// ###############################################################
package AST;

import java.io.*;

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("}");
    }


}

// ###############################################################
package AST;

import java.io.*;

public class ProcedureCall extends Statement {

    public ProcedureCall( Procedure procedure, ExprList exprList ) {
        this.procedure = procedure;
        this.exprList = exprList;
    }
```

```
    public void genC( PW pw ) {
        pw.print( procedure.getName() + "(" );
        if ( exprList != null )
          exprList.genC(pw);
        pw.out.println(");");
    }

    Procedure procedure;
    ExprList exprList;

}

// ###############################################################
package AST;

import java.util.*; import java.io.*;

public class Program {

    public Program( ArrayList<Subroutine> procfuncList ) {
        this.procfuncList = procfuncList;
    }

    public void genC( PW pw) {

        pw.out.println("#include <stdio.h>");
        pw.out.println("");

          // generate code for procedures and functions
        for( Subroutine s : procfuncList ) {
            s.genC(pw);
            pw.out.println("");
            pw.out.println("");
        }

    }

    private ArrayList<Subroutine> procfuncList;
}

// ###############################################################
package AST;

import java.io.*;

public class ReadStatement extends Statement {
```

```
    public ReadStatement( Variable v ) {
        this.v = v;
    }

    public void genC( PW pw ) {
        if ( v.getType() == Type.charType )
          pw.print("{ char s[256]; gets(s); sscanf(s, \"%c\", &"  );
        else // should only be an integer
          pw.println("{ char s[256]; gets(s); sscanf(s, \"%d\", &" +
             v.getName() + "); }" );
    }
    private Variable v;
}

// ##############################################################
package AST;

import java.io.*;

public class ReturnStatement extends Statement {

    public ReturnStatement( Expr expr ) {
        this.expr = expr;
    }

    public void genC( PW pw ) {
        pw.print("return ");
        expr.genC(pw);
        pw.out.println(";");
    }

    private Expr expr;

}


// ##############################################################
package AST;

import java.io.PrintWriter;

abstract public class Statement {
    abstract public void genC( PW pw );
}
```

```
// ##############################################################
package AST;

import java.util.*; import java.io.*;

public class StatementList {

    public StatementList(ArrayList<Statement> v) {
        this.v = v;
    }

    public void genC( PW pw ) {

      for( Statement s: v )
          s.genC(pw);
    }

    private ArrayList<Statement> v;
}

// ##############################################################
package AST;

import java.io.*;

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;


}

// ############################################################
package AST;

import java.io.*;

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 static Type undefinedType = new UndefinedType();

    public String getName() {
        return name;
    }

    abstract public String getCname();

    private String name;
}


// ############################################################
package AST;

import java.io.PrintWriter; import Lexer.*;

public class UnaryExpr extends Expr {

    public UnaryExpr( Expr expr, Symbol op ) {
```

```
        this.expr = expr;
        this.op = op;
    }

    public void genC( PW pw ) {
        switch ( op ) {
            case PLUS :
              pw.out.print("+");
              break;
            case MINUS :
              pw.out.print("-");
              break;
            case NOT :
              pw.out.print("!");
              break;
        }
        expr.genC(pw, false);
    }

    public Type getType() {
        return expr.getType();
    }

    private Expr expr;
    private Symbol op;
}


// ###############################################################
package AST;

import java.io.*;

public class UndefinedType extends Type {
    // variables that are not declared have this type

   public UndefinedType() { super("undefined"); }

   public String getCname() {
      return "int";
   }

}


// ###############################################################
```

```
package AST;

import java.io.*;

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 Type getType() {
        return type;
    }

    private String name;
    private Type type;
}

// ###############################################################
package AST;

import java.io.PrintWriter;

public class VariableExpr extends Expr {

    public VariableExpr( Variable v ) {
        this.v = v;
    }

    public void genC( PW pw ) {
        pw.out.print( v.getName() );
    }

    public Type getType() {
        return v.getType();
```

```
    }

    private Variable v;
}

// ##############################################################
package AST;

import java.io.*;

public class WhileStatement extends Statement {

    public WhileStatement( Expr expr, Statement statement ) {
        this.expr = expr;
        this.statement = statement;
    }

    public void genC( PW pw ) {

        pw.print("while ( ");
        expr.genC(pw);
        pw.out.println(" )");
        if ( statement instanceof CompositeStatement )
           statement.genC(pw);
        else {
           pw.add();
           statement.genC(pw);
           pw.sub();
        }
    }

    private Expr expr;
    private Statement statement;
}

// ##############################################################
package AST;

import java.io.*;

public class WriteStatement extends Statement {

    public WriteStatement( Expr expr ) {
        this.expr = expr;
    }
```

```
    public void genC( PW pw ) {

        if ( expr.getType() == Type.booleanType ) {
          pw.print("printf(\"%s\\n\", ");
          expr.genC(pw, false);
          pw.out.print(" ? \"True\" : \"False\"");
        }
        else {
            if ( expr.getType() == Type.charType )
              pw.print("printf(\"%c\\n\", ");
            else
              pw.print("printf(\"%d\\n\", ");
            expr.genC(pw);
        }
        pw.out.println(" );");
    }


    private Expr expr;
}
```