Compilation 2012

Hand-Written One-Pass Compilers

Jan Midtgaard
Michael I. Schwartzbach
Aarhus University

Compiler Technology

- A Joos compiler uses big technology:
  - scanner and parser generators
  - 8 different tree types
  - 14 passes through the AST
  - ML type inference
- Even a Joos 0 compiler requires:
  - 4,324 lines of hand-written code
  - 2,533 lines of auto-generated code
- This is orthogonal to the conceptual complexity:
  - scanner, parser, weeder, scopes, environments, static type checking, static analysis, code templates, optimization

Light-Weight Technology

- A one-pass (or narrow) compiler:
  - reads the source file one character at a time
  - constructs no internal representation of the full program
  - outputs the generated code simultaneously
- A hand-written compiler:
  - contains no auto-generated code
- Benefits of light-weight technology:
  - simple, fast, and fun
  - suitable for JIT compiling
- Downside of light-weight technology:
  - Not obvious how to scale to complex languages like Java (designed for multi-pass compilation)

Limitations of One-Pass Technology

- Limited scope rules:
  - we can't see anything that occurs later in the file
- Lack of static analysis:
  - we never get a complete picture of the program
- Lack of optimization:
  - we can't look at the generated code twice

The Original One-Pass Language

- The Pascal language (1970):
  - a hand-written, one-pass compiler
  - implemented in 4000 lines of Pascal
  - simplicity of the compiler was a major design criteria
- The light-weight tradition continued to:
  - countless Pascal dialects (at least 15 languages)
  - the Modula-2 language (1978)
  - the Oberon language (1988)

One-Pass Scopes in Pascal

- Pascal introduced `forward` declarations:

```
procedure foo(x: alpha; var y: integer); forward;
```

  to permit mutually recursive procedures

- Also, (as yet) unknown pointers were allowed:

```
type List = ^Item;
type Item = record
              head: integer;
              tail: List;
            end;
```

  to permit recursive data structures

An Example Language: C0

- The C0 language is a simple subset of C
- Limitations:
  - only integer types
  - control structures `if`, `else`, `while`, `return`
  - operators `+`, `*`, `/`, `-`, `%`, `!`, `&`, `|`, `==`, `!=`, `<`, `>`, `<=`, `>=`
  - I/O only with `putchar` and `getchar`
- Includes function prototypes (similar to `forward`)

- Every C0 program can also be compiled by `gcc`
- We shall hand-write a one-pass compiler

Context-Free Grammar for C0

```
program  → header function* main                main → main() body
header   → #include <stdio.h>
function → int id ( formals? ) body | int id ( formals? ) ;
formals  → formal | formal , formals
formal   → int id
body     → { decl* stm* }
decl     → int id init? ;
init     → = const
stm      → id = exp; | while ( exp ) { stm* } | putchar ( exp ) ; |
           return exp ; | if ( exp ) { stm* } |
           if ( exp ) { stm* } else { stm* }
exp      → const | id | - exp | ! exp | id ( actuals? ) |
           exp op exp | getchar() | ( exp )
actuals  → exp | exp , actuals
const    → intconst | 'char' | '\n'
op       → + | - | * | / | % | & | | | == | != | < | > | <= | >=
```

Example C0 Program

```
#include <stdio.h>

int writedigits(int n) {
  int w;
  if (n!=0) {
    w = writedigits(n/10);
    putchar('0'+n%10);
  }
  return 0;
}

int writeint(int n) {
  int w;
  if (n==0) {
    putchar('0');
  } else {
    if (n<0) {
      putchar('-');
      n = -n;
    }
    w = writedigits(n);
  }
  return 0;
}
```

```
int gcd(int x, int y) {
  while ( x != y ) {
    if (x < y) { y = y - x; }
    else { x = x - y; }
  }
  return x;
}

main() {
  int w;
  w = writeint(gcd(15,35));
}
```

The IJVM Target Architecture

```
program   → method+
method    → .method symbol directive* insn+
directive → .args expr | .locals expr | .define symbol = expr
insn      → bipush expr | dup | goto symbol | iadd |
             iand | ifeq symbol  | iflt symbol | if_icmpeq symbol |
             iinc  expr , expr | iload expr | invokevirtual symbol |
             ior | ireturn | istore expr | isub | ldc_w expr |
             nop | pop | swap | symbol :
expr      → integer | symbol | expr + expr | expr – expr | ( expr )
```

Example IJVM Code

```
.method writedigits
.args 2
.locals 1
iload 1
bipush 44
swap
ldc_w 0
invokevirtual ne_
ifeq L0
bipush 44
iload 1
bipush 44
swap
ldc_w 10
invokevirtual div_
invokevirtual writedigits
istore 2
bipush 44
ldc_w 48
iload 1
bipush 44
swap
ldc_w 10
invokevirtual mod_
iadd
invokevirtual putchar
goto L1
L0:
L1:
```

```
ldc_w 0
ireturn
bipush 0
ireturn
.method writeint
.args 2
.locals 1
iload 1
bipush 44
swap
ldc_w 0
invokevirtual eq_
ifeq L2
bipush 44
ldc_w 48
invokevirtual putchar
goto L3
L2:
iload 1
bipush 44
swap
ldc_w 0
invokevirtual lt_
ifeq L4
bipush 44
ldc_w 45
invokevirtual putchar
bipush 0
iload 1
```

```
isub
istore 1
goto L5
L4:
L5:
bipush 44
iload 1
invokevirtual writedigits
istore 2
L3:
ldc_w 0
ireturn
bipush 0
ireturn
.method gcd
.args 3
L6:
iload 1
bipush 44
swap
iload 2
invokevirtual ne_
ifeq L7
iload 1
bipush 44
swap
iload 2
invokevirtual lt_
ifeq L8
```

```
iload 2
iload 1
isub
istore 2
goto L9
L8:
iload 1
iload 2
isub
istore 1
L9:
goto L6
L7:
iload 1
ireturn
bipush 0
ireturn
.method main
.args 1
.locals 1
bipush 44
bipush 44
ldc_w 15
ldc_w 35
invokevirtual gcd
invokevirtual writeint
istore 1
bipush 0
ireturn
```

Defining Tokens

```
static final int tLPAR = 0;
static final int tRPAR = 1;
static final int tASSIGN = 2;
static final int tSEMI = 3;
static final int tCOMMA = 4;
static final int tEQ = 5;
static final int tNE = 6;
static final int tID = 7;
static final int tCONST = 8;
static final int tCHAR = 9;
static final int tADD = 10;
...
static String tFile;       // source file name
static int tLine;          // current line
static int tCol;           // current column
static int tIntValue;      // value if tCONST
static String tIdValue;    // value if tID
static int tKind;          // current token kind
```

A Hand-Written Scanner

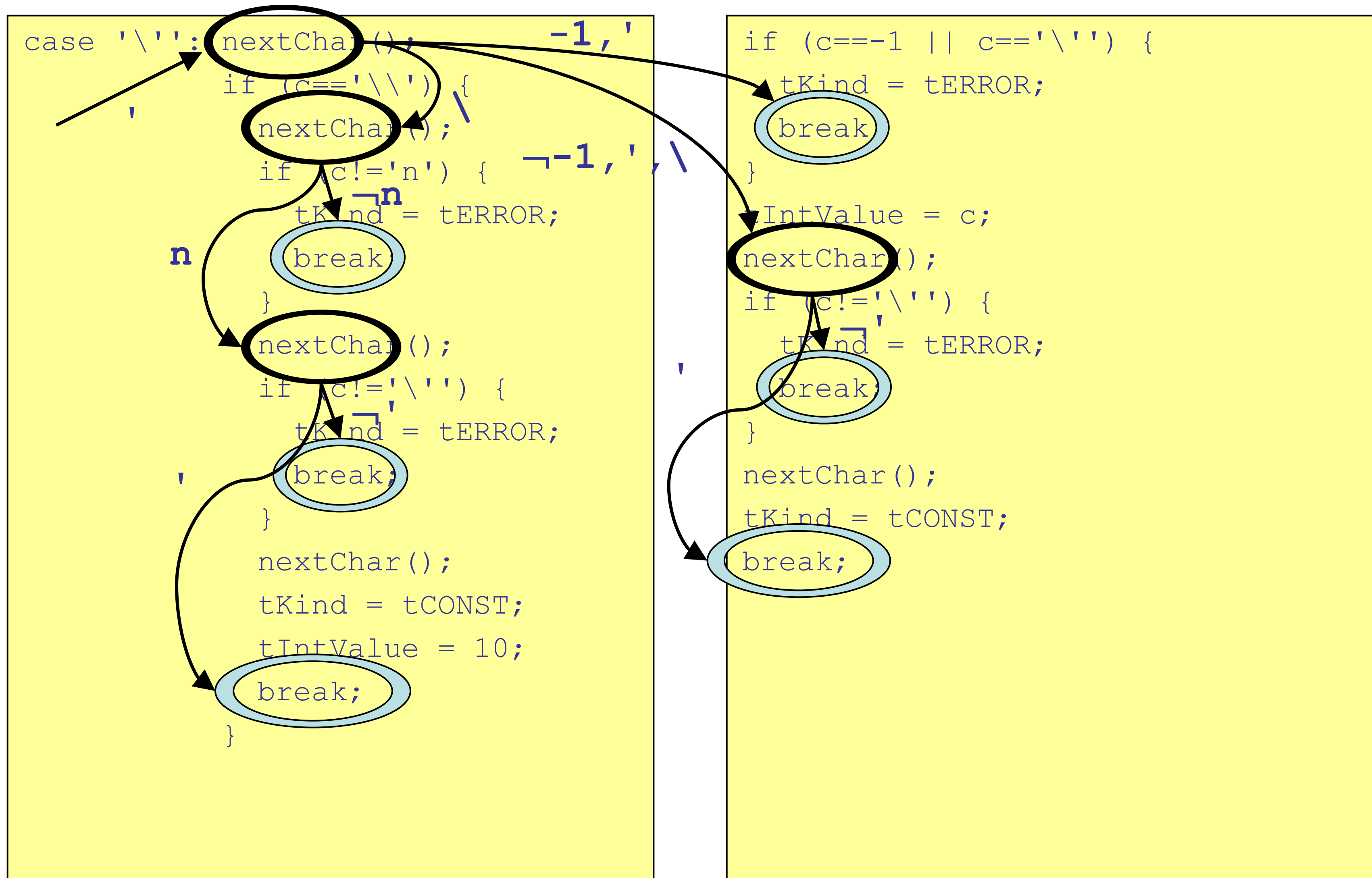
```
static int c;                        // current char

static int nextChar() {              // read next char from the source file
  try {
    c = in.read();
  } catch (Exception e) {
    c = -1;
  }
  if (c=='\n') {
    tLine++;
    tCol = 1;
  } else tCol++;
  return c;
}
...
static int nextToken() {           // recognize next token
  switch (c) {
    ...
  }
}
```

Embedding a DFA in Java (1/3)

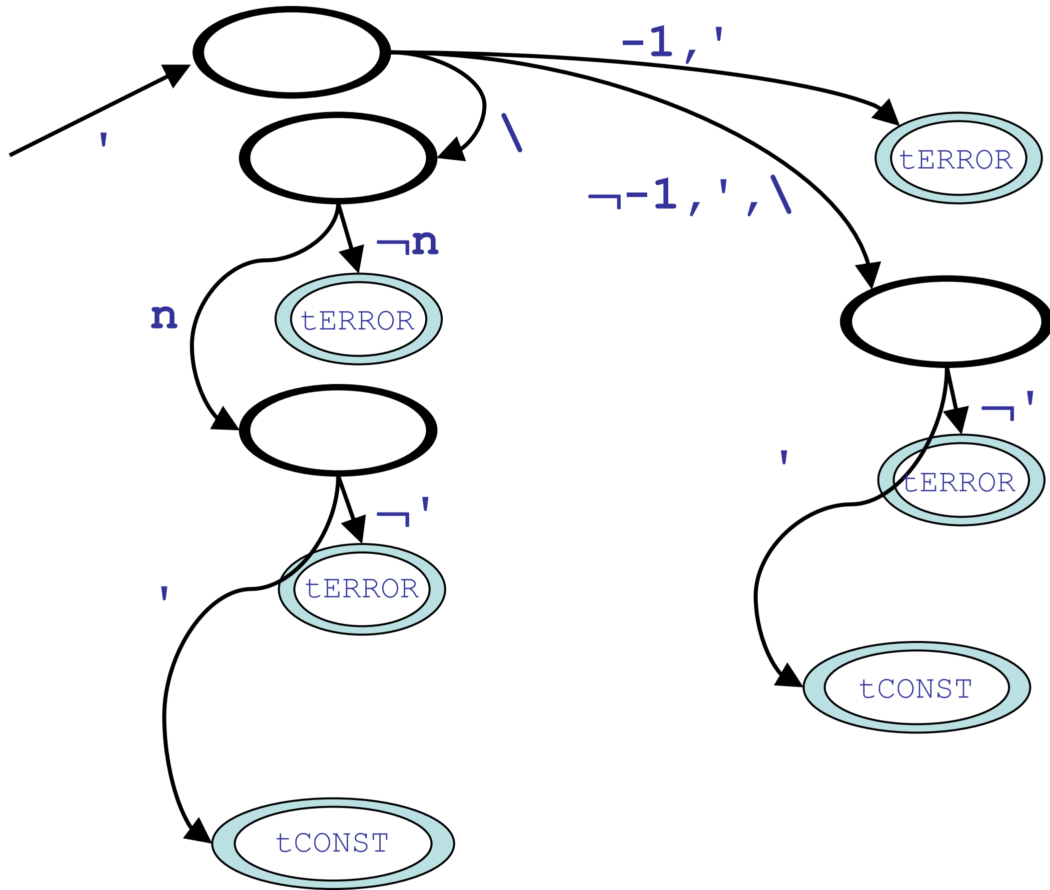
```
case '\'': nextChar();
           if (c=='\\') {
             nextChar();
             if (c!='n') {
               tKind = tERROR;
               break;
             }
             nextChar();
             if (c!='\'') {
               tKind = tERROR;
               break;
             }
             nextChar();
             tKind = tCONST;
             tIntValue = 10;
             break;
           }
```

```
if (c==-1 || c=='\'') {
  tKind = tERROR;
  break;
}
tIntValue = c;
nextChar();
if (c!='\'') {
  tKind = tERROR;
  break;
}
nextChar();
tKind = tCONST;
break;
```

Embedding a DFA in Java (2/3)



Embedding a DFA in Java (3/3)



Identifiers, Keywords, and Integers

```
if (letter(c)) {
  tKind = tID;
  tIdValue = "";
  while (alpha(c)) {
    tIdValue = tIdValue+(char)c;
    nextChar();
  }
  if (tIdValue.equals("int")) tKind = tINT;
  else if (tIdValue.equals("if")) tKind = tIF;
  else if (tIdValue.equals("else")) tKind = tELSE;
  else if (tIdValue.equals("while")) tKind = tWHILE;
  else if (tIdValue.equals("getchar")) tKind = tGETCHAR;
  else if (tIdValue.equals("putchar")) tKind = tPUTCHAR;
  else if (tIdValue.equals("include")) tKind = tINCLUDE;
  else if (tIdValue.equals("return")) tKind = tRETURN;
  else if (tIdValue.equals("main")) tKind = tMAIN;
} else if (digit(c)) {
  tKind = tCONST;
  tIntValue = 0;
  while (digit(c)) {
    tIntValue = 10*tIntValue+c-'0';
    nextChar();
  }
} else tKind = tERROR;
```

Recursive Descent Parsing

- Each nonterminal and its productions:

  $$A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_k$$

  are turned into a method:

```
void parseA() {
  if (tKind ∈ predict(α1)) parse(α1);
  else if (tKind ∈ predict(α2)) parse(α2);
  ...
  else if (tKind ∈ predict(αk)) parse(αk);
}
```

LL(1) Grammars

- For $\alpha \in (V \cup \Sigma)^*$ and $A \in V$ we define:
  - $\mathit{first}(\alpha) = \{ a \in \Sigma \mid \exists \beta: \alpha \Rightarrow^* a\beta \}$
  - $\mathit{follow}(A) = \{ a \in \Sigma \mid \exists \beta_1, \beta_2: S \Rightarrow^* \beta_1 A a \beta_2 \}$
- For the production $X \rightarrow \alpha$ we define:
  - $\mathit{predict}(\alpha) = \mathit{first}(\alpha) \cup \mathit{follow}(X)$ if $\alpha \Rightarrow^* \Lambda$

    $= \mathit{first}(\alpha)$ otherwise
- A grammar is LL(1) iff for every pair:

  $X \rightarrow \alpha_i \mid X \rightarrow \alpha_j$

  we have that:

  $\mathit{predict}(\alpha_i) \cap \mathit{predict}(\alpha_j) = \emptyset$
- For LL(1) grammars, recursive descent works

Parsing Right-Hand Sides

- The function *parse*(α) is defined for each symbol
  - for a$\in\Sigma$: invoke the method `checkToken(a);`
  - for A$\in$V: invoke the method `parseA();`

where we have:

```
static void checkToken(int k) {
  if (tKind!=k) parseError();
  nextToken();
}

static void checkToken(String s) {
  if (tKind!=tID || !tIdValue.equals(s)) parseError();
  nextToken();
}
```

Ignoring Whitespace

- Whitespace is handled by invocations of:

```
static void skip() {
  while (tKind==tWHITE) nextToken();
}

static void skipToken(int k) {
  skip();
  checkToken(k);
}
```

A Hand-Written Parser

```
static void parseStatement() {
  if (tKind==tID) {
    nextToken();
    skipToken(tASSIGN);
    skip();
    parseExpression();
    skipToken(tSEMI);
  } else if (tKind==tRETURN) {
    nextToken();
    skip();
    parseExpression();
    skipToken(tSEMI);
```

```
  } else if (tKind==tIF) {
    nextToken();
    skipToken(tLPAR);
    skip();
    parseExpression();
    skipToken(tRPAR);
    skipToken(tLBRACK);
    parseStatements();
    skipToken(tRBRACK);
    skip();
    if (tKind==tELSE) {
      nextToken();
      skipToken(tLBRACK);
      skip();
      parseStatements();
      skipToken(tRBRACK);
    }
  }
```

```
  } else if (tKind==tWHILE) {
    nextToken();
    skipToken(tLPAR);
    skip();
    parseExpression();
    skipToken(tRPAR);
    skipToken(tLBRACK);
    skip();
    parseStatements();
    skipToken(tRBRACK);
  } else if (tKind==tPUTCHAR) {
    nextToken();
    skipToken(tLPAR);
    skip();
    parseExpression();
    skipToken(tRPAR);
    skipToken(tSEMI);
  } else parseError();
}
```

Parsing EBNF Grammars

- X*:

```
while (tKind ∈ predict(X)) parseX();
```

- X+:

```
parseX();
while (tKind ∈ predict(X)) parseX();
```

- X?:

```
if (tKind ∈ predict(X)) parseX();
```

Parsing Expressions (1/2)

- Consider a general operator hierarchy:
  - Unary (e.g. `-`, `!`)
  - Binary_1 (e.g. `*`, `/`, `%`, `&`)
  - Binary_2 (e.g. `+`, `-`, `|`)
  - Binary_3 (e.g. `==`, `!=`, `<`, `>`, `<=`, `>=`)
  - ...

- Also, all binary operators must be left-associative:

  `2+3+4+5+6` ≡ `((((2+3)+4)+5)+6)`

Parsing Expressions (2/2)

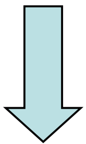
```
void parseExp0() {
  ...
  if (tKind ∈ Unary) {
    ...
    parseExp0();
  }
}

void parseExp1() {
  parseExp0();
  while (tKind ∈ Binary1) {
    ...
    parseExp0();
  }
}
```

```
void parseExp2() {
  parseExp1();
  while (tKind ∈ Binary2) {
    ...
    parseExp1();
  }
}

void parseExp3() {
  parseExp2();
  while (tKind ∈ Binary3) {
    ...
    parseExp2();
  }
}

...
```

Action Code

- So far, the parser decides language membership
- The code for the remaining phases must be mixed into the right-hand sides:

```
parseX(); checkToken(a); parseY(); parseZ();
```



```
parseX(); CODE checktoken(a); CODE parseY(); CODE parseZ();
```

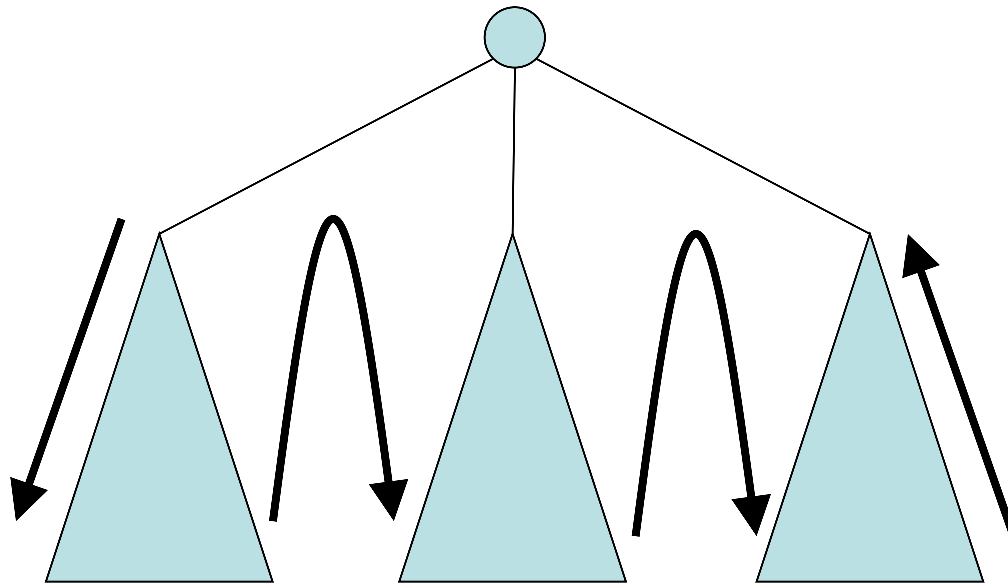
One-Pass Static Checking

- Only having integer types simplifies the checks:
  - Used variables must be declared
  - Used functions must be prototyped or implemented
  - Function calls must have the correct arguments
  - A function must not be prototyped twice
  - A function must not be implemented twice
  - A prototype and an implementation must not conflict
  - All prototypes must be implemented

- This must all be checked during parsing

L-Attributed Traversals

- The parser offers a single pre-order traversal
- Attributes can be carried along from left to right:



Attributes for Static Checking

- The traversal uses these attributes:
  - `Map funcs`
    a map from names to number of arguments
  - `Map prototypes`
    a map from names to number of arguments
  - `Map vars`
    a map from variables to offsets
  - `int offset`
    the offset of the next declared variable

Parsing and Static Checking (1/3)

```
static void parseFunction(Map funcs, Map prototypes) {
  Map vars = new HashMap();
  String name;
  checkToken(tINT); skipToken(tID);
  name = tIdValue;
  skipToken(tLPAR);
  int args = parseFormals(vars);
  skipToken(tRPAR); skip();
  if (tKind==tSEMI) {
    nextToken();
    if (prototypes.containsKey(name))
      compileError("duplicate declaration of "+name);
    if (funcs.containsKey(name) && args!=funcs.get(name))
      compileError("conflicting declaration of "+name);
    prototypes.put(name,args);
  } else {
    if (funcs.containsKey(name))
      compileError("duplicate implementation of "+name);
    if (prototypes.containsKey(name) && args!=prototypes.get(name))
      compileError("conflicting implementation of "+name);
    funcs.put(name,args);
    parseBody(args,vars,funcs,prototypes);
  }
}
```

Parsing and Static Checking (2/3)

```
static void parseExp0(Map vars, Map funcs, Map prototypes) {
  ...
  } else if (tKind==tID) {
    String name = tIdValue;
    nextToken(); skip();
    if (tKind==tLPAR) {
      nextToken(); skip();
      int args = parseActuals(vars,funcs,prototypes);
      skipToken(tRPAR);
      if (funcs.containsKey(name)) {
        if (args!=funcs.get(name))
          compileError("incorrect number of arguments: "+name);
      } else if (prototypes.containsKey(name)) {
        if (args!=prototypes.get(name))
          compileError("incorrect number of arguments: "+name);
      } else {
        compileError("undeclared function: "+name);
      }
    } else {
      if (!vars.containsKey(name))
        compileError("undeclared variable: "+name);
    }
  ...
}
```

Parsing and Static Checking (3/3)

```
static void parseProgram() {
   Map funcs = new HashMap();
   Map prototypes = new HashMap();
   skipToken(tSHARP);
   checkToken(tINCLUDE);
   skipToken(tLT);
   checkToken("stdio");
   checkToken(tDOT);
   checkToken("h");
   checkToken(tGT);
   parseFunctions(funcs,prototypes);
   parseMain(funcs,prototypes);
   Iterator i = prototypes.keySet().iterator();
   while (i.hasNext()) {
     String s = (String)i.next();
     if (!funcs.containsKey(s))
       compileError("missing implementation of "+s);
   }
 }
```

One-Pass Code Templates (1/2)

- Code must be generated on-the-fly
- This places some limits on the possible templates

- The syntax for `if` is a prefix of that for `if-else`

  $\Downarrow$

  The code for `if` is a prefix of that for `if-else`

- This imposes some superfluous jumps for `if`

One-Pass Code Templates (2/2)

```
} else if (tKind==tIF) {
  int elselabel = nextLabel++;
  int donelabel = nextLabel++;
  nextToken();
  skipToken(tLPAR);
  skip();
  parseExpression(vars,funcs,prototypes);
  skipToken(tRPAR);
  code("ifeq L"+elselabel);
  skipToken(tLBRACK);
  parseStatements(vars,funcs,prototypes);
  skipToken(tRBRACK);
  code("goto L"+donelabel);
  code("L"+elselabel+":");
  skip();
  if (tKind==tELSE) {
    nextToken();
    skipToken(tLBRACK);
    skip();
    parseStatements(vars,funcs,prototypes);
    skipToken(tRBRACK);
  }
  code("L"+donelabel+":");
```

Parsing, Checking, and Generating Code

```
} else if (tKind==tID) {
     String name = tIdValue;
     nextToken(); skip();
     if (tKind==tLPAR) {
       nextToken(); skip();
       code("bipush 44");
       int args = parseActuals(vars,funcs,prototypes);
       code("invokevirtual "+name);
       skipToken(tRPAR);
       if (funcs.containsKey(name)) {
         if (args!=funcs.get(name))
           compileError("incorrect number of arguments: "+name);
       } else if (prototypes.containsKey(name)) {
         if (args!=prototypes.get(name))
           compileError("incorrect number of arguments: "+name);
       } else {
         compileError("undeclared function: "+name);
       }
     } else {
       if (!vars.containsKey(name))
         compileError("undeclared variable: "+name);
       code("iload "+vars.get(name));
     }
```

Exploiting the Recursion Stack

- A code template may demand later information
- Example:
  - first the number of locals must be emitted
  - then the initialization code must be emitted
  - but we don't know the number of locals until afterward
- Several solutions:
  - backpatching: insert dummy into code, patch the dummy later once correct value is known
  - use the recursion stack to visit the "AST" twice (on the way down + on the way back through the recursion)
  - …

Generating Declaration Code

```
static void parseDeclarations(int offset, int locals, Map vars) {
   int initValue = 0;
   boolean initialized = false;
   skip();
   if (tKind==tINT) {
     nextToken(); skipToken(tID);
     offset++;
     vars.put(tIdValue,offset);
     skip();
     if (tKind==tASSIGN) {
       nextToken(); skipToken(tCONST);
       initialized = true;
       initValue = tIntValue;
     }
     checkToken(tSEMI);
     parseDeclarations(offset,locals+1,vars);
   } else {
     if (locals>0) code(".locals "+locals);
   }
   if (initialized) {
     code("bipush "+initValue);
     code("istore "+offset);
   }
 }
```

first visit

second visit

IJVM Library Code

- The IJVM does not directly support the operators:

  `*, /, %, ==, !=, <, >, <=, >=`

- To handle these, we generate a library:

```
.method mul_
.method div_
.method mod_
.method eq_
.method ne_
.method lt_
.method gt_
.method le_
.methid ge_
```

The Complete C0 Compiler

- Light-weight technology:
  - 628 lines of hand-written Java code
  - 213 lines of IJVM library code
  - 0 lines of auto-generated code
- A small but interesting subset of C
  - enough to handle the dComNet examples