The SimpleCompiler for the TEENSY language

Christian Collberg

August 23, 2009

1 The TEENSY Language

TEENSY is an extremely simple language with only two statements: assignment statements, and print statements. Expressions may contain identifiers, literal integers, and addition operators. They are evaluated strictly left-to-right. There are no loops, if-statements, procedures, classes, etc. Only integer expressions are supported

Here's an example TEENSY program:

```
1  BEGIN
2     x = 5;
3     y = 99;
4     z = y + x + 9;
5     PRINT z;
6  END
```

Here's TEENSY s grammar:

program	→	'BEGIN' stats 'END'	(1)
stats	→	stat stats	(2)
	|	ϵ	(3)
stat	→	ident '=' expr ';'	(4)
	|	'PRINT' expr ';'	(5)
expr	→	expr '+' expr	(6)
	|	ident	(7)
	|	int	(8)
ident	→	LETTER idp	(9)
idp	→	LETTER idp	(10)
	|	DIGIT idp	(11)
	|	ϵ	(12)
int	→	DIGIT intp	(13)
intp	→	DIGIT intp	(14)
	|	ϵ	(15)

There are no declarations per se, a variable comes into scope the first time it's assigned to. It's an error to use a variable before it's defined. TEENSY is case sensitive.

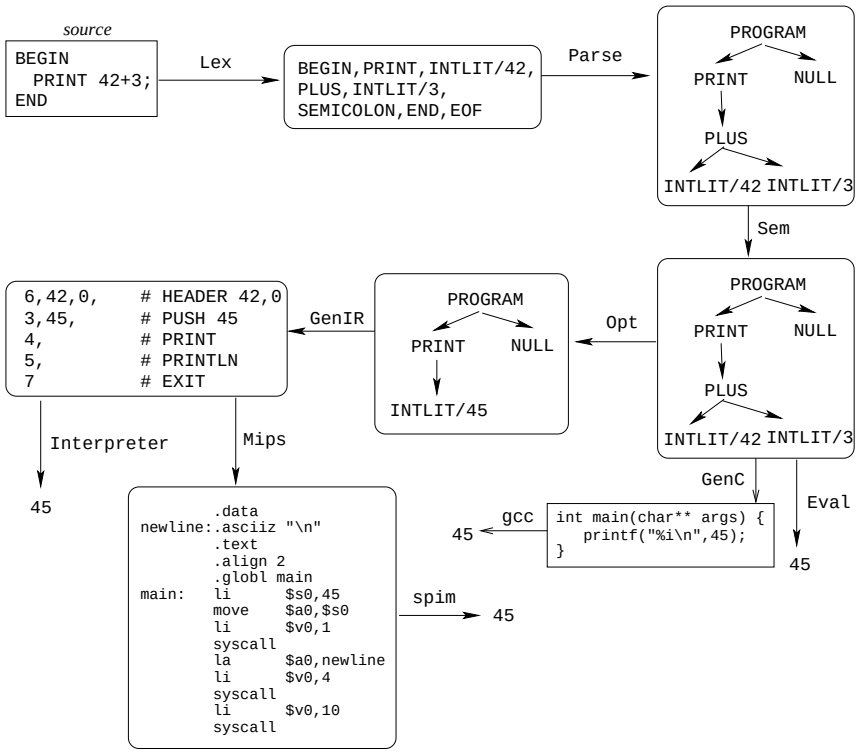

Figure 1: Translation Overview.

---

2 The SimpleCompiler

Simpleis a trivial compiler for TEENSY In spite of its simplicity the structure of Simpleis the same as that of a full-fledged compiler for a real language.

In less than 1000 lines of (uncommented) Java Simpledoes:

- lexical analysis (`Token`, `Lex`),
- parsing (`Matcher`, `Parse`),
- abstract syntax tree construction (`Parse`, `AST`, `PROGRAM`, `STAT`, `STATSEQ`, `ASSIGN`, `PRINT`, `EXPR`, `NULL`, `IDENT`, `INTLIT`, `BINOP`),
- semantic analysis (`SyTab`, `Sem`),
- tree-walk interpretation (`Eval`), optimization (`Opt`),
- C code generation (`GenC`),
- intermediate code generation (`IR`, `GenIR`),
- stack-code interpretation (`Interpreter`), and
- machine-code generation (`GenMips`).

The class `Compiler` ties it all together.

Figure 1 shows an overview of how Simpletranslates and executes the program:

```
1  BEGIN
2     PRINT 42+3;
3  END
```

Lexical Analysis

The purpose of the scanner (lexical analyser) is to read the input file and split it into lexical units. `Token` defines the lexical items of the language. `Lex` does the actual scanning. It provides a method `nextToken()` which is called by the parser to get one token at a time. The `IDENT` and `INTLIT` tokens also have data associated with them, the identifier/literal integer that was actually read. A special end-of-file token (`EOF`) is generated when all the input is consumed.

Syntactic Analysis

The purpose of the parser (syntactic analyser) is to determine the structure of the input program. `Matcher` consists of a set of mutually recursive procedures which get one token at a time from the scanner and determine if the program has the correct syntax. If it does, `Matcher` will print a trace of the calls to the recursive procedures, otherwise an error message will be printed.

This is called a *recursive descent* parser.

`Parse` is similar to `Matcher`, except that a the result of the parse is a tree, an *abstract syntax tree* (AST). The AST will form the basis for all further processing, including semantic analysis and intermediate code generation.

Abstract Syntax

The parser produces an abstract syntax tree, the AST. The structure of the AST is the same as the basic structure of the language, and is called the *abstract syntax*. For example, since a binary operator (`+` in Teensy takes two arguments which both are expressions, the abstract syntax rule for BINOP is

$$\text{BINOP} \quad \rightarrow \quad \text{op EXPR EXPR}$$

The Java class that implements BINOP (`BINOP`) is a direct translation of the abstract rule:

```
1  /* Copyright 2001, Christian Collberg, collberg@cs.arizona.edu. */
2
3  public class BINOP extends EXPR {
4      public int OP;
5      public EXPR left;
6      public EXPR right;
7
8      public BINOP(int OP, EXPR left, EXPR right) {
9         this.OP = OP; this.left = left; this.right = right;
10     }
11
12     public String toString() {
13        String op = "";
14        if (OP == Token.PLUS)
15            op = "+";
16        else if (OP == Token.STAR)
17            op = "*";
18        return "(" + op + ", " + left.toString() + ", " + right.toString() + ")";
19     }
20 }
```

I.e. `BINOP` has three fields (`op`, `left`, and `right`). The rest of the code in the class is just the constructor and `toString()`.

Here's the complete abstract syntax for the TEENSY language:

$$
\begin{aligned}
\text{PROGRAM} &\rightarrow \text{STATSEQ} && (16)\\
\text{STATSEQ} &\rightarrow \text{STAT STATSEQ} && (17)\\
&\mid \text{NULL} && (18)\\
\text{STAT} &\rightarrow \text{ASSIGN} && (19)\\
&\mid \text{PRINT} && (20)\\
\text{ASSIGN} &\rightarrow \text{ident EXPR} && (21)\\
\text{PRINT} &\rightarrow \text{EXPR} && (22)\\
\text{EXPR} &\rightarrow \text{BINOP} && (23)\\
&\mid \text{IDENT} && (24)\\
&\mid \text{INTLIT} && (25)\\
\text{BINOP} &\rightarrow \text{op EXPR EXPR} && (26)\\
\text{IDENT} &\rightarrow \text{ident} && (27)\\
\text{INTLIT} &\rightarrow \text{int} && (28)
\end{aligned}
$$

Each of the rules translates into a class in Simple

Semantic Analysis

The only possible semantic error in TEENSY is a variable being used before it's first defined. `Sem` walks the AST, inserts any identifiers found on the left hand side of an assignment statement into the symbol table, and prints an error message if an expression contains an undefined identifier.

The symbol table is defined in `SyTab`. Each name inserted into the table is mapped to a number, starting at 0. These identifier numbers are used during intermediate code generation since the IR uses numbers, not names, to identify variables.

Optimization

`Opt` walks the AST and attempts to evaluate constant sub-expressions. The result is new, simplified tree.

The optimizer is not very clever. While `PRINT 3+4+5+x` will simplify to `PRINT 12+x`, `PRINT x+3+4+5` will not be optimized at all.

C Code Generation

`GenC` walks the AST and generates an equivalent C program, which can then be compiled and executed. Any code optimization or smart code generation done by the C compiler will of course benefit our program as well.

IR Generation

Simpleuses a very simple, stack-based, intermediate code. It is defined in `IR`. There are only 8 bytecodes:

mnemonic	opcode	stack-pre	stack-post	side-effects
ADD	0	[A,B]	[A+B]	
LOAD X	1	[]	[Memory[X]]	
STORE X	2	[A]	[]	Memory[X] = A
PUSH X	3	[]	[X]	
PRINT	4	[A]	[]	Print A
PRINTLN	5	[]	[]	Print a newline
HEADER M,V	6	[]	[]	
EXIT	7	[]	[]	The interpreter exits

The `HEADER` bytecode must be the first one in the instruction stream. It's first argument is a magic number (*42*) and the second argument the number of variables used in the following code.

`GenIR` generates a bytecode program from the abstract syntax tree.

Code generation

`GenMips` reads the IR code and prints the corresponding `Mips` assembly code. We use two internal data structures:

- The `regs` array is used to manage register allocation. The variable `nextReg` simply keeps track of the next register to allocate. We don't attempt to reuse freed registers, and there's no atempt to recover from running out of registers completely.
- We use a stack of register names to hold the currently active registers. For example, when generating code for the expression `x+y` the stack would hold the registers into which `x` and `y` have been loaded, say `$s0` and `$s1`, respectively. When the code generator reaches the IR `PLUS` operator it will pop `$s0` and `$s1` off the stack, generate the instruction `add $s2,$s0,$s1`, and push `$s2` onto the stack.

Execution

There are three ways to execute Teensy programs. `Eval` will walk and evaluate the abstract syntax tree; `Interpreter` will interpret the stack-based IR bytecodes; and we can use `spim` to execute the generated `Mips` code.

Putting it all together

To simplify testing and experimentation, each Simplepass is self-contained. `Compiler` ties everything together. Here are some examples of how the different passes can be executed:

```
java Lex test1                  # Print the tokens
java Matcher test2              # Print the ``parse tree''
java Parse test2                # Print the AST and syntactic error messages
java Sem test4                  # Print semantic error messages
java Eval test4                 # Evaluate the AST
java GenIR test4                # Generate IR code

java GenIR test4 > test4.vm     # Generate IR code
java Interpreter test4.vm       # Interpret the IR code

java Opt test5                  # Optimize the AST

java GenMips test4.vm           # Print Mips code generated from the IR
java GenMips test4.vm > test4.s   # Generate Mips code
spim -file test4.s              # Execute Mips code

java Compiler -ir test4         # Generate IR code
java Compiler -mips test4       # Generate Mips code
```

3 Token.java

```
1  /* Copyright 2001, Christian Collberg, collberg@cs.arizona.edu. */
2
3  public class Token {
4      public final static int ILLEGAL    = 0;
5      public final static int PLUS       = 1;
6      public final static int INTLIT     = 2;
7      public final static int IDENT      = 3;
8      public final static int SEMICOLON  = 4;
9      public final static int EQUAL      = 5;
10     public final static int BEGIN      = 6;
11     public final static int END        = 7;
12     public final static int PRINT      = 8;
13     public final static int STAR       = 9;
14     public final static int EOF        = 10;
15
16     public int kind;
17     public String ident;
18     public int value;
19     public int position;
20
21     public Token(int kind, int position) {
22        this.kind = kind; this.position = position;
23     }
24
25     public Token(int kind, int position, String ident) {
26       this.kind = kind; this.ident = ident; this.position = position;
27     }
28
29     public Token(int kind, int position, int value) {
30       this.kind = kind; this.value = value; this.position = position;
31     }
32
33     public String toString() {
34        String s = "@" + position + " ";
35        switch (kind) {
36           case ILLEGAL   : return s + "ILLEGAL";
37           case PRINT     : return s + "PRINT";
38           case PLUS      : return s + "PLUS";
39           case STAR      : return s + "STAR";
40           case INTLIT    : return s + "INTLIT" + ": " + value;
41           case IDENT     : return s + "IDENT" + ": " + ident;
42           case SEMICOLON : return s + "SEMICOLON";
43           case EQUAL     : return s + "EQUAL";
44           case BEGIN     : return s + "BEGIN";
45           case END       : return s + "END";
46           case EOF       : return s + "EOF";
47           default        : return "";
48        }
49     }
50 }
```

4 Lex.java

```
1  /* Copyright 2001, Christian Collberg, collberg@cs.arizona.edu. */
2
3  import java.lang.*;
4  import java.io.*;
5
6  public class Lex {
7      LineNumberReader str;   // input stream
8      char ch;                // lookahead character
9      boolean done = false;   // reached end-of-file
10
11     public Lex(String filename) throws IOException {
12         str = new LineNumberReader(new FileReader(filename));
13         get();
14     }
15
16     int pos() {return str.getLineNumber();}
17
18     // read the next input character
19     void get() {
20         try {
21            int r = str.read();
22            ch = (char)r;
23            if (r == -1) done=true;
24         } catch (Exception e) {
25            done=true;
26         }
27     }
28
29     // We've found the beginning of a literal integer. Continue
30     // scanning it and convert the resulting string to an int.
31     Token scanNumber() {
32         String s = "";
33         int ival = -1;
34         while ((!done) && Character.isDigit(ch)) {s+=ch; get();}
35         try {
36            ival = Integer.parseInt(s.toString());
37         } catch (NumberFormatException e) {
38            System.err.println("not an integer");
39         }
40         return new Token(Token.INTLIT, pos(), ival);
41     }
42
43     // We've found the beginning of an identifier or keyword.
44     // Continue scanning until the end is found, check if
45     // the string's a keyword, otherwise return the IDENT token.
46     Token scanName()  {
47         String ident = "";
48         while ((!done) && Character.isLetterOrDigit(ch)) {ident+=ch; get();}
49         if (ident.equals("BEGIN"))      return new Token(Token.BEGIN, pos());
50         else if (ident.equals("END"))   return new Token(Token.END, pos());
51         else if (ident.equals("PRINT")) return new Token(Token.PRINT, pos());
52         else                            return new Token(Token.IDENT, pos(), ident);
53     }
54
55     // Used by the parser to get the next token. EOF will
56     // be the last token generated.
57     public Token nextToken() {
58         while ((!done) && ch <= ' ') get();  // scan over whitespace
59         if (done) return new Token(Token.EOF, pos());
60         switch (ch) {
61             case '+': get(); return new Token(Token.PLUS, pos());
```

```
62              case '*': get(); return new Token(Token.STAR, pos());
63              case ';': get(); return new Token(Token.SEMICOLON, pos());
64              case '=': get(); return new Token(Token.EQUAL, pos());
65              default:  if (Character.isLetter(ch)) return scanName();
66                        else if (Character.isDigit(ch)) return scanNumber();
67                        else {
68                            System.err.println("illegal character " + ch);
69                            get(); return new Token(Token.ILLEGAL, pos());
70                        }
71          }
72      }
73
74      public static void main (String args[]) throws IOException{
75         Lex scanner = new Lex(args[0]);
76         while(true) {
77            Token token = scanner.nextToken();
78            System.out.println(token.toString());
79            if (token.kind == Token.EOF) return;
80         }
81      }
82  }
```

---

5 Matcher.java

```
1  /* Copyright 2001, Christian Collberg, collberg@cs.arizona.edu. */
2
3  import java.io.*;
4
5  public class Matcher {
6
7      Lex scanner;
8      Token currentToken;
9
10     public Matcher (Lex scanner) {
11        this.scanner = scanner;
12        next();
13     }
14
15     // Get the next token from the lexer.
16     void next() {
17        currentToken = scanner.nextToken();
18     }
19
20     // Check if the current token is of kind tokenKind.
21     // tokenKind should be one of the constants defined in
22     // Token.java.
23     boolean lookahead(int tokenKind) {
24        return currentToken.kind == tokenKind;
25     }
26
27     // Make sure that the current token is of kind tokenKind.
28     // If not, print an error message. Advance to the next
29     // token.
30     void match(int tokenKind) {
31        if (!lookahead(tokenKind)) {
32             System.err.println("Parsing error, line " + currentToken.position);
33             System.exit(-1);
34         }
35         next();
36     }
37
38     // ENTER and EXIT are used to print out a representation
39     // of the parse tree.
40     String indent = "";
41     void ENTER(String rule) {
42       System.out.println(indent + "ENTER " + rule);
43       indent += "   ";
44     }
45
46     void EXIT(String rule) {
47         indent = indent.substring(3);
48         System.out.println(indent + "EXIT " + rule);
49     }
50
51     // Parse an ident or integer literal in an expression.
52     void factor() {
53        ENTER("factor");
54        if (lookahead(Token.IDENT)) {
55            match(Token.IDENT);
56        } else if (lookahead(Token.INTLIT)) {
57            match(Token.INTLIT);
58        }
59        EXIT("factor");
60     }
61
```

```
62      // Parse an expression which is a summation of variables
63      // and integer literals.
64      void expr() {
65         ENTER("expr");
66         factor();
67         while (lookahead(Token.PLUS)) {
68            match(Token.PLUS);
69            factor();
70         }
71         EXIT("expr");
72      }
73
74      // Parse an assignment statement, 'ident = expression'.
75      void assign() {
76         ENTER("assign");
77         match(Token.IDENT);
78         match(Token.EQUAL);
79         expr();
80         EXIT("assign");
81      }
82
83      // Parse a print statement, 'PRINT expression'.
84      void print() {
85         ENTER("print");
86         match(Token.PRINT);
87         expr();
88         EXIT("print");
89      }
90
91      // Parse a list of statements.
92      void stats() {
93         ENTER("stats");
94         if (lookahead(Token.IDENT)) {
95            assign();
96            match(Token.SEMICOLON);
97            stats();
98         } else if (lookahead(Token.PRINT)) {
99             print();
100            match(Token.SEMICOLON);
101            stats();
102        }
103        EXIT("stats");
104     }
105
106     // Parse a program, 'BEGIN statement sequence END'.
107     public void parse() {
108        ENTER("parse");
109        match(Token.BEGIN);
110        stats();
111        match(Token.END);
112        match(Token.EOF);
113        EXIT("parse");
114     }
115
116     public static void main (String args[]) throws IOException{
117         Lex scanner = new Lex(args[0]);
118         Matcher parser = new Matcher(scanner);
119         parser.parse();
120     }
121 }
```

---

6 Parse.java

```
1  /* Copyright 2001, Christian Collberg, collberg@cs.arizona.edu. */
2
3  import java.io.*;
4
5  public class Parse {
6      Lex scanner;
7      Token currentToken;
8      public AST ast;
9
10     public Parse (Lex scanner) {
11        this.scanner = scanner;
12        next();
13        ast = parse();
14     }
15
16     void next() {
17        currentToken = scanner.nextToken();
18     }
19
20     boolean lookahead(int tokenKind) {
21        return currentToken.kind == tokenKind;
22     }
23
24     void match(int tokenKind) {
25        if (!lookahead(tokenKind)) {
26             System.err.println("Parsing error, line " + currentToken.position);
27             System.exit(-1);
28         }
29         next();
30     }
31
32     // Build an AST node for either a variable reference
33     // or a literal integer reference.
34     EXPR factor() {
35         EXPR expr = null;
36         if (lookahead(Token.IDENT)) {
37            expr = new IDENT(currentToken.ident);
38            match(Token.IDENT);
39         } else if (lookahead(Token.INTLIT)) {
40            expr = new INTLIT(currentToken.value);
41            match(Token.INTLIT);
42         }
43         return expr;
44     }
45
46     // Build an AST subtree for an expression.
47    EXPR expr() {
48       EXPR f = factor();
49       while (true) {
50          if (lookahead(Token.PLUS)) {
51             match(Token.PLUS);
52             EXPR e = factor();
53             f = new BINOP(Token.PLUS, f, e);
54          } else if (lookahead(Token.STAR)) {
55             match(Token.STAR);
56             EXPR e = factor();
57             f = new BINOP(Token.STAR, f, e);
58          } else
59              break;
60       }
61       return f;
```

```
62      }
63
64       // Build an ASSIGN subtree.
65      STAT assign() {
66          String ident = currentToken.ident;
67          match(Token.IDENT);
68          match(Token.EQUAL);
69          EXPR e = expr();
70          return new ASSIGN(ident, e);
71      }
72
73       // Build a PRINT subtree.
74       STAT print() {
75          match(Token.PRINT);
76          EXPR e = expr();
77          return new PRINT(e);
78       }
79
80       // Build a STATSEQ subtree. The bottom/rightmost
81       // subtree will be a NULL node, indicating the
82       // end of the statement sequence.
83       STATSEQ stats() {
84          STAT stat;
85          if (lookahead(Token.IDENT)) {
86             stat = assign();
87          } else if (lookahead(Token.PRINT)) {
88             stat = print();
89          } else
90             return new NULL();
91          match(Token.SEMICOLON);
92          STATSEQ next = stats();
93          return new STATSEQ(stat, next);
94       }
95
96       // Build a tree whose root is a PROGRAM node.
97       AST parse() {
98          match(Token.BEGIN);
99          STATSEQ s = stats();
100         PROGRAM p = new PROGRAM(s);
101         match(Token.END);
102         match(Token.EOF);
103         return p;
104      }
105
106      public static void main (String args[]) throws IOException{
107         Lex scanner = new Lex(args[0]);
108         Parse parser = new Parse(scanner);
109         System.out.println(parser.ast.toString());
110      }
111  }
```

---

7 AST.java

```
1  /* Copyright 2001, Christian Collberg, collberg@cs.arizona.edu. */
2
3  // The base class for all abstract syntax tree classes.
4  public abstract class AST {}
```

8 PROGRAM.java

```
1  /* Copyright 2001, Christian Collberg, collberg@cs.arizona.edu. */
2
3  public class PROGRAM extends AST {
4      public STATSEQ stats;
5      public PROGRAM (STATSEQ stats) {this.stats = stats;}
6      public String toString() {
7         return "(PROGRAM\n" +
8                     stats.toString("   ") +
9                "\n)";
10     }
11 }
```

9 STAT.java

```
1  /* Copyright 2001, Christian Collberg, collberg@cs.arizona.edu. */
2
3  public abstract class STAT extends AST {}
```

10 STATSEQ.java

```
1  /* Copyright 2001, Christian Collberg, collberg@cs.arizona.edu. */
2
3  public class STATSEQ extends AST {
4      public STAT stat;
5      public STATSEQ next;
6
7      public STATSEQ() {}
8
9      public STATSEQ(STAT stat, STATSEQ next) {
10       this.stat = stat;
11       this.next = next;
12     }
13
14     public String toString(String indent) {
15        return indent +
16                "(STATSEQ\n" +
17                 indent + "   " + stat.toString() + "\n" +
18                 next.toString(indent + "   ") +
```

```
19                  "\n" + indent + ")";
20      }
21  }
```

---

11 ASSIGN.java

```
1   /* Copyright 2001, Christian Collberg, collberg@cs.arizona.edu. */
2
3   public class ASSIGN extends STAT {
4       public String ident;
5       public EXPR expr;
6       public ASSIGN(String ident, EXPR expr) {this.ident = ident;this.expr = expr;}
7       public String toString() {return "(ASSIGN " + ident + ", " + expr.toString() + ")"; }
8   }
```

---

12 PRINT.java

```
1   /* Copyright 2001, Christian Collberg, collberg@cs.arizona.edu. */
2
3   public class PRINT extends STAT {
4       public EXPR expr;
5       public PRINT(EXPR expr) {this.expr = expr;}
6       public String toString() {return "(PRINT " + expr.toString() + ")";}
7   }
```

---

13 EXPR.java

```
1   /* Copyright 2001, Christian Collberg, collberg@cs.arizona.edu. */
2
3   public abstract class EXPR extends AST {}
```

---

14 NULL.java

```
1   /* Copyright 2001, Christian Collberg, collberg@cs.arizona.edu. */
2
3   public class NULL extends STATSEQ {
4       public NULL() {}
5       public String toString(String indent) {
6          return indent + "NULL";
7       }
8   }
```

---

15 IDENT.java

```
1  /* Copyright 2001, Christian Collberg, collberg@cs.arizona.edu. */
2
3  public class IDENT extends EXPR {
4      public String ident;
5      public IDENT(String ident) {this.ident = ident;}
6      public String toString() {return "(IDENT " + ident + ")";}
7  }
```

16 INTLIT.java

```
1  /* Copyright 2001, Christian Collberg, collberg@cs.arizona.edu. */
2
3  public class INTLIT extends EXPR {
4      public int val;
5      public INTLIT(int val) {this.val = val;}
6      public String toString() {return "(INTLIT " + val + ")";}
7  }
```

17 BINOP.java

```
1  /* Copyright 2001, Christian Collberg, collberg@cs.arizona.edu. */
2
3  public class BINOP extends EXPR {
4      public int OP;
5      public EXPR left;
6      public EXPR right;
7
8      public BINOP(int OP, EXPR left, EXPR right) {
9         this.OP = OP; this.left = left; this.right = right;
10     }
11
12     public String toString() {
13        String op = "";
14        if (OP == Token.PLUS)
15            op = "+";
16        else if (OP == Token.STAR)
17            op = "*";
18        return "(" + op + ", " + left.toString() + ", " + right.toString() + ")";
19     }
20 }
```

18 SyTab.java

```
1  /* Copyright 2001, Christian Collberg, collberg@cs.arizona.edu. */
2
3  import java.util.*;
4
5  public class SyTab {
6      Hashtable sytab = new Hashtable();
7      int currentID = 0;
8
9      // Insert ident into the symbol table, unless it's
10     // already there. Assign a new number to the identifier.
11     public void insert(String ident) {
12        if (!sytab.containsKey(ident))
13           sytab.put(ident, new java.lang.Integer(currentID++));
14     }
15
16     // Return the number of 'ident'. If 'ident' is not in the
17     // symbol table, return -1.
18     public int lookup(String ident) {
19       if (sytab.containsKey(ident))
20          return ((Integer)sytab.get(ident)).intValue();
21       else
22          return -1;
23     }
24
25     // Return the number of identifiers in the table.
26     public int size() {
27        return sytab.size();
28     }
29 }
```

19 Sem.java

```
1  /* Copyright 2001, Christian Collberg, collberg@cs.arizona.edu. */
2
3  import java.io.*;
4  import java.util.*;
5
6  public class Sem {
7      AST ast;
8      public SyTab sytab = new SyTab();
9
10     public Sem (AST ast) {
11        this.ast = ast;
12        program((PROGRAM) ast);
13     }
14
15     // Start walking the AST at the root, PROGRAM, node.
16     void program(PROGRAM n) {
17        stats(n.stats);
18     }
19
20     // Recursively walk a sequence a statements. NULL indicates
21     // the end of the sequence.
22     void stats(STATSEQ n) {
23        if (n instanceof NULL) return;
24        stat(n.stat);
25        stats(n.next);
26     }
27
28     // Walk assignment or print statements.
29     void stat(STAT n) {
30        if (n instanceof ASSIGN)
31           assign((ASSIGN)n);
32        else if (n instanceof PRINT)
33           print((PRINT)n);
34     }
35
36     // Insert the identifier on the left hand side of the
37     // assignment statement into the symbol table, if it's
38     // not already there.
39     void assign(ASSIGN n) {
40        sytab.insert(n.ident);
41        expr(n.expr);
42     }
43
44     // Walk a print statement.
45     void print(PRINT n) {
46        expr(n.expr);
47     }
48
49     // Walk an arithmetic expression.
50     void expr(EXPR n) {
51        if (n instanceof IDENT)
52            ident((IDENT) n);
53        else if (n instanceof INTLIT)
54            intlit((INTLIT) n);
55        else if (n instanceof BINOP)
56            binop((BINOP) n);
57     }
58
59     // If an identifier in an expression has not been assigned
60     // to before it's used, issue an error message.
61     void ident(IDENT n) {
```

```
62          if (sytab.lookup(n.ident) < 0)
63             System.err.println("Identifier not declared: " + n.ident);
64       }
65
66       // Walk an integer literal.
67       void intlit(INTLIT n) {
68       }
69
70       // Walk a binary arithmetic operator.
71       void binop(BINOP n) {
72          expr(n.left);
73          expr(n.right);
74       }
75
76       public static void main (String args[]) throws IOException{
77           Lex scanner = new Lex(args[0]);
78           Parse parser = new Parse(scanner);
79           Sem sem = new Sem(parser.ast);
80       }
81    }
```

---

20 Eval.java

```
1  /* Copyright 2001, Christian Collberg, collberg@cs.arizona.edu. */
2  import java.io.*;
3  import java.util.*;
4
5  public class Eval {
6      Sem sem;
7      int[] memory;        // Variable store.
8
9      public Eval (Sem sem) {
10        this.sem = sem;
11        program((PROGRAM) sem.ast);
12     }
13
14     // Start evaluating an AST at the root, PROGRAM, node.
15     // We must have performed semantic analysis before
16     // the evaluation, so that variables have been assigned
17     // identifier numbers. These numbers are used to index
18     // 'memory', an array that holds current variable values.
19     void program(PROGRAM n) {
20        memory = new int[sem.sytab.size()];
21        stats(n.stats);
22     }
23
24     void stats(STATSEQ n) {
25        if (n instanceof NULL) return;
26        stat(n.stat);
27        stats(n.next);
28     }
29
30     void stat(STAT n) {
31        if (n instanceof ASSIGN)
32           assign((ASSIGN)n);
33        else if (n instanceof PRINT)
34           print((PRINT)n);
35     }
36
37     // Evaluate the expression, and assign the result to
38     // the appropriate variable in 'memory'.
39     void assign(ASSIGN n) {
40         int v = expr(n.expr);
41         memory[sem.sytab.lookup(n.ident)] = v;
42     }
43
44     // Evaluate the expression, and print the result.
45     void print(PRINT n) {
46        int v = expr(n.expr);
47        System.out.println(v);
48     }
49
50     // Evaluate an expression.
51     int expr(EXPR n) {
52        if (n instanceof IDENT)
53            return ident((IDENT) n);
54        else if (n instanceof INTLIT)
55            return intlit((INTLIT) n);
56        else if (n instanceof BINOP)
57            return binop((BINOP) n);
58        return -1;
59     }
60
61     // Look up the identifier number, and return the current
```

```
62      // value from the memory cell.
63      int ident(IDENT n) {
64          return memory[sem.sytab.lookup(n.ident)];
65      }
66
67      int intlit(INTLIT n) {
68          return n.val;
69      }
70
71      // Evaluate an binary arithmetic expression.
72     int binop(BINOP n) {
73        int l = expr(n.left);
74        int r = expr(n.right);
75        if (n.OP == Token.PLUS)
76            return l + r;
77        return -1;
78      }
79
80     public static void main (String args[]) throws Exception{
81         Lex scanner = new Lex(args[0]);
82         Parse parser = new Parse(scanner);
83         Sem sem = new Sem(parser.ast);
84         Eval eval = new Eval(sem);
85      }
86  }
87
88
```

21 Opt.java

```
1  /* Copyright 2001, Christian Collberg, collberg@cs.arizona.edu. */
2
3  import java.io.*;
4  import java.util.*;
5
6  public class Opt {
7      Sem sem;
8
9      public Opt (Sem sem) {
10        this.sem = sem;
11        program((PROGRAM) sem.ast);
12     }
13
14     // Begin optimizing the AST at the root, PROGRAM, node.
15     void program(PROGRAM n) {
16        stats(n.stats);
17     }
18
19     void stats(STATSEQ n) {
20        if (n instanceof NULL) return;
21        stat(n.stat);
22        stats(n.next);
23     }
24
25     void stat(STAT n) {
26        if (n instanceof ASSIGN)
27           assign((ASSIGN)n);
28        else if (n instanceof PRINT)
29           print((PRINT)n);
30     }
31
32     // Optimize the expression and replace the old
33     // expression subtree with the new one.
34     void assign(ASSIGN n) {
35        n.expr = expr(n.expr);
36     }
37
38     // Optimize the expression and replace the old
39     // expression subtree with the new one.
40     void print(PRINT n) {
41        n.expr = expr(n.expr);
42     }
43
44     EXPR expr(EXPR n) {
45         if (n instanceof IDENT)
46            return n;
47         else if (n instanceof INTLIT)
48            return n;
49         else if (n instanceof BINOP)
50            return binop((BINOP) n);
51         return null;
52     }
53
54     // Optimize a binary arithmetic expression by replacing
55     // 'BINOP(+,INTLIT,INTLIT)' with 'INTLIT(INTLIT+INTLIT)'.
56     // This only partially works: 'x+5+6' is parsed as
57     // 'BINOP(+,BINOP(+,IDENT(x),INTLIT(5)),INTLIT(6))', and
58     // hence we will never see the '5+6' subn-expression.
59     EXPR binop(BINOP n) {
60         EXPR L = expr(n.left);
61         EXPR R = expr(n.right);
```

```
62          if ((n.OP == Token.PLUS) &&
63              (L instanceof INTLIT) &&
64              (R instanceof INTLIT)) {
65              int l = ((INTLIT)L).val;
66              int r = ((INTLIT)R).val;
67              return new INTLIT(l+r);
68          }
69          return new BINOP(Token.PLUS, L, R);
70      }
71
72      public static void main (String args[]) throws IOException{
73          Lex scanner = new Lex(args[0]);
74          Parse parser = new Parse(scanner);
75          Sem sem = new Sem(parser.ast);
76          Opt opt = new Opt(sem);
77          System.out.println(opt.sem.ast.toString());
78      }
79  }
```

---

22 GenC.java

```
1  /* Copyright 2001, Christian Collberg, collberg@cs.arizona.edu. */
2
3  import java.io.*;
4  import java.util.*;
5
6  public class GenC {
7      Sem sem;
8
9      public GenC (Sem sem) {
10        this.sem = sem;
11        program((PROGRAM) sem.ast);
12     }
13
14     public String code = "";
15     void add(String instr) {
16        code += instr + "\n";
17     }
18
19     // Start generating IR code from the AST.
20     void program(PROGRAM n) {
21        add("#include<stdio.h>");
22        add("int main(char* args) {");
23        add("   int vars[" + sem.sytab.size() + "];");
24        stats(n.stats);
25        add("   exit(0);");
26        add("}");
27     }
28
29     void stats(STATSEQ n) {
30        if (n instanceof NULL) return;
31        stat(n.stat);
32        stats(n.next);
33     }
34
35     void stat(STAT n) {
36        if (n instanceof ASSIGN)
37           assign((ASSIGN)n);
38        else if (n instanceof PRINT)
39           print((PRINT)n);
40     }
41
42     void assign(ASSIGN n) {
43         String e = expr(n.expr);
44         add("   vars[" + sem.sytab.lookup(n.ident) + "] = " + e + ";");
45     }
46
47     void print(PRINT n) {
48        String e = expr(n.expr);
49        add("   printf(\"%i\\n\"," + e + ");");
50     }
51
52     String expr(EXPR n) {
53        if (n instanceof IDENT)
54          return ident((IDENT) n);
55        else if (n instanceof INTLIT)
56          return intlit((INTLIT) n);
57        else if (n instanceof BINOP)
58          return binop((BINOP) n);
59        return "";
60     }
61
```

```
62      String ident(IDENT n) {
63          return "vars[" + sem.sytab.lookup(n.ident) + "]";
64      }
65
66      String intlit(INTLIT n) {
67          return java.lang.Integer.toString(n.val);
68      }
69
70      String binop(BINOP n) {
71         String l = expr(n.left);
72         String r = expr(n.right);
73         if (n.OP == Token.PLUS)
74            return "(" + l + " + " + r + ")";
75         else
76            return "";
77      }
78
79      public void write () {
80        System.out.println(code);
81      }
82
83      public static void main (String args[]) throws Exception{
84         Lex scanner = new Lex(args[0]);
85         Parse parser = new Parse(scanner);
86         Sem sem = new Sem(parser.ast);
87         GenC ir = new GenC(sem);
88         ir.write();
89      }
90  }
91
92
```

---

23 IR.java

```
1  /* Copyright 2001, Christian Collberg, collberg@cs.arizona.edu. */
2
3  import java.lang.*;
4  import java.io.*;
5
6  public class IR {
7
8     public static final int ADD    = 0;
9     public static final int LOAD   = 1;
10    public static final int STORE  = 2;
11    public static final int PUSH   = 3;
12    public static final int PRINT  = 4;
13    public static final int PRINTLN= 5;
14    public static final int HEADER = 6;
15    public static final int EXIT   = 7;
16    public static final int MUL    = 8;
17
18    public static final int MAGIC  = 42;
19
20    // Read an IR program from file. There is one integer
21    // code per line.
22    public static int[] read(String filename) {
23         int[] code = new int[100];
24         int pc = 0;
25         try {
26            BufferedReader str = new BufferedReader(new FileReader(filename));
27            while(true)
28               code[pc++] = Integer.parseInt(str.readLine());
29         } catch (Exception e) {
30         }
31         return code;
32     }
33
34     // Write an IR program to standard out, one integer code
35     // per line.
36     public static void write(int code[], int pc) {
37         for(int i=0; i<pc; i++)
38             System.out.println(code[i]);
39     }
40
41 }
```

24 GenIR.java

```
1  /* Copyright 2001, Christian Collberg, collberg@cs.arizona.edu. */
2
3  import java.io.*;
4  import java.util.*;
5
6  public class GenIR {
7      Sem sem;
8      int pc = 0;
9
10     public GenIR (Sem sem) {
11        this.sem = sem;
12        program((PROGRAM) sem.ast);
13     }
14
15     // List of generated IR instructions.
16     public int[] code = new int[100];
17     void add(int instr) {
18        code[pc++] = instr;
19     }
20
21     // Start generating IR code from the AST.
22     void program(PROGRAM n) {
23        add(IR.HEADER);  add(IR.MAGIC); add(sem.sytab.size());
24        stats(n.stats);
25        add(IR.EXIT);
26     }
27
28     void stats(STATSEQ n) {
29        if (n instanceof NULL) return;
30        stat(n.stat);
31        stats(n.next);
32     }
33
34     void stat(STAT n) {
35        if (n instanceof ASSIGN)
36           assign((ASSIGN)n);
37        else if (n instanceof PRINT)
38           print((PRINT)n);
39     }
40
41     // Generate code for an assignment statement. We first
42     // generate code that pushes the value of the expression
43     // on the stack, then issue a 'STORE' instruction that
44     // pops this value off the stack and assigns it to the
45     // appropriate memory cell.
46     void assign(ASSIGN n) {
47         expr(n.expr);
48         add(IR.STORE);
49         add(sem.sytab.lookup(n.ident));
50     }
51
52     // Generate code for a print statement. We first
53     // generate code that pushes the value of the expression
54     // on the stack, then issue a 'PRINT' instruction which
55     // pops this value off the stack and prints it.
56     void print(PRINT n) {
57        expr(n.expr);
58        add(IR.PRINT);
59        add(IR.PRINTLN);
60     }
61
```

```
62      void expr(EXPR n) {
63         if (n instanceof IDENT)
64           ident((IDENT) n);
65         else if (n instanceof INTLIT)
66           intlit((INTLIT) n);
67         else if (n instanceof BINOP)
68           binop((BINOP) n);
69      }
70
71      // Push the value of a variable.
72      void ident(IDENT n) {
73          add(IR.LOAD);
74          add(sem.sytab.lookup(n.ident));
75      }
76
77      // Push an integer literal value.
78      void intlit(INTLIT n) {
79          add(IR.PUSH);
80          add(n.val);
81      }
82
83      // Generate code for a binary arithmetic expression. First
84      // generate code that pushes the value of the left hand
85      // and right hand sides on the stack. Then generate an 'ADD'
86      // instruction which pops these two values, adds them together,
87      // and pushes the result.
88      void binop(BINOP n) {
89         expr(n.left);
90         expr(n.right);
91         if (n.OP == Token.PLUS)
92             add(IR.ADD);
93         else if (n.OP == Token.STAR)
94             add(IR.MUL);
95      }
96
97      public void write () {
98        IR.write(code,pc);
99      }
100
101     public static void main (String args[]) throws Exception{
102        Lex scanner = new Lex(args[0]);
103        Parse parser = new Parse(scanner);
104        Sem sem = new Sem(parser.ast);
105        GenIR ir = new GenIR(sem);
106        ir.write();
107     }
108 }
109
110
```

25 Interpreter.java

```
1  /* Copyright 2001, Christian Collberg, collberg@cs.arizona.edu. */
2
3  import java.lang.*;
4  import java.io.*;
5
6  public class Interpreter {
7
8     // Evaluation stack.
9     static int[] stack = new int[100];
10    static int sp = 0;
11    static void push (int v) {stack[sp++] = v;}
12    static int pop() {return stack[--sp]; }
13
14   static void run (int[] prog) throws Exception {
15       int[] memory=null;
16       int pc = 0;
17       while (true) {
18       switch (prog[pc]) {
19       case IR.HEADER    : {
20          if (prog[pc+1]!=IR.MAGIC) {
21              System.err.println("Wrong magic number.");
22              throw new Exception();
23          }
24          memory = new int[prog[pc+2]];
25          pc+=3; break;
26       }
27       case IR.ADD    : {
28          int right = pop(); int left  = pop(); push(left+right); pc++; break;
29       }
30       case IR.MUL    : {
31          int right = pop(); int left  = pop(); push(left*right); pc++; break;
32       }
33       case IR.LOAD   : {
34          push(memory[(int)prog[pc+1]]); pc+=2; break;
35       }
36       case IR.STORE  : {
37          memory[prog[pc+1]] = pop(); pc+=2; break;
38       }
39       case IR.PUSH  : {
40          push(prog[pc+1]); pc+=2; break;
41       }
42       case IR.PRINT  : {
43          System.out.print(pop()); pc++; break;
44       }
45       case IR.PRINTLN: {
46          System.out.println(); pc++; break;
47       }
48       case IR.EXIT   : {
49         return;
50       }
51       default :
52               System.err.println("Illegal instruction: " + prog[pc]);
53               throw new Exception();
54         }
55       }
56    }
57
58     public static void main(String args[]) throws Exception{
59         int[] code = IR.read(args[0]);
60         run(code);
61     }
```

```
62  }
```

---

26 GenMips.java

```
1   /* Copyright 2001, Christian Collberg, collberg@cs.arizona.edu. */
2
3   import java.lang.*;
4   import java.io.*;
5
6   public class GenMips {
7       int[] prog;
8
9       public GenMips(int[] prog) {
10         this.prog = prog;
11         translate();
12      }
13
14     // Collection of free registers.
15     static String[] regs = {"$s0","$s1","$s2","$s3","$s4","$s5","$s6",
16                             "$t0","$t1","$t2","$t3","$t4","$t5","$t6"};
17     int nextReg = 0;
18
19     // Return all used registers, and start allocating them from scratch.
20     void initRegs() {
21        nextReg = 0;
22     }
23
24      // Return the next free register.
25     String freeReg() {
26        return regs[nextReg++];
27     }
28
29     // Register stack.
30     String[] stack = new String[100];
31     int sp = 0;
32     void push (String v) {stack[sp++] = v;}
33     String pop() {return stack[--sp]; }
34
35     // The generated assembly code is stored in a string 'code'.
36     public String code = "";
37     void add(String instr) {
38        code += instr + "\n";
39     }
40
41     void translate() {
42        int pc = 0;
43        initRegs();
44        while (true) {
45        switch (prog[pc]) {
46           case IR.HEADER    : {
47              add("\t.data");
48              add("newline:\t.asciiz \"\\n\"");
49              int vars = prog[pc+2];
50              for(int i=0; i<vars; i++)
51                  add("var" + i + ":\t\t.word 0");
52              pc+=3;
53              add("\t.text");
54              add("\t.align 2");
55              add("\t.globl main");
56              add("main:");
57              break;
58           }
59           case IR.ADD    : {
60              String right = pop();             // The register holding the left hand side.
61              String left  = pop();             // The register holding the right hand side.
```

```
62              String res   = freeReg();          // The register to hold the result.
63              add("\tadd\t" + res + "," + left + "," + right);
64              push(res);
65              pc++; break;
66           }
67           case IR.MUL    : {
68              String right = pop();              // The register holding the left hand side.
69              String left  = pop();              // The register holding the right hand side.
70              String res   = freeReg();          // The register to hold the result.
71              add("\tmul\t" + res + "," + left + "," + right);
72              push(res);
73              pc++; break;
74           }
75           case IR.LOAD   : {
76              String id = "var" + prog[pc+1];    // The variable identifier.
77              String reg = freeReg();            // The register to hold variable value.
78              push(reg);
79              add("\tlw\t" + reg + "," + id);    // Load the variable into the register.
80              pc+=2; break;
81           }
82           case IR.STORE  : {
83              String id = "var" + prog[pc+1];    // The variable identifier.
84              String reg = pop();                // The register holding the variable value.
85              add("\tsw\t" + reg + "," + id);    // Store the register value into the variable.
86              pc+=2;
87              initRegs();
88              break;
89           }
90           case IR.PUSH  : {
91              int val = prog[pc+1];              // The integer literal value.
92              String res   = freeReg();          // The register to hold the result.
93              add("\tli\t" + res + "," + val);   // Load the integer into the register.
94              push(res);
95              pc+=2; break;
96           }
97           case IR.PRINT  : {
98              String reg = pop();                // The register holding the expression value.
99              add("\tmove\t$a0," + reg);         // Move into $a0.
100             add("\tli\t$v0,1");
101             add("\tsyscall");                  // Print the value.
102             pc++;
103             initRegs();
104             break;
105          }
106          case IR.PRINTLN: {
107             add("\tla\t$a0,newline");
108             add("\tli\t$v0,4");
109             add("\tsyscall");
110             pc++; break;
111          }
112          case IR.EXIT   : {
113             add("\tli\t$v0,10");
114             add("\tsyscall");
115             return;
116          }
117          default :
118        }
119     }
120  }
121
122   public static void main(String args[]) throws Exception{
123      int[] code = IR.read(args[0]);
124      GenMips mips = new GenMips(code);
125      System.out.println(mips.code);
126   }
```

```
127     }
```

---

27 Compiler.java

```
1  /* Copyright 2001, Christian Collberg, collberg@cs.arizona.edu. */
2
3  import java.io.*;
4  import java.util.*;
5
6  public class Compiler {
7
8      // The first argument should be '-ir' or '-mips'.
9      // The second argument should be the name of the source
10     // code file.
11     public static void main (String args[]) throws IOException {
12         String what = args[0];
13         String file = args[1];
14         Lex scanner = new Lex(file);
15         Parse parser = new Parse(scanner);
16         Sem sem = new Sem(parser.ast);
17         Opt opt = new Opt(sem);
18         GenIR ir = new GenIR(opt.sem);
19
20         if (what.equals("-ir"))
21            ir.write();
22         else if (what.equals("-mips")) {
23            GenMips mips = new GenMips(ir.code);
24            System.out.println(mips.code);
25         }
26     }
27
28 }
```