CSc 453

Compilers and Systems Software

2 : Teensy Simple I

Department of Computer Science
University of Arizona

collberg@gmail.com

Copyright © 2009 Christian Collberg

# Introduction

## A simple language and compiler

- To understand the overall organization of a modern compiler, we will design a minimal language, TEENSY, and a compiler for this language, SIMPLE.
- Get the source from the file Simple1.zip from the class website. Then `unzip Simple1.zip`; `cd PROGRAMS`; `make`; `make test`.
- Here's a simple TEENSY program:

```
BEGIN
    x = 5; y = 99; z = y + x + 9; PRINT z;
END
```

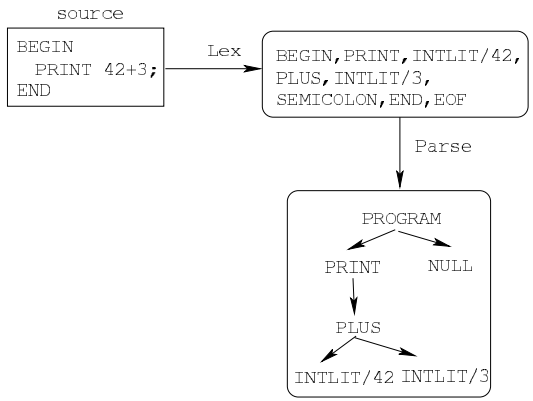
## TEENSY's concrete grammar

| | | |
|---|---|---|
| program | → | 'BEGIN' stats 'END' |
| stats | → | stat stats \| $\epsilon$ |
| stat | → | ident '=' expr ';' |
| | \| | 'PRINT' expr ';' |
| expr | → | expr '+' expr \| ident \| int |
| ident | → | LETTER idp |
| idp | → | LETTER idp \| DIGIT idp \| $\epsilon$ |
| int | → | DIGIT intp |
| intp | → | DIGIT intp \| $\epsilon$ |

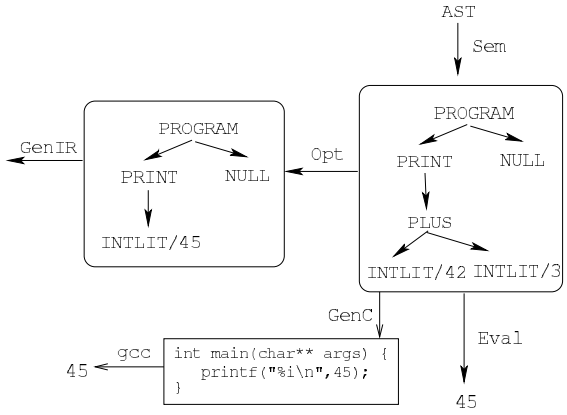
## Java classes in the SIMPLE compiler

- 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`),
- intermediate code generation (`IR`, `GenIR`),
- stack-code interpretation (`Interpreter`), and
- machine-code generation (`Mips`).
- The class `Compiler` ties it all together.

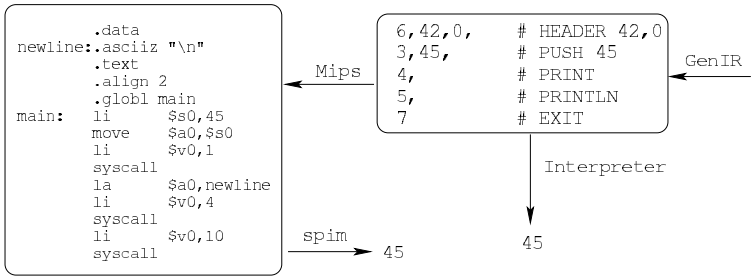
## Overview – Lexing and Parsing




## Overview – Semantics & Optimization




## Overview – IR & Code Generation



## Token.java

```
public class Token {
    public final static int ILLEGAL    = 0;
    public final static int PLUS       = 1;
    public final static int INTLIT     = 2;
    public final static int IDENT      = 3;
    public final static int SEMICOLON  = 4;
    public final static int EQUAL      = 5;
    public final static int BEGIN      = 6;
    public final static int END        = 7;
    public final static int PRINT      = 8;
    public final static int EOF        = 9;

    public int kind;
    public String ident; public int value;
    public int position;
}
```

## Lex.java

```
char ch;                // lookahead character
boolean done = false;   // reached end-of-file

public Lex(String filename) throws IOException {
    str = new LineNumberReader(new FileReader(filename));
    get();
}

// read the next input character
void get() {...}

Token scanNumber() {
    ...
    return new Token(Token.INTLIT, ival);
}
```

## Lex.java...

```
Token scanName()  {
    String ident = "";
    while ((!done) && Character.isLetterOrDigit(ch)) {
        ident+=ch; get();}
    if (ident.equals("BEGIN"))
         return new Token(Token.BEGIN);
    else if (ident.equals("END"))
         return new Token(Token.END);
    else if (ident.equals("PRINT"))
         return new Token(Token.PRINT);
    else
         return new Token(Token.IDENT, ident);
}
```

## Lex.java...

```
public Token nextToken() {
    while ((!done) && ch <= ' ') get();
    if (done) return new Token(Token.EOF);
    switch (ch) {
        case '+': get(); return new Token(Token.PLUS);
        case ';': get(); return new Token(Token.SEMICOLON);
        case '=': get(); return new Token(Token.EQUAL);
        default:  if (Character.isLetter(ch))
                      return scanName();
                  else if (Character.isDigit(ch))
                      return scanNumber();
                  else {
                      get(); return new Token(Token.ILLEGAL
                   }
     }
}
```

## Parsing

The Lexer takes an input TEENSY source file

```
BEGIN
   PRINT y;
END
```

and generates a stream of tokens

```
BEGIN
PRINT
IDENT: y
SEMICOLON
END
EOF
```

which is then consumed by the parser.

## Parsing

- The parser does two things:
  1. it checks that the input program conforms to the syntax of the language. If not, error messages are generated.
  2. it generates an abstract syntax tree (AST), a tree representation of the input program.
- SIMPLE supports two parsers: `Matcher` only tests for syntax conformance, `Parse` also generates the AST.
- The AST forms the basis for all further processing in SIMPLE.

## Matcher.java...

```
Lex scanner;
Token currentToken;

public Matcher (Lex scanner) {
   this.scanner = scanner;
   next();
}

void next() {
   currentToken = scanner.nextToken();
}

boolean lookahead(int tokenKind) {
   return currentToken.kind == tokenKind;
}
```

## Matcher.java...

```
void match(int tokenKind) {
   if (!lookahead(tokenKind)) {
        System.err.println("Parsing error");
    }
    next();
}

public void parse() {
   ENTER("parse");
   match(Token.BEGIN);
   stats();
   match(Token.END);
   match(Token.EOF);
   EXIT("parse");
}
```

## Matcher.java...

```
void stats() {
   if (lookahead(Token.IDENT)) {
      assign();  match(Token.SEMICOLON); stats();
   } else if (lookahead(Token.PRINT)) {
       print();  match(Token.SEMICOLON); stats();
   }
}

void assign() {
   match(Token.IDENT); match(Token.EQUAL); expr();
}

void print() {
   match(Token.PRINT); expr();
}
```

## Matcher.java...

```
void expr() {
   factor();
   while (lookahead(Token.PLUS)) {
      match(Token.PLUS);
      factor();
   }
}

void factor() {
   if (lookahead(Token.IDENT)) {
       match(Token.IDENT);
   } else if (lookahead(Token.INTLIT)) {
       match(Token.INTLIT);
   }
}
```

## Parsing...

```
> cat test23
BEGIN
   PRINT y;
END
```

```
> java Matcher test2
ENTER parse
   ENTER stats
      ENTER print
         ENTER expr
            ENTER factor
            EXIT factor
         EXIT expr
      EXIT print
      ENTER stats
      EXIT stats
   EXIT stats
EXIT parse
```

## Concrete vs. Abstract Syntax

- A language's concrete syntax describes how it is written by the programmer – where whitspace goes, where semicolons are needed, etc.
- The abstract syntax describes the "logical structure" of the language – that while-statements have two parts (the expresion and the loop body), that procedure calls consist of a sequence of actual parameters (expression) and the name of the called procedure, etc.
- The abstract syntax also defines the nodes of the abstract syntax tree. For example, an AST while-node would have two children, one pointing to the expression subtree, the other to the loop body subtree.

## Teensy's Abstract Syntax

| | | |
|---|---|---|
| PROGRAM | → | STATSEQ |
| STATSEQ | → | STAT STATSEQ |
| | \| | NULL |
| STAT | → | ASSIGN |
| | \| | PRINT |
| ASSIGN | → | ident EXPR |
| PRINT | → | EXPR |
| EXPR | → | BINOP \| IDENT \| INTLIT |
| BINOP | → | op EXPR EXPR |
| IDENT | → | ident |
| INTLIT | → | int |

## AST, EXPR, INTLIT

```
public abstract class AST {}
```

```
public abstract class EXPR extends AST {}
```

```
public class INTLIT extends EXPR {
    public int val;
    public INTLIT(int val) {this.val = val;}
}
```

```
public class IDENT extends EXPR {
    public String ident;
    public IDENT(String ident) {
       this.ident = ident;}
}
```

## BINOP.java

```
public class BINOP extends EXPR {
   public int OP;
   public EXPR left, right;

   public BINOP(int OP, EXPR left, EXPR right) {
      this.OP=OP; this.left=left;
                  this.right=right;}

   public String toString() {
      String op = (OP == Token.PLUS)?"+":"";
      return "("+op+", "+left.toString()+", "+
             right.toString()+")";}
}
```

## Building the AST – Parse.java

```
AST parse() {
    match(Token.BEGIN);
    STATSEQ s = stats();
    match(Token.END); match(Token.EOF);
    return new PROGRAM(s);
}

STATSEQ stats() {
   STAT stat;
   if (lookahead(Token.IDENT))       stat = assign();
   else if (lookahead(Token.PRINT)) stat = print();
   else                              return new NULL();
   match(Token.SEMICOLON);
   return new STATSEQ(stat, stats());
}
```

## Parse.java...

```
STAT assign() {
    String ident = currentToken.ident;
    match(Token.IDENT);  match(Token.EQUAL);
    return new ASSIGN(ident, expr());
}

EXPR expr() {
   EXPR f = factor();
   while (lookahead(Token.PLUS)) {
      match(Token.PLUS);
      f = new BINOP(Token.PLUS, f, factor());
   }
   return f;
}
```

## Parse.java...

```
EXPR factor() {
    EXPR expr = null;
    if (lookahead(Token.IDENT)) {
       expr = new IDENT(currentToken.ident);
       match(Token.IDENT);
    } else if (lookahead(Token.INTLIT)) {
       expr = new INTLIT(currentToken.value);
       match(Token.INTLIT);
    }
    return expr;
}
```

## Parsing...

```
> cat test4
BEGIN
   x = 5;
   y = 99;
   z = y + x + 9;
   PRINT z;
END
> java Parse test4
PROGRAM
   (ASSIGN x, (INTLIT 5))
   (ASSIGN y, (INTLIT 99))
   (ASSIGN z, (+, (+, (IDENT y), (IDENT x)), (INTLIT 9)))
   (PRINT (IDENT z))
   NULL
```

## Semantic Analysis

- The only possible semantic error in TEENSY is a variable being used before it's first defined.
- `Sem.java` walks the AST, and inserts any identifiers found on the left hand side of an assignment statement in the symbol table.
- The symbol table is defined in `SyTab.java`.
- Each name inserted into the table is mapped to a number.
- Notice how similar the code is in `Eval.java`, `Sem.java`, `Opt.java`, and `GenIR.java`. They all do recursive tree walks over the abstract syntax tree.

## SyTab.java – Symbol Table

```
Hashtable sytab = new Hashtable();
int currentID = 0;

public void insert(String ident) {
   if (!sytab.containsKey(ident))
      sytab.put(ident, new java.lang.Integer(currentID++));
}

public int lookup(String ident) {
   if (sytab.containsKey(ident))
      return ((Integer)sytab.get(ident)).intValue();
   else  return -1;
}

public int size() {return sytab.size();}
```

## Sem.java – Semantic Analysis

```
public SyTab sytab = new SyTab();

void program(PROGRAM n) {stats(n.stats);}

void stats(STATSEQ n) {
   if (n instanceof NULL) return;
   stat(n.stat);  stats(n.next);
}

void stat(STAT n) {
   if (n instanceof ASSIGN)     assign((ASSIGN)n);
   else if (n instanceof PRINT) print((PRINT)n);
}
```

## Sem.java – Semantic Analysis...

```
void assign(ASSIGN n) {sytab.insert(n.ident);expr(n.expr);}
void print(PRINT n) {expr(n.expr);}

void expr(EXPR n) {
   if (n instanceof IDENT)        ident((IDENT) n);
   else if (n instanceof INTLIT)  intlit((INTLIT) n);
   else if (n instanceof BINOP)   binop((BINOP) n);
}

void ident(IDENT n) {
   if (sytab.lookup(n.ident)<0) println("Undeclared!");
}

void intlit(INTLIT n) {}
void binop(BINOP n) {expr(n.left); expr(n.right);}
```

## Eval.java – AST Evaluation

```
int[] memory;        // Variable store.

void program(PROGRAM n) {
   memory = new int[sem.sytab.size()]; stats(n.stats);
}

void stats(STATSEQ n) {
   if (!(n instanceof NULL)) {stat(n.stat); stats(n.next)}
}

void stat(STAT n) {
   if (n instanceof ASSIGN)     assign((ASSIGN)n);
   else if (n instanceof PRINT) print((PRINT)n);
}
```

## Eval.java...

```
void assign(ASSIGN n) {
    memory[sem.sytab.lookup(n.ident)] = expr(n.expr);
}

void print(PRINT n) {
   System.out.println(expr(n.expr));
}

int expr(EXPR n) {
   if (n instanceof IDENT)       return ident((IDENT) n);
   else if (n instanceof INTLIT) return intlit((INTLIT) n);
   else if (n instanceof BINOP)  return binop((BINOP) n);
   return -1;
}
```

## Eval.java...

```
int ident(IDENT n) {
    return memory[sem.sytab.lookup(n.ident)];
}

int intlit(INTLIT n) {
   return n.val;
}

int binop(BINOP n) {
  if (n.OP == Token.PLUS)
      return expr(n.left) + expr(n.right);
  return -1;
}
```

## IR Generation

- SIMPLE uses a straight-forward stack-based intermediate representation. There are only 8 bytecodes.
- IR.java defines the opcodes.
- GenIR walks the AST and emits code. The code is simply an array of integers.

## IR.java – IR Generation

```
public static final int ADD    = 0;
public static final int LOAD   = 1;
public static final int STORE  = 2;
public static final int PUSH   = 3;
public static final int PRINT  = 4;
public static final int PRINTLN= 5;
public static final int HEADER = 6;
public static final int EXIT   = 7;

public static final int MAGIC  = 42;

public static int[] read(String filename) {...}
public static void write(int code[], int pc) {...}
```

## IR Opcode Semantics

| mnemonic | Op | 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 |

## GenIR.java

```
int pc = 0;
public int[] code = new int[100];
void add(int instr) {code[pc++] = instr;}

void program(PROGRAM n) {
   add(IR.HEADER);  add(IR.MAGIC); add(sem.sytab.size());
   stats(n.stats);
   add(IR.EXIT);
}

void stats(STATSEQ n) {
   if (n instanceof NULL) return;
   stat(n.stat);
   stats(n.next);
}
```

## GenIR.java...

```
void stat(STAT n) {
   if (n instanceof ASSIGN)     assign((ASSIGN)n);
   else if (n instanceof PRINT)  print((PRINT)n);
}

void assign(ASSIGN n) {
    expr(n.expr);
    add(IR.STORE);
    add(sem.sytab.lookup(n.ident));
}

void expr(EXPR n) {
   if      (n instanceof IDENT)    ident((IDENT) n);
   else if (n instanceof INTLIT)   intlit((INTLIT) n);
   else if (n instanceof BINOP)    binop((BINOP) n);
}
```

## GenIR.java...

```
void ident(IDENT n) {
    add(IR.LOAD); add(sem.sytab.lookup(n.ident));
}

void intlit(INTLIT n) {
    add(IR.PUSH); add(n.val);
}

void binop(BINOP n) {
   expr(n.left); expr(n.right);
   if (n.OP == Token.PLUS) add(IR.ADD);
}
```

## Interpreting the IR – Interpreter.java

```
static void run (int[] prog){
   int[] memory=null; int pc = 0;
   while (true) {
   switch (prog[pc]) {
      case IR.HEADER    : {
         if (prog[pc+1]!=IR.MAGIC) {...}
         memory = new int[prog[pc+2]]; pc+=3; break;
      }
      case IR.ADD     : {
        int right=pop(); int left=pop();
        push(left+right); pc++; break;
      }
      case IR.LOAD    : {
         push(memory[(int)prog[pc+1]]); pc+=2; break;
      }
```

## Interpreting the IR – Interpreter.java...

```
      case IR.STORE   : {
         memory[prog[pc+1]] = pop(); pc+=2; break;
      }
      case IR.PUSH    : {
         push(prog[pc+1]); pc+=2; break;
      }
      case IR.PRINT   : {
         System.out.print(pop()); pc++; break;
      }
      case IR.PRINTLN: {
         System.out.println(); pc++; break;
      }
      case IR.EXIT    : return;
      default :
}}}
```

```
> cat test7
BEGIN
   PRINT 42+3;
END
```

```
> java GenIR test7 | tee test7.vm
6
42
0
3
42
3
3
0
4
5
7
> java Interpreter test7.vm
45
```

## Code generation

- The code generator "simulates" the execution of the IR code, but instead of operating on a stack of computed values (the way `Interpreter.java` has), we have a stack of register names. These are the registers into which the current subexpressions have been computed.
- `freeReg` returns the next available register. We don't attempt to handle the case when we run out of registers, nor do we attempt to minimize the number of registers used.

## Code generation – Mips.java

```
// Collection of free registers.
static String[] regs = {"$s0","$s1","$s2","$s3",...};
int nextReg = 0;

void initRegs() {nextReg = 0;}
String freeReg() {return regs[nextReg++]; }

// Register stack.
String[] stack = new String[100];
int sp = 0;
void push (String v) {stack[sp++] = v;}
String pop() {return stack[--sp]; }
```

## Code generation – Mips.java

```
public String code = "";
void add(String instr) {code += instr + "\n";}

void translate() {
   int pc = 0; initRegs();
   while (true) {
   switch (prog[pc]) {
      case IR.HEADER    : {
         add(".data"); add("newline:.asciiz \"\\n\"");
         int vars = prog[pc+2];
         for(int i=0; i<vars; i++)
            add("var"+i + ":.word 0");
         pc+=3;
         add(".text "); add(".align 2");
         add(".globl main");
         add("main:"); break;
```

## Code generation – Mips.java

```
case IR.ADD    : {
    String right = pop();
    String left  = pop();
    String res   = freeReg();
    add("add " + res + "," + left + "," + right);
    push(res);
    pc++; break;
 }
 case IR.LOAD   : {
    String id = "var" + prog[pc+1];
    String reg = freeReg();
    push(reg);
    add("lw " + reg + "," + id);
    pc+=2; break;
 }
```

## Code generation – Mips.java

```
case IR.STORE  : {
   String id = "var" + prog[pc+1];
   String reg = pop();
   add("sw " + reg + "," + id);
   pc+=2;
   initRegs();
   break;
}
case IR.PUSH   : {
   int val = prog[pc+1];
   String res   = freeReg();
   add("li " + res + "," + val);
   push(res);
   pc+=2; break;
}
```

## Code generation – Mips.java

```
        case IR.PRINT  : {
           String reg = pop();
           add("move $a0," + reg);
           add("li   $v0,1");
           add("syscall");
           pc++;
           initRegs();
           break;
        }
        case IR.PRINTLN: {
           add("la $a0,newline");
           add("li $v0,4");
           add("syscall");
           pc++; break;
        }
```

## Code generation – Mips.java

```
        case IR.EXIT   : {
           add("li $v0,10");
           add("syscall");
           return;
        }
        default :
      }
    }
  }
}
```

```
> java Mips test7.vm | tee test7.s
        .data
newline:.asciiz "\verb+\+n"
        .text
        .align 2
        .globl main
main:   li      $s0,42
        li      $s1,3
        add     $s2,$s0,$s1
        move    $a0,$s2
        li      $v0,1
        syscall
        la      $a0,newline
        li      $v0,4
        syscall
        li      $v0,10
        syscall
```

## Putting it all together – Compiler.java

```
public static void main (String args[]) throws IOException
    Lex scanner = new Lex(args[1]);
    Parse parser = new Parse(scanner);
    Sem sem = new Sem(parser.ast);
    Opt opt = new Opt(sem);
    GenIR ir = new GenIR(opt.sem);

    if ( args[0].equals("-ir"))
       ir.write();
    else if (what.equals("-mips")) {
       Mips mips = new Mips(ir.code);
       System.out.println(mips.code);
    }
}
```

## Putting it all together...

```
java Lex test1                # Print the tokens
java Matcher test2            # Print the ``parse tree''
java Parse test2              # Print the AST and syntax er
java Sem test4                # Print semantic error messag
java Eval test4               # Evaluate the AST
java GenIR test4 > test4.vm   # Generate IR code
java Interpreter test4.vm     # Interpret the IR code
java Opt test5                # Optimize the AST
java Mips 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
```

## Readings and References

- Read Louden:
  Introduction pp. 1–18, 21–27.
  Lexical Analysis pp. 31–34.
  Syntactic Analysis pp. 95–100.
  Recursive Descent Parsing pp. 143–151.
  Semantic Analysis pp. 257–259.
  Code Generation pp. 397–399.
- or the Dragon Book:
  A Simple Compiler pp. 25–82