Review

- Regular expressions to NFA
- NFA to DFA
- Nullable, first, and follow
- LL(1) parsing
- LR(0), SLR, LR(1), LALR(1) parsing
More Generally

- Definition of formal language, regular expression
- Recursive descent parsers
- Scanners versus recognizers
- Definition of grammars, parse trees, ambiguity
- Hierarch of formal languages
Overview of Assignment 5

• Study AST for MiniJava
• Add to JavaCC parser semantic actions to create AST
• Understand visitor pattern
• Design symbol table
• Code visitor to create symbol table
• Code visitor to perform semantic checking.
Chapter 4: Abstract Syntax
source program

scanner & parser

abstract syntax

semantic analysis

abstract syntax

IR code generation

intermediate code

compiler backend

---

all lexical and syntax errors

all type and semantic errors

401
Chapter 4: Abstract Syntax

4.1: Semantic Actions
Semantic Actions

A compiler must do more than recognize whether a sentence belongs to the language of a grammar. The semantic actions of a parser can do useful things with the phrases that are parsed.

Appel, 2nd, page 86.
Semantic Actions

In a recursive-descent parser, semantic action code is interspersed with the control flow of the parsing actions. In a parser specified in JavaCC, semantic actions are fragments of Java program code attached to grammar productions. SableCC, on the other hand, automatically generates syntax trees as it parses.

Appel, 2nd, page 86.

For JavaCC, there are several companion tools, including JJTree and JTB (the Java Tree Builder), which generate syntax tree classes and insert action code into the grammar for building syntax trees.

Appel, 2nd, page 89.
Recursive Descent

A recursive-descent parser can act as an interpreter if we add semantic actions that return values.

See Appel, 2nd, Program 4.1, page 87.
Recursive Descent


0  \( S \rightarrow E \, $ \)
1  \( E \rightarrow T \, E' \)
2  \( E' \rightarrow + \, T \, E' \)
3  \( E' \rightarrow \epsilon \)
4  \( T \rightarrow F \, T' \)
5  \( T' \rightarrow * \, F \, T' \)
6  \( T' \rightarrow \epsilon \)
7  \( F \rightarrow \text{id} \)
8  \( F \rightarrow ( \, E \, ) \)
class Token { int kind; Object val;
    Token(int k, Object v) {kind=k; val=v;}
}

final int EOF=0, ID=1, NUM=2, PLUS=3, MINUS=4, ...

int lookup(String id) { ... }

int F_follow[] = { PLUS, TIMES, RPAREN, EOF };

int F() { switch (tok.kind) {
    case ID:  int i=lookup((String)(tok.val)); advance(); return i;
    case NUM: int i=((Integer)(tok.val)).intValue();
              advance(); return i;
    case LPAREN: eat(LPAREN);
                int i = R();
                eatOrSkipTo(RPAREN, F_follow);
                return i;
    case EOF: print("expected ID, NUM, or left-paren");
                skipTo(F_follow); return 0;
    default: print("expected ID, NUM, or left-paren");
                skipTo(F_follow); return 0;
}

int T_follow[] = { PLUS, RPAREN, EOF };

int T() { switch (tok.kind) {
    case ID:  case NUM:
    case LPAREN: return Tprime(F());
    default: print("expected ID, NUM, or left-paren");
                skipTo(T_follow);
                return 0;
}

int Tprime(int a) { switch (tok.kind) {
    case TIMES: eat(TIMES); return Tprime(a*F());
    case PLUS:
    case RPAREN:
    case EOF: return a;
}
Had the production been $T \rightarrow T \ast F$, then the semantic action would have been

```c
int a = T(); eat(TIIMES); int b=F(); return a*b;
```

But the artificial syntactic category $T'$ is tricky. The production $T' \rightarrow \ast F T'$ is missing the left operand of $\ast$. So we pass the left operand as an argument to $T'$, as shown in Program 4.1.

```c
int Tprime (int a) //...
    eat(TIIMES); return Tprime (a \ast F());
```
A Parser specified with JavaCC

```java
void ExpressionList() :
{
  {}
  {
    Expression() ( ExpressionListRest() )* 
  }
}

void ExpressionList (final List<Expression> el) :
{
  final Expression e;
}
{
  e = Expression () {el.add (e);} ( ExpressionListRest(el) )* 
}
```
Chapter 4: Abstract Syntax

4.2: Abstract Parse Trees
Abstract Syntax Data Structures

Many early compilers did not use an abstract syntax data structure because early computers did not have enough memory to represent an entire compilations unit’s syntax tree. Modern computers rarely have this problem. And many modern programming languages (ML, Modula-3, Java) allow forward reference to identifiers defined later in the same module; using an abstract syntax tree makes compilation easier for these languages. It may be that Pascal and C require clumsy forward declarations because their designers wanted to avoid an extra compiler pass on the machines of the 1970s.

A Parser specified with CUP

terminal PLUS, MINUS, TIMES, UMINUS, INT;
non terminal exp;
start with exp;

precedence left PLUS, MINUS;
precedence left TIMES;
precedence left UMINUS;

exp ::= INT
  | exp PLUS exp
  | exp MINUS exp
  | exp TIMES exp
  | MINUS exp % prec UMINUS
;

412
Question

1. What do the precedence declarations in a CUP specification do?

2. Is there something similar in JavaCC?
Semantic Actions

terminal PLUS, MINUS, TIMES, UMINUS;
terminal Integer INT;
non terminal Integer exp;
start with exp;

precedence left PLUS, MINUS;
precedence left TIMES;
precedence left UMINUS;

exp ::= INT:i { : RESULT = i; : }
| exp:x PLUS exp:y { : RESULT=new Integer(x.intValue()+y.intValue()); : }
| exp:x MINUS exp:y { : RESULT=new Integer(x.intValue()-y.intValue()); : }
| exp:x TIMES exp:y { : RESULT=new Integer(x.intValue()*y.intValue()); : }
| MINUS exp:e % prec UMINUS { : RESULT=new Integer(-e.intValue()); : }
;
Semantic Actions

How do semantic actions work? We keep a parallel stack.
## Semantic Actions

<table>
<thead>
<tr>
<th>step</th>
<th>stack</th>
<th>input</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>$1 + 2 * 3$</td>
<td>shift 3</td>
</tr>
<tr>
<td>1</td>
<td>1 3</td>
<td>num +2 * 3$</td>
<td>$E \rightarrow$ num</td>
</tr>
<tr>
<td>2</td>
<td>1 5</td>
<td>$E +2 * 3$</td>
<td>shift 8</td>
</tr>
<tr>
<td>3</td>
<td>1 5 8</td>
<td>$E + 2 * 3$</td>
<td>shift 3</td>
</tr>
<tr>
<td>4</td>
<td>1 5 8 3</td>
<td>$E +$ num *3$</td>
<td></td>
</tr>
</tbody>
</table>
## Semantic Actions

<table>
<thead>
<tr>
<th>step</th>
<th>stack</th>
<th>input</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1 5 8</td>
<td>$E + 2 \times 3$</td>
<td>shift 3</td>
</tr>
<tr>
<td>4</td>
<td>1 5 8 3</td>
<td>$E + \text{num} \times 3$</td>
<td>$E \rightarrow \text{num}$</td>
</tr>
<tr>
<td>5</td>
<td>1 5 8 5</td>
<td>$E + E \times 3$</td>
<td>shift 12</td>
</tr>
<tr>
<td>6</td>
<td>1 5 8 5 12</td>
<td>$E + E \times 3$</td>
<td>shift 3</td>
</tr>
<tr>
<td>7</td>
<td>1 5 8 5 12 3</td>
<td>$E + E \times \text{num}$</td>
<td></td>
</tr>
</tbody>
</table>
# Semantic Actions

<table>
<thead>
<tr>
<th>step</th>
<th>stack</th>
<th>input</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>1 5 8 5 12</td>
<td>$ E + E * 3$</td>
<td>shift 3</td>
</tr>
<tr>
<td></td>
<td>$ E$</td>
<td>1 2</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>1 5 8 5 12 3</td>
<td>$ E + E * num $</td>
<td>$E \rightarrow num$</td>
</tr>
<tr>
<td></td>
<td>$ E$</td>
<td>1 2 3</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1 5 8 5 12 13</td>
<td>$ E + E * E $</td>
<td>$E \rightarrow E * E$</td>
</tr>
<tr>
<td></td>
<td>$ E$</td>
<td>1 2 3</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>1 5 8 9</td>
<td>$ E + E $</td>
<td>$E \rightarrow E + E$</td>
</tr>
<tr>
<td></td>
<td>$ E$</td>
<td>1 6</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1 5</td>
<td>$ E $</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$ E$</td>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>
Semantic Actions

An LR parser does perform reductions, and associated semantic actions, in a deterministic and predictable order; a bottom-up, left-to-right traversal of the parse tree. In other words, the (virtual) parse tree is traversed in postorder. Thus, one can write imperative semantics actions with global side effects, and be able to predict the order of their occurrence.


As always, it is better not to employ side effects.
Why Parse Trees?

It is possible to write an entire compiler that fits within the semantics actions phrases of a JavaCC or SableCC parser. However, such a compiler is difficult to read and maintain, and this approach constrains the compiler to analyze the program in exactly the order it is parsed.

Appel, 2nd, Section 4.2, page 89.

Today there is enough computer memory to fit a program’s entire parse tree.

We use JavaCC to construct an abstract parse tree and go on from there.
Abstract Syntax Trees

Technically, a parse tree has exactly one leaf for each token of the input and one internal node for each grammar rule reduced during the parse.

Such a tree is inconvenient as it has too much redundant information. It holds all the punctuation; it depends too much on the quirks of the grammar. We call it a concrete parse tree because it represents the concrete syntax of the source language.

An abstract syntax tree makes a clean interface between the parser and the later phases of the compiler.
Abstract Syntax Tree

Concrete parse trees may have a lot of extra stuff and are inconvenient to use directly.

\[
\begin{align*}
\text{assign} & \rightarrow \text{id} := \text{expr} \\
\text{expr} & \rightarrow \text{id} \ast \text{expr} \\
\text{expr} & \rightarrow \text{id} + \text{expr} \\
\text{expr} & \rightarrow \text{id} \\
\text{expr} & \rightarrow ( \text{expr} )
\end{align*}
\]
Example Derivation

assign ⇒

\[
\begin{align*}
id & := expr \Rightarrow \\
A & := expr \Rightarrow \\
A & := id \ast expr \Rightarrow \\
A & := B \ast expr \Rightarrow \\
A & := B \ast (expr) \Rightarrow \\
A & := B \ast (id + expr) \Rightarrow \\
A & := B \ast (A + expr) \Rightarrow \\
A & := B \ast (A + id) \Rightarrow \\
A & := B \ast (A + C)
\end{align*}
\]
Concrete Parse Tree

```
assign
  id := exp
    exp
      *
        id
          exp
            ( id + exp
              id
                A
              id
                C
            )
          exp
            id
              A
            id
              C
```
Abstract Syntax Tree

\[
A := B \ast (A + C)
\]

The nonterminal `expr` can be replaced by the kind of expression \(+\), \(*\), etc. The nonterminal `id` contains no information. Parentheses are no longer needed.
If $A:=(B*A)+C$ was the statement, then the abstract syntax would be different and so no important information would be lost.
Concrete Syntax For MiniJava

http://www.cs.fit.edu/ ryan/cse4251/mini_java_grammar.html
Abstract Syntax For MiniJava

Where are the positions?

Perhaps an abstract class parent of all the abstract syntax classes has it.
Abstract Syntax For MiniJava

Significant changes from Appel: Line number and column position, java.util.List, Java constants for primitive types.
package syntax;

abstract class AST () // line and column number

Program (MainClass m, List<ClassDecl> cl)
MainClass (Identifier i, Identifier j, Statement s)

abstract class ClassDecl

VarDecl (Type t, Identifier i)
MethoDecl (Type t, Identifier i, List<Formal> fl, List<VarDecl> vl,
          List<Statement> sl, Exp e)
Formal (Type t, Identifier i)

abstract class Type

abstract class Statement
abstract class Exp

Identifier (String s)
Abstract Syntax For MiniJava (continued)

abstract class ClassDecl
ClassDeclSimple (Identifier i, List<VarDec> vl, List<MethodDecl> ml)
ClassDeclExtends (Identifier i, Identifier j, /* ... */)

abstract class Type
IntArrayType() BooleanType () IntegerType() IdentifierType (String

abstract class Statement
Block (List<Statement> sl)
If (Exp e, Statement s1, Statement s2)
While (Exp e, Statement s)
Print (Exp e)
Assign (Identifier i, Exp e)
ArrayAssign (Identifier i, Exp e1, Exp e2)
abstract class Exp
And, LessThan, Plus, Minus, Times (Exp e1, Exp e2)
ArrayLookUp (Exp e1, Exp e3)
ArrayLength (Exp e3)
Call (Exp e, Identifier i, ExpList e1)
IntegerLiteral (int i)
True()
False()
IdentifierExp (String s)
This()
NewArray (Exp e)
NewObject (Identifier i)
Not (Exp e)
Abstract Syntax For MiniJava

\[ x = y.m(1, 4+5) \]

new Assign(
    new Identifiers("x"),
    new Call(
        new IdentifierExp("y"),
        new Identifier("m"),
        Arrays.asList(
            new ArrayLiteral(1),
            new Plus(
                new IntegerLiteral(4),
                new IntegerLiteral(5)
            )
        )
    )
);
Design Patterns

The Gang of Four defines the *visitor* as:

Represent an operation to be performed on elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.
Object-Oriented

Each *interpretation* must be applied to each *kind*; if we add a new kind, we must implement each interpretation for it; and if we add a new interpretation, we must implement it for each kind.

Appel, 2nd, Figure 4.6, page 94. Kinds versus Interpretations, orthogonal axes of modularity.
## Interpretations

<table>
<thead>
<tr>
<th>Kinds</th>
<th>Type-check</th>
<th>Translate to Pentium</th>
<th>Translate to Sparc</th>
<th>Find uninitialized vars</th>
<th>Optimize</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>IdExp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NumExp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PlusExp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MinusExp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TimesExp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SeqExp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(a) Compiler

<table>
<thead>
<tr>
<th>Kinds</th>
<th>Redisplay</th>
<th>Move</th>
<th>Iconize</th>
<th>Deiconize</th>
<th>Highlight</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scrollbar</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Menu</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Canvas</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DialogBox</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Text</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>StatusBar</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(b) Graphic user interface
Visitor

With the object-oreinted style, each interpretation is just a method in all the classes. It is easy and modular to add a new kind: All the interpretations of that kind are grouped together as methods of the new class. But it is not modular to add a new interpretation: A new method must be added to every class.

Appel, 2nd, page 95-96.
Visitor

In category theory, the concept of catamorphism (from Greek: \( \kappa \alpha \tau \acute{\alpha} \) = downwards or according to; \( \mu \omega \rho \phi \) = form or shape) denotes the unique homomorphism from an initial algebra into some other algebra.

In functional programming, catamorphisms provide generalizations of folds of lists to arbitrary algebraic data types, which can be described as initial algebras. Consider the Haskell class **Foldable**.
Compiler Project

Add semantic actions to your parser to produce abstract syntax for the Mini-Java language.
Chapter 5: Semantic Analysis

5.1. Symbol Tables
Scope entry/exit

declare
  A:Integer:=0; B:Integer:=0; C:Integer:=0;
begin
  declare
    A:Integer:=1; B:Integer:=1; C:Integer:=1;
  begin
    declare
      C:Integer:=2; D:Integer:=2;
    begin
      C := A + D;
    end;
    A := B + C;
  end;
declarer
  A:Integer:=3; B:Integer:=3;
begin
  A := B + C;
end;
begin scope
decl a, decl b, decl c
begin scope
decl a, decl b, decl c
begin scope
decl c, decl d
lookup c, a, d
end scope
lookup a, b, c
end scope
begin scope
decl a, decl b
lookup a, b, c
end scope
end scope
end scope
Scope entry/exit
E: declare
    A:Integer:=0; B:Integer:=0; C:Integer:=0;
begin -- E
F: declare
    A:Integer:=1; B:Integer:=1; C:Integer:=1;
begin
    A := B + C;
end F;
G: declare
    A:Integer:=2; B:Integer:=2;
begin
    H: declare
        C:Integer:=3; D:Integer:=3;
    begin
        C := A + D;
    end I;
end I;
I: declare
    B:Integer:=4; D:Integer:=4;
begin
    B := A + C;
end I;
B := A + C;
end G;
J: declare
    B:Integer:=5; D:Integer:=5;
begin scope G
decl a, decl b, decl c
begin scope F
decl a, decl b, decl c
lookup a, b, c
end scope F
begin scope G
decl a, decl b
begin scope H
decl c, decl d
lookup c, a, d
end scope H
begin scope I
decl b, decl d
lookup b, a, c
end scope I
lookup b, a, c
end G
begin scope J
decl b, d
lookup b, a, d
end scope
lookup a, b, c
end scope
let
  var a:=0 var b:=0 var c:=0
in ( 
  let
    var a:=1 var b:=1 var c:=1
  in ( 
    let 
      var c:=2 var d:=2
    in 
      c := a + d 
      a := b + c 
    end; 
    a := b + c 
  ) 
  end; 
let 
  var a:=3 var b:=3 
in 
  a := b + c 
end 
) 
end 
begin scope 
decl a, decl b, decl c 
begin scope 
decl a, decl b, decl c 
begin scope 
decl c, decl d 
lookup c, a, d 
end scope 
lookup a, b, c 
end scope 
begin scope 
decl a, decl b 
lookup a, b, c 
end scope 
end scope
Hash Table Review

Collision resolution.

- *chaining* treats each entry as a collection or bucket of values instead of as one value.

- *open address method* looks for an available position in the table other than the one to which the element is originally hashed.
  1. linear probing: $H + 1, H + 2, H + 3, \ldots$
  2. quadradric probing $H + 1^2, H + 2^2, H + 3^2, \ldots$
  3. double hashing
Symbol Table

The entry in the symbol table keeps track of any hidden entries of the same name, and the stack of entries pushed in the current scope.

```java
public class Binder<Value> {
    public final Value value;
    public final String nextInLevel;
    public final Binder<Value> prevBind;
    Binder (Value v, String p, Binder<Value> t) {
        value=v; nextInLevel=p; prevBind=t;
    }
}
```
Symbol Table

See Table.java
Persistent Binary Trees

But binary trees are not $O(1)$!

Hash tables can’t be implemented efficiently without mutable state, because they’re based on array lookup. The key is hashed and the hash determines the index into an array of buckets. Without mutable state, inserting elements into the hashtable becomes $O(n)$ because the entire array must be copied (alternative non-copying implementations

Binary-tree implementations can share most of their structure so only a couple pointers need to be copied on inserts.

Haskell certainly can support traditional hash tables, provided that the updates are in a suitable monad. The hashtables package is probably the most widely used implementation.

One advantage of binary trees and other non-mutating structures is that they’re persistent: it’s possible to keep older copies of data around with no extra book-keeping. This is useful in compiler symbol tables.
After the initial `begin_scope`.
Declare a.
persistent search tree

fun insert (x, E) = T (E, x, E)
  | insert (x, s as T (a, y, b)) =
    if x < y then T (insert (x, a), y, b)
    else if x > y then T (a, y, insert (x, b))
    else s
Chapter 5: Semantic Analysis

5.1. Symbol Tables
String interning is a method of storing only one copy of each distinct string value, which must be immutable. Interning strings makes some string processing tasks more time- or space-efficient at the cost of requiring more time when the string is created or interned. The distinct values are stored in a string intern pool. A single copy of each string is called its “intern.”

In Java, all compile-time constant strings in Java are automatically interned. Lisp, Scheme, and Smalltalk are among the languages with a symbol type that are basically interned strings.

Objects other than strings can be interned. For example, in Java, when primitive values are boxed into a wrapper object, certain values (any boolean, any byte, any char from 0 to 127, and any short or int between -128 and 127) are interned.
String interning speeds up string comparisons, which are sometimes a performance bottleneck in applications such as compilers that rely heavily on hash tables with string keys. Without interning, checking that two different strings are equal involves examining every character of both strings. This is slow for several reasons: it is inherently $O(n)$ in the length of the strings; it typically requires reads from several regions of memory, which take time; and the reads fill up the processor cache, meaning there is less cache available for other needs. With interned strings, a simple object identity test suffices; this is typically implemented as a pointer equality test, normally just a single machine instruction with no memory reference at all.
Chapter 5: Semantic Analysis

5.2. Type-Checking Minijava
Symbol Table

The environment contains a symbol table for variables, methods, and classes

- To each variable name (and formal parameter) is bound its type;
- To each method name is bound its parameters result type, and local variables;
- To each class is bound its variable and method declarations.

See Appel, 2nd, Figure 5.7, page 111.
Symbol Table

```
class B {
    C f;  int [] j;  int q;
    public int start(int p, int q) {
        int ret;   int a;
        /* ... */
        return ret;
    }
    public boolean stop(int p) {
        /* ... */
        return false;
    }
}

class C {
    /* ... */
}
```

**FIGURE 5.7.** A Minijava Program and its symbol table
Type-checking of a MiniJava program proceeds in two phases. First, we build the symbol table, and then we type-check the statements and expressions. During the second phase, the symbol table is consulted for each identifier that is found. It is convenient to use two phases because, in Java and MiniJava, the classes are mutually recursive. If we tried to do type-checking in a single phase, then we might need to type-check a call to a method that is not yet entered into the symbol table.

See Appel, 2nd, page 112.
Visit – Build

public Void visit (VarDecl n) {
    // could be local variable or field of class
    table.addVarDecl (n.i.s, n.t);
    return null;
}
public Type visit (Plus n) {
    if (n.e1.accept(this) != Type.THE_INTEGER_TYPE) {
        System.err.println("Left operand of '+' must be of type 'int'.");
    } else if (n.e2.accept(this) != Type.THE_INTEGER_TYPE) {
        System.err.println("Right operand of '+' must be of type 'int'.");
    }
    return Type.THE_INTEGER_TYPE;
}
Semantic Errors

A semantic error in MiniJava.

class Factorial {
    public static void main (String[] args) {
        System.out.println (new Fac().fact (10));
    }
}

class Fac {
    public int fact (int num) {
        System.out.println (XXXX);
    }
}

The identifier XXXX is undeclared.
Semantic Errors

A semantic error in MiniJava.

class Factorial {
    public static void main (String[] args) {
        System.out.println (new XXXX());
    }
}

The class identifier XXXX is undefined.
Semantic Errors for MiniJava

E.java:011.013: Undeclared identifier 'num_mispelled' in 'Fac.ComputeFac'.
E.java:011.013: First argument of '<' does not have type integer.
E.java:003.032: The class 'Fac_mispelled' is not declared anywhere in the program.
E.java:011.034: Call to method 'Fac.ComputeFac' has the wrong number of arguments.
E.java:000.000: Second argument of '*' does not have type integer.
E.java:007.019: Undeclared super class 'XXXX'
E.java:007.007: Class 'Fac' and super class 'Fac' in circular class hierarchy.
E.java:011.021: Condition of 'if' not of type boolean.
E.java:000.000: Argument 1 of 1 in call to method 'Fac.ComputeFac' has the wrong type.
  arg has type: boolean, formal has type int
E.java:061.020: Incompatible types in assignment: LHS='int', RHS='boolean'.
E.java:090.016: Incompatible types of return; declared type 'boolean', found type 'int'.
E.java:000.000: 'length' for array only; found type 'BS'.
E.java:000.000: Argument 1 of 3 in call to method 'Bob.carrol' has the wrong type.
  arg has type: int, formal has type int[]
E.java:000.000: Argument 3 of 3 in call to method 'Bob.carrol' has the wrong type.
  arg has type: int[], formal has type int
E.java:000.000: Method 'alice' not found in class 'Bob'
Semantic Errors

A semantic error in MiniJava.

class Factorial {
    public static void main (String[] args) {
        System.out.println (args);
    }
}

We might as well make the argument argument to main undeclared.
Overloading

Overloading of methods will not be tested.
Overview of Assignment 6

• Review the visitor pattern
• Review AST for MiniJava
• Study the give IR code
• Fix the design symbol table
• Detect missing semantics errors
• Study chapter 6
• Advise: use a sparc package, but ignore the books abstract classes
• Code visiter to translate to IR code
• Use or don’t use “generic IR trees”
Chapter 6: Activation Records
Activation Records

Because of recursion, several invocations of a subroutine may exist simultaneously.

Each invocation of a subroutine has its own instances of the local variables lasting from subroutine entry to return.

A subroutine invocation requires the runtime system maintain information about the invocation including the values of local variables (and much more: return address, saved state of the machine, temporaries values, access to the non-local environment, ...).

This collection of information is called an activation record.
A subroutine returns only after all the function it has called have returned. Thus subroutine invocations behave in a LIFO manner. So keep activation records in a stack.
Activation Records

Appel, 2nd edition, page 118. “For historical reasons, the run-time stack usually starts at a high memory address . . .”

Runtime Organization of a Program in Memory

- high memory
  - the stack (local variables)
- low memory
  - the heap (dynamically allocated variables)
  - globals variables, program code (read only)
Static link
Dynamic link
Return address
Parameters
Local variables
Register save area
Usual Layout of Activation Record for SPARC architecture
SPARC calling sequence

1. Caller puts arguments in %o0, %o1, ..., %o5. For example:
   
   ```
   mov 1,%o0 ! pass argument 0 in the register %o0
   set LABEL,%o1 ! pass argument 1 in the register %o1
   mov %l4,%o2 ! pass argument 2 in the register %o2
   ```

2. If more than six arguments, then the caller puts them in the (caller’s) argument build area. For example:

   ```
   mov 6,[%sp+4*(16+6)] ! pass argument 6 on the stack
   set LABEL,[%sp+4*(16+7)] ! pass argument 7 on the stack
   ```

3. Caller then executes call instruction, which saves the pc in %o7 and jumps to a label. For example:

   ```
   call f
   nop
   ```
4. Callee executes `save` instruction, which shifts the register window set and reserves space for new stack frame. For example:

```
save %sp, -4*(16+1+t+x)&-8, %sp
```

where \( t \) is the number of locals and temporaries, and \( x \) is the maximum arguments needed by any subroutines of the callee. Stack pointers must be double word aligned.

5. If needed, the callee saves the incoming arguments (0 through 5) into the argument build area of the caller’s frame. For example:

```
st %i0, [%fp+4*(16+0)]    ! store arg 0 into stack
st %i1, [%fp+4*(16+1)]    ! store arg 1 into stack
```

Local variables and temporaries of the current frame are accessed with negative offsets from the `%fp`. For example:

```
ld [%fp-4*(1+0)], %l0    ! load local/temp 0
ld [%fp-4*(1+1)], %l1    ! load local/temp 1
```
SPARC return sequence

1. Callee puts (integer) return value in %i0. For example:
   mov 1,%i0  ! put return value in %i0

2. Callee executes `restore` instruction, which resets the register window set and pops frame.

3. Callee then executes `ret` instruction, which jumps to %i7+8 (just past the call and its delay slot in the caller). For example:
   ret       ! "jmpl %i7+8, %g0" after next instruction
Nonlocal variable access

1. Whenever a function $f$ is called, it can be passed a pointer to the frame of the function statically enclosing $f$; this pointer is the *static link*.

2. A global array can be maintained, containing – in position $i$ – a pointer to the frame of the most recently entered procedure whose *static nesting depth* is $i$. This array is called a *display*.

3. When $g$ calls $f$, each variable of $g$ that is actually accessed by $f$ (or by any function nested inside $f$) is passed to $f$ as an extra argument. This is called *lambda lifting*. 
Nonlocal Variable Access

A local variable is accessed as follows:

\[ r := \text{fp} \quad \text{! relative to frame pointer} \]
\[ r := \text{M}[r+offset] \quad \text{! access local variable} \]

A non-local variable at static distance two is accessed as follows:

\[ r := \text{fp} \quad \text{! relative to frame pointer} \]
\[ r := \text{M}[r+sl] \quad \text{! follow first static link} \]
\[ r := \text{M}[r+sl] \quad \text{! follow second static link} \]
\[ r := \text{M}[r+offset] \quad \text{! access the variable at sd=2} \]
Nonlocal Variable Access

\[ M[fp+k] \]

\[
\text{MEM (BINOP (PLUS,}
\text{ new TEMP (frame.FP()), CONST (k)))}
\]
line 21. prettyprint calls show
    (show’s SL is set to prettyprint’s own frame pointer)

line 15. show calls indent
    (indent’s SL is set to show’s own frame pointer)

line 13. indent calls write

line 17. show calls itself recursively

line 15. show calls indent

line 13. indent calls write
line 14. indent accesses the variable output
An abstract structure to represent frame information

```java
package frame;

/* Appel, 2nd edition, Chapter 6, page 127 */
public abstract class Frame {
    public final Label instructionPointer;
    public final List<Access> formals; // including implicit static

    public Frame (Label i, List<? extends Access> f) {
        instructionPointer = i; formals = f;
    }

    public int getNumberOfFormals () { return formals.size(); }

    abstract public Frame newFrame (Label name, List formals);
    abstract public Access allocateLocal (boolean escape);
    public Access allocateLocal () { return allocateLocal(true); }

    public abstract temp.Temp FP();  /* Chapter 7, page 143 */
}
```
package frame;
import tree.Exp;

/*
 */
public abstract class Access {
    public abstract Exp access (Exp framePtr);
}
The class Frame holds information about formal parameters and local variables allocated in this frame. To make a new frame for a function $f$ with $k$ formal parameters, call `newFrame(f,l)`, where $l$ is a list of $k$ booleans: `true` for each parameter that escapes and `false` for each parameter that does not. (In MiniJava, no parameters ever escape.) The result will be a Frame object.


Calling `newFrame()` and getting a new frame might change the caller’s frame. For example, the maximum size of the argument build area may increase.
package temp;

public class Temp {}

public class Label {}

new temp.Label ("ClassName"+"$"+"methodName")
package translate;
import framel.Frame;
import translate.Access;
import java.util.LinkedList;

public class Level {
    public final Level parent;
    public final Frame frame;
    public Level (Level p, String name, LinkedList formals);
    public Access allocateLocal (boolean escape);
    public Access allocateLocal () { return allocateLocal(true); }
}

Chapter 7: Translation to Intermediate Code
Intermediate Representation (IR)

- IR should be easy to convert from the abstract syntax; easy to convert to assembly. It should support machine-independent optimizations.
- Often compilers use several IRs.

\[
\text{syntax } \Rightarrow \text{ IR}_1 \Rightarrow \text{ IR}_2 \ldots \Rightarrow \text{ IR}_k \Rightarrow \text{ assembly code}
\]

![Diagram](image)

Figure 6.2: A compiler might use a sequence of intermediate representations

- MiniJava compiler uses only one IR: `tree.Exp`
An intermediate language helps modularize the back end of the compiler.

**FIGURE 7.1.** Compilers for five languages and four target machines: (a) without an IR, (b) with an IR.
Figure 10.3: A middle-end and its ILs simplify construction of a compiler suite that must support multiple source languages and multiple target architectures.
abstract class Exp
    class CONST (int value)
    class NAME (tree.Label label)
    class TEMP (tree.NameOfTemp temp)
    class BINOP (int binop, Exp left, Exp right)
    class MEM (Exp exp)
    class CALL (Exp fun, ExpList args)
    class ESEQ/RET (Stm stm, Exp exp)

abstract class Stm
    class MOVE (Exp dst, Exp src)
    class EVAL (Exp exp)
    class JUMP (Exp exp, List<tree.Label> targets)
    class CJUMP (int rel, Exp l, Exp r, Label t, Label f)
    class SEQ (Stm left, Stm right)
    class LABEL (tree.NameOfLabel label)

[RET is more pronouncible than ESEQ.]
Intermediate Representation

Expressions which stand for the computation of some value (possibly with side effects):

**CONST(i)** The integer constant \( i \).

**NAME(n)** The symbolic constant \( n \) corresponding to an assembly language label.

**Temp As Exp(t)** Temporary \( t \). A temporary in the abstract machine is similar to a register in a real machine. However, the abstract machine has an infinite number of temporaries.

**BINOP()** The application of binary operator \( o \) to operands \( e_1 \) and \( e_2 \). Subexpression \( e_1 \) is evaluated before \( e_2 \).
Intermediate Representation

The remaining expressions which stand for the computation of some value (possibly with side effects):

\textbf{MEM(e)} The contents of a word of memory at address \texttt{e}.

\textbf{CALL()} Procedure call.

\textbf{EQ(s,e)/RET} The statement \texttt{s} is evaluated for side effects, then \texttt{e} is evaluated for a result.
Intermediate Representation

The statements of the intermediate representation which perform side effects and control flow:

MOVE(TEMP t,e) Store the results of evaluating e into the temporary t.

MOVE(MEM e1,e2) Store the results of evaluating e2 at the address e1.

EVAL(e) Evaluate e for its side effects and discard the result.
(Called EXP by Appel.)

JUMP(e,l) Transfer control to address e
The remaining statements which perform side effects and control flow:

CJUMP   Evaluate $e_1$ and $e_2$ in that order; compare using relational operator $o$. If the result is true, jump to $t$, otherwise jump to $f$.

SEQ(s1,s2) The statement $s_1$ followed by $s_2$.

LABEL(n) Define name to be the current machine code address.
Example

new SEQ (new MOVE (temp,1), // temp := 1;
new SEQ (new CJUMP(<,x,5,T,F), // if x<5 goto T else F
new SEQ (new LABEL(F), // F:
new SEQ (new MOVE (temp,0), // temp := 0;
        new LABEL(T)  // T:
    )))
}}

Example 1

Abstract Syntax

\begin{align*}
\text{OpExp} & (\text{PLUS}, \\
\text{IntExp} & (3), \\
\text{IntExp} & (4))
\end{align*}

Intermediate Trees

\begin{align*}
\text{BINOP} & (\text{PLUS}, \\
\text{CONST} & 3, \\
\text{CONST} & 4)
\end{align*}
Lazy IR Trees

How do you generate good IR code bottom-up in a tree traversal?

The problem is the code you want to produce depends on the context in which it is used. The boolean conjunctions are the most notable example. In some contexts the code must produce a 0 (false) or a 1 (true). In other contexts the code is used to control a test.
Lazy IR Trees

A solution

Do not produce the IR tree directly. Instead use a class that will make the tree later, when it is known which one of three contexts is desired.
package translate;
class LazyIRTree {
    abstract Exp asExp();
    abstract Stm asStm();
    Stm asCond (Label t, Label f) { throw new UnsupportedOperation; }
    public String toString () {
        return String.format ("IR: %s", asStm().toString());
    }
}

Three views of the code:
1. as an expression
2. as a statement
3. as a conditional
“The whole point of the Cx representation is that conditional expressions can be combined easily with the MiniJava operator &&.” Appel, 2nd, Section 7.2, page 149.
Lazy IR Trees

class ExpIRTree extends LazyIRTree

class StmIRTree extends LazyIRTree

abstract class Cx extends LazyIRTree
    class RelCx extends Cx /* Page 149. */
Lazy IR Trees

class ExpIRTree extends LazyIRTree {
    private final tree.Exp exp;
    ExpIRTree (tree.Exp e) { exp = e; }
    tree.Exp asExp() { return exp; }
    tree.Stm asStm() { return new tree.EVAL(exp); }
    // asCond not implemented
}

class StmIRTree extends LazyIRTree {
    private final tree.Stm stm;
    StmIRTree (tree.Stm s) { stm = s; }
    tree.Stm asStm() { return stm; }
    // asExp, asCond not implemented
}
abstract class Cx extends LazyIRTree {
    tree.Exp unEx() { /* Program 7.2, page 161.*/ } 
    abstract tree.Stm unCx (Label t, Label f);
    // unNx "left as exercise"
}
“Making ‘simple’ Cx expressions from Absyn comparison operators is easy with the CJUMP operator.” Appel, 2nd, Section 7.2, page 149.

class RelCx extends Cx { /* Page 149. */
    final private int relop;
    final private tree.Exp left, right;

    RelCx (int op, tree.Exp l, tree.Exp r) {
        relop = op; left = l; right = r;
    }

    public tree.Stm asCond (Label t, Label f) {
        // new tree.CJUMP
    }
}
If $i=$translate("x<5"), then $i$.asCond(t,f) should be 
CJUMP(LT,$x$,5,t,f).
class Rel_LIRT extends Cond_LIRT {
    final private int relop;
    final private Exp left, right;
    Rel_LIRT (final int op, final Exp l, final Exp r) {
        relop = op; left = l; right = r;
    }
    @Override
    public Stm asCond (NameOfLabel t, NameOfLabel f) {
        return new CJUMP (relop, left, right, t, f);
    }
    // This conditional test is used only for its side effects!
    public Stm asStm() {
        return new SEQ(new EVAL(left),new EVAL(right));
    }
}
This sketch of the class appears in Appel, 2nd, Section 7.2, page 150.

class IfThenElseExp extends LazyIRTree {
    private final LazyIRTree cond, e2, e3;
    Label t, f, join;
    IfThenElseExp (LazyIRTree c, LazyIRTree thenClause, LazyIRTree elseClause)
        assert c!=null; assert thenClause!=null;
        cond = c; e2 = thenClause; e3 = elseClause;
    }
    public Exp asExp() { /* ... */ }
    public Stm asStm() { /* ... */ }
    public Stm asCond (Label tt, Label ff) { /* ... */ }
}
A Simpler Translate

“To simplify the implementation of translate, you may do without the Ex, Nx, Cx constructors. The entire translate module can be done with ordinary value-expressions.” Appel, page 178.
Fragments

syntax. Program

symbol table

Chapter 5

fragments

Chapter 7

Chapter 4
Fragments

Chapter 4

Chapter 7
7.1 Draw a picture of the IR tree that results from each of the following expressions. Assume all variables escape.

b.

```plaintext
let
    var i := 8
    type intArray = array of int
    var b := intArray[10] of 0
in
    b[i+1]
end
```
Chapter 8: Basic Blocks and Traces
Translation of a Method

Stm  Stm list  Inst list

\[ \text{linearize} \]

emit

chap 7 8 9
The constructs of the intermediate representation trees of package tree are crafted to match the capabilities of real machines. This facilitates the translation from the abstract syntax. Yet some aspects do not correspond exactly to a real machine.

- **CJUMP** can jump to one of two labels, but real machine instructions fall through to the next instruction if the condition is false.

- **ESEQ/RET** within expressions require attention to the order of evaluating the subtrees; the left subtree must always be evaluated first.

\[ 2*(x:=3)+x \]
\[ \text{BINOP}(+, \text{BINOP}(\times, 2, \text{ESEQ/RET}(x:=3, 3)), x) \]

- **CALL** within expressions

\[ \text{BINOP}(+, \text{call f2}(1, 2), \text{call f3}(1, 2, 3)) \]

- **CALL** within **CALL** complicates passing parameters in a fixed set of registers

\[ \text{call f1}(<\text{call f2}(1, 2), \text{call f3}(1, 2, 3)> \)
We can take any tree and rewrite it to an equivalent tree without the troublesome cases. The **SEQ** constructs will all move to the top of the tree and become unnecessary. The tree can be replaced by a list of the other constructs.

1. The tree is flattened by removing **SEQ** and **ESEQ**
2. The list is broken into basic blocks containing no internal jumps
3. The basic blocks are ordered into a set of *traces* in which every **CJUMP** is followed by its false label.

The package **canon** does all of this.

```
List<tree.Stmt> canon.Main.transform (tree.Stmt body)
```
Chapter 8: Basic Blocks and Traces
Section 8.1: Canonical Trees
A *canonical tree* is one in which

1. there are no SEQ or ESEQ[=RET]

2. the parent of each CALL is either EVAL or MOVE(TMP t, ...).

How can the ESEQ/RET nodes be eliminated? The idea is to lift them higher and higher in the tree, until they can become SEQ nodes.

Appel, 2nd, page 164.

See Appel, 2nd, Figure 8.1, page 165.
The statement $s$ is said to *commute* with an expression $e$, if the value of $e$ is not changed where $s$ executed immediately before $e$.

$$\text{BINOP}(op, e_1, \text{ESEQ/RET}(s, e_2)) = \text{ESEQ/RET}(s, \text{BINOP}(op, e_1, e_2)) =$$
static boolean commute (Stm a, Exp b) {
    return isNop(a) || 
    b instanceof NAME || 
    b instanceof COSNT;
}

static boolean isNop (Stm a) {
    return a instanceof EVAL && 
    ((EVAL)a).exp instanceof CONST;
}
See Appel, 2nd, Figure 8.1, page 165.
Move calls to the top level. Create temps to hold values of calls.
linearize
Chapter 8: Basic Blocks and Traces

Section 8.2: Taming Conditional Branches
A **basic block** is a sequence of statements that is always entered at the beginning and exited at the end:

- The first statement is a LABEL
- The last statement is a JUMP or CJUMP
- There are no other LABELs, JUMPs or CJUMPs.

Basic blocks can be arranged in any order, and the result of executing the program will be the same.
Schedule a trace.

Put blocks in list Q
while Q is not empty
  Start a new (empty) trace, call it T
  Remove the first block b from Q
  while b is not marked
    mark b
    append b to the end of T
  let s be the set of blocks to which b branches (if any)
  if there is any successor c in s
    b := c
  end the current trace T
MOVE(
    MEM (ESEQ (LABEL s, CONST 1828)),
    CONST 2718)

After linearize
1: LABEL s
2: MOVE (MEM (CONST 1828), CONST 2718)

After trace scheduling
1: LABEL s
2: MOVE (MEM (CONST 1828), CONST 2718)
3: JUMP (NAME BB$1)
MOVE (
    MEM (CONST 1828),
    ESEQ (LABEL s, CONST 2718))

After linearize
1: LABEL s
2: MOVE (MEM (CONST 1828), CONST 2718)

After trace scheduling
1: LABEL s
2: MOVE (MEM (CONST 1828), CONST 2718)
3: JUMP (NAME BB$1)
EVAL(
    ESEQ(
        LABEL s, 
        CONST 2718))

After linearize
1: LABEL s

After trace scheduling
1: LABEL s
2: JUMP (NAME BB$1)
EVAL(
    CALL(
        ESEQ (LABEL s, CONST 2718),
        CONST 1828,
        MEM( NAME s$c$1)
    )
))

After linearize
1: LABEL s
2: EVAL ( CALL (CONST 2718, CONST 1828, MEM (NAME s$c$1)))
MOVE(
    TEMP t,
    CALL(
        ESEQ(
            LABEL s,
            CONST 2718),
            CONST 1828,
            MEM( NAME s$c$1)))

After linearize
1: LABEL s
2: MOVE (TEMP t, CALL (CONST 2718, CONST 1828, MEM (NAME s$c$1)))
MOVE(
  MEM(
    ESEQ(
      SEQ(
        CJUMP(LT,
          TEMP t,
          CONST 0,
          out,ok),
        LABEL ok),
        TEMP t)),
    CONST 2718)

After linearize
1: CJUMP (LT, TEMP t, CONST 0, out,ok)
2: LABEL ok
3: MOVE( MEM( TEMP t), CONST 2718)
m := 0; v := 0;
if v >= n goto 6
r := v; s := 0;
if r < n goto 4
v := v + 1; goto 2
x := M[r]; s := s + 1; if s <= m goto 8
m := 3;
r := r + 1; goto 4
return m
Optimal traces

prolog
while 1>N do
    body_stmts
od
epilog
begin:
    jump test
test:
    cjump 1>N done, body_label
body_label:
    body_stmts
    jump test
done:
    epilog
    jump elsewhere
Chapter 9: Instruction Selection
Instruction Selection

Problem: How to assign the machine register to all the temporaries?

Solution: Allocate registers *after* instruction selection—generate instructions templates.

1. Generate abstract instructions—instructions with holes for registers.

2. Allocate registers; spilling may require inserting some instructions.

3. Generate procedure entry and exit sequences
<table>
<thead>
<tr>
<th>Operation</th>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>$r_i \leftarrow r_j + r_k$</td>
<td>add $%r_j, %r_k, %r_i$</td>
</tr>
<tr>
<td>MUL</td>
<td>$r_i \leftarrow r_j \times r_k$</td>
<td>smul $%r_j, %r_k, %r_i$</td>
</tr>
<tr>
<td>SUB</td>
<td>$r_i \leftarrow r_j - r_k$</td>
<td>sub $%r_j, %r_k, %r_i$</td>
</tr>
<tr>
<td>ADDI</td>
<td>$r_i \leftarrow r_j + c$</td>
<td>add $%r_j, c, %r_i$</td>
</tr>
<tr>
<td>SUBI</td>
<td>$r_i \leftarrow r_j - c$</td>
<td>sub $%r_j, c, %r_i$</td>
</tr>
<tr>
<td>LOAD</td>
<td>$r_i \leftarrow M[r_j + c]$</td>
<td>ld $[%r_j, %c], %r_i$</td>
</tr>
<tr>
<td>STORE</td>
<td>$r_i \rightarrow M[r_j + c]$</td>
<td>st $%r_i, [%r_j + %c]$</td>
</tr>
<tr>
<td>MOVEM</td>
<td>$M[r_i] \rightarrow M[r_j]$</td>
<td>ld $[%r_j], %r; st %r, [%j_i]$</td>
</tr>
</tbody>
</table>
ld [%r1+%r2],%r3 ! r3 := MEM[r1+r2]
ld [%r1+c],%r3 ! r3 := MEM[r1+c]
ld [%r1-c],%r3 ! r3 := MEM[r1-c]
ld [%r1],%r3 ! r3 := MEM[r1]
st %r1, [%r2+%r3] ! MEM[r2+r3] := r1
st %r1, [%r2+c] ! MEM[r2+c] := r1
st %r1, [%r2-c] ! MEM[r2-c] := r1
st %r1, [%r2] ! MEM[r2] := r1
add %r1, %r2, %r3 ! r3 := r1 + r2
sub %r1, %r2, %r3 ! r3 := r1 - r2
smul %r1, %r2, %r3 ! (%y)r3 := r1 * r2 signed, integer multiplication
sdiv %r1, %r2, %r3 ! r3 := (%y)r1 / r2 signed, integer division
package assem;
public abstract class Instruction {
    // A concrete subclass should override these
    public List<Temp> use() { return null; }
    public List<Temp> def() { return null; }
    public List<Label> jumps() { return null; }

    public final String assem;
    protected Instruction (String a) { assem=a; }
    public String toString () { return assem; }

    public String format () {
        return format (DEFAULT_MAP);
    }

    public String format (Map<Temp,String> map) {

/* body provided */
Abstract Instruction Format

The string of an instr may refer to source registers ‘s0, ‘s1, … ‘s(k − 1), and destination registers ‘d0, ‘d1, etc. Jumps are OPER instructions that refer to labels ‘j0, ‘j1, etc. conditional jumps, which may branch away or fall through, typically have two labels in the jump list but refer to only one of them in the assem string.

(Later in dataflow analysis, these will be used variables and defined variables, respectively. And the jump list will be the successors of the statement.)
Abstract Instruction Format

Calling `i.format(m)` formats an assembly instruction as a string; `m` is an object implementing the `Map<Temp, String>` interface. The “parameters,” e.g., ’s0, ’d1, ’j2, are replaced with real registers and labels from the map.
Type of Instructions

The concrete subclasses of Instruction:

abstract Instruction (String assem)

Comment (String text)

LabelInstruction (NameOfLabel l)

OperationInstruction (String assem, String comment, 
  NameOfTemp d, NameOfTemp s1, NameOfTemp s2)

MoveInstruction (String assem, String comment, 
  NameOfTemp d, NameOfTemp s)
Abstract Instruction Format

OperationInstruction ("LOAD 'd0 <- M['s0+8]", new Temp(), frame.FP())

OperationInstruction ("ld ['s0+8]', 'd0", new Temp(), frame.FP())

ld [%fp+8], %10
Munch

Pattern match trees and possibly generate more temps.

```
+( *(TEMP t87, CONST 4), MEM (TEMP t92))

MULI 'd0 <- 's0*4 t908 t87
LOAD 'd0 <- M['s0+0] t909 t92
ADD 'd0 <- 's0*'s1 t910 t908,t909

sll 's0, 2, 'd0 t908 t87
ld ['s0+0], 'd0 t909 t92
add 's0, 's1, 'd0 t910 t908, t909
```