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
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:   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;
    case ID:
    case NUM:
    case LPAREN: return Tprime(F());
    default: print("expected ID, NUM, or left-paren");
                skipto(T_follow);
                return 0;
    }
}
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 $T' \rightarrow \ast FT'$ 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 * 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 compilation 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
;


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>$1 + 2 \times 3$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>.</td>
<td>shift 3</td>
</tr>
<tr>
<td>1</td>
<td>1 3</td>
<td>$\text{num} + 2 \times 3$</td>
<td>$E \rightarrow \text{num}$</td>
</tr>
<tr>
<td>2</td>
<td>1 5</td>
<td>$E + 2 \times 3$</td>
<td>shift 8</td>
</tr>
<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></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 * 3 $</td>
<td>shift 3</td>
</tr>
<tr>
<td></td>
<td>$ E + 2 * 3 $</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1 5 8 3</td>
<td>$ E + num $ *3 $</td>
<td>E → num</td>
</tr>
<tr>
<td></td>
<td>$ E + num $ *3 $</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1 5 8 5 12</td>
<td>$ E + E $ *3 $</td>
<td>shift 12</td>
</tr>
<tr>
<td></td>
<td>$ E + E $ *3 $</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>1 5 8 5 12 3</td>
<td>$ E + E $ * 3 $</td>
<td>shift 3</td>
</tr>
<tr>
<td></td>
<td>$ E + E $ * 3 $</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>1 5 8 5 12 3</td>
<td>$ E + E $ * num $</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$ E + E $ * num $</td>
<td></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 $ E + E * 3 $</td>
<td>· 1 · 2 ·</td>
<td>shift 3</td>
</tr>
<tr>
<td></td>
<td>1 5 8 5 12 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>$ E + E * num $</td>
<td>· 1 · 2 · 3</td>
<td>E → num</td>
</tr>
<tr>
<td></td>
<td>1 5 8 5 12 13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>$ E + E * E $</td>
<td>· 1 · 2 · 3</td>
<td>E → E*E</td>
</tr>
<tr>
<td></td>
<td>1 5 8 9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>$ E + E $</td>
<td>· 1 · 6</td>
<td>E → E+E</td>
</tr>
<tr>
<td></td>
<td>1 5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>$ 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

\[ \text{assign} \Rightarrow \]
\[ id := expr \Rightarrow \]
\[ A := expr \Rightarrow \]
\[ A := id \ast expr \Rightarrow \]
\[ A := B \ast expr \Rightarrow \]
\[ A := B \ast (expr) \Rightarrow \]
\[ A := B \ast (id + expr) \Rightarrow \]
\[ A := B \ast (A + expr) \Rightarrow \]
\[ A := B \ast (A + id) \Rightarrow \]
\[ A := B \ast (A + C) \]
Concrete Parse Tree

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

A := B * (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.
Abstract Syntax Tree

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
IntArrayTypet() 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.
Visitor Design Pattern

The visitor design pattern is a way of separating an algorithm from an object structure on which it operates. A result of this separation is the ability to add new operations to existing object structures without modifying those structures.

In essence, the visitor allows one to add new virtual functions to a family of classes without modifying the classes themselves; instead, one creates a visitor class that implements all of the appropriate specializations of the virtual function.
Double Dispatch

1. When the accept method is called in the program, its implementation is chosen based on both: the dynamic type of the element and the static type of the visitor.

2. Then the associated visit method is called, its implementation is chosen based on both: the dynamic type of the visitor and the static type of the element.
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 much 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>
</tr>
</thead>
<tbody>
<tr>
<td>IdExp</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>
</tr>
<tr>
<td>PlusExp</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>
</tr>
<tr>
<td>TimesExp</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>
</tr>
<tr>
<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>
</tr>
</thead>
<tbody>
<tr>
<td>Scrollbar</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>
</tr>
<tr>
<td>Canvas</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>
</tr>
<tr>
<td>Text</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>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(b) Graphic user interface
Visitor

With the object-oriented 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: \( \kappa \alpha \tau \alpha \) = 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 \textbf{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

~ryan/java/programs/class/Static.java

~ryan/ada/programs/package/Init.java
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;
  declare
    A:Integer:=3; B:Integer:=3;
  begin
    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
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 -- F
   A := B + C;
end F;
G: declare
   A:Integer:=2; B:Integer:=2;
H: declare
   C:Integer:=3; D:Integer:=3;
begin -- H
   C := A + D;
end I;
I: declare
   B:Integer:=4; D:Integer:=4;
begin
   B := A + C;
end I;
begin -- G
   B := A + C;
end G;
J: declare

454
begin scope E
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
end scope G
begin scope I
decl b, decl d
lookup b, a, c
end scope I
lookup b, a, c
end scope G
begin scope J
decl b, d
lookup b, a, d
end scope J
lookup a, b, c
end scope E
map={}; stack=[]; top=[bot]

Begin scope.

map={}; stack=[[bot]]; top=[none]
Declare 'a' with 'Table$Value@232204a1'.
map={a=v:[none]/}; stack=[[bot]]; top=a
Declare 'b' with 'Table$Value@4aa298b7'.
map={a=v:[none]/, b=v:a/}; stack=[[bot]]; top=b
Declare 'c' with 'Table$Value@7d4991ad'.
map={a=v:[none]/, b=v:a/, c=v:b/}; stack=[[bot]]; top=c

Begin scope.

map={a=v:[none]/, b=v:a/, c=v:b/}; stack=[c, [bot]]; top=[none]
Declare 'a' with 'Table$Value@232204a1'.
map={a=v:[none]/v:[none]/, b=v:a/, c=v:b/}; stack=[c, [bot]]; top=a
Declare 'b' with 'Table$Value@4aa298b7'.
map={a=v:[none]/v:[none]/, b=v:a/v:a/, c=v:b/v:b/}; stack=[c, [bot]]; top=b
Declare 'c' with 'Table$Value@7d4991ad'.
map={a=v2:[none]/v1:[none]/, b=v2:a/v1:a/, c=v2:b/v1:b/}; stack=[c, [bot]]; top=c

Lookup of 'a', found 'Table$Value@232204a1'.
Lookup of 'b', found 'Table$Value@4aa298b7'.
Lookup of 'c', found 'Table$Value@7d4991ad'.

End scope.

map={a=v:[none]/, b=v:a/, c=v:b/}; stack=[[bot]]; top=c
Begin scope.

map={a=v:[none]/, b=v:a/, c=v:b/}; stack=[c, [bot]]; top=[none]

Declare 'a' with 'Table$Value@28d93b30'.

map={a=v:[none]/v:[none]/, b=v:a/, c=v:b/}; stack=[c, [bot]]; top=a

Declare 'b' with 'Table$Value@1b6d3586'.

map={a=v:[none]/v:[none]/, b=v:a/v:a/, c=v:b/}; stack=[c, [bot]]; top=b

Begin scope.

map={a=v:[none]/v:[none]/, b=v:a/v:a/, c=v:b/}; stack=[b, c, [bot]]; top=[none]

Declare 'c' with 'Table$Value@4554617c'.

map={a=v:[none]/v:[none]/, b=v:a/v:a/, c=v:[none]/v:b/}; stack=[b, c, [bot]]; top=c

Declare 'd' with 'Table$Value@74a14482'.

map={a=v:[none]/v:[none]/, b=v:a/v:a/, c=v:[none]/v:b/, d=v:c/}; stack=[b, c, [bot]]; top=d

Lookup of 'c', found 'Table$Value@4554617c'.

Lookup of 'a', found 'Table$Value@28d93b30'.

Lookup of 'd', found 'Table$Value@74a14482'.

End scope.

map={a=v:[none]/v:[none]/, b=v:a/v:a/, c=v:b/}; stack=[c, [bot]]; top=b

Begin scope.

map={a=v2:[none]/v1:[none]/, b=v2:a/v1:a/, c=v:b/}; stack=[b, c, [bot]]; top=[none]

Declare 'b' with 'Table$Value@1540e19d'.

map={a=v2:[none]/v1:[none]/, b=v3:[none]/v2:a/v1:a/, c=v:b/}; stack=[b, c, [bot]]; top=b

Declare 'd' with 'Table$Value@677327b6'.

map={a=v2:[none]/v1:[none]/, b=v:[none]/v:a/v:a/, c=v:b/, d=v:b/}; stack=[b, c, [bot]]; top=d
Lookup of 'b', found 'Table$Value@1540e19d'.
Lookup of 'a', found 'Table$Value@28d93b30'.
Lookup of 'c', found 'Table$Value@6bc7c054'.

End scope.
map={a=v2:[none]/v1:[none]/, b=v2:a/v1:a/, c=v:b/}; stack=[c, [bot]]; top=b

Lookup of 'b', found 'Table$Value@1b6d3586'.
Lookup of 'a', found 'Table$Value@28d93b30'.
Lookup of 'c', found 'Table$Value@6bc7c054'.

End scope.

map={a=v:[none]/, b=v:a/, c=v:b/}; stack=[[bot]]; top=c

Begin scope.

map={a=v:[none]/, b=v:a/, c=v:b/}; stack=[c, [bot]]; top=[none]
Declare 'b' with 'Table$Value@14ae5a5'.
map={a=v:[none]/, b=v:[none]/v:a/, c=v:b/}; stack=[c, [bot]]; top=b
Declare 'd' with 'Table$Value@7f31245a'.
map={a=v:[none]/, b=v:[none]/v:a/, c=v:b/, d=v:b/}; stack=[c, [bot]]; top=d

Lookup of 'b', found 'Table$Value@14ae5a5'.
Lookup of 'a', found 'Table$Value@42a57993'.
Lookup of 'd', found 'Table$Value@7f31245a'.

End scope.

map={a=v:[none]/, b=v:a/, c=v:b/}; stack=[[bot]]; top=c

Lookup of 'a', found 'Table$Value@42a57993'.
Lookup of 'b', found 'Table$Value@75b84c92'.
Lookup of 'c', found 'Table$Value@6bc7c054'.

End scope.

map={}; stack=[]; top=[bot]
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;
let
  var a:=3 var b:=3
in
  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
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
final Map<String,Binder<Value>> map = new HashMap<>();
final Deque<String> lastOfLevel = new ArrayDeque<>();

public class Binder<V> {
    public final V value;
    public final String previousTop;
    public final Binder<V> tail;
    Binder (final V v, final String p, final Binder<V> t) {
        value=v; previousTop=p; tail=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`.

```
   null marks
   null top
```
Declare a.

<table>
<thead>
<tr>
<th>Symbol Table Example—2 of 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declare a.</td>
</tr>
<tr>
<td>a</td>
</tr>
<tr>
<td>b</td>
</tr>
<tr>
<td>c</td>
</tr>
<tr>
<td>null</td>
</tr>
<tr>
<td>top</td>
</tr>
</tbody>
</table>

```
Symbol Table

null null

null null

null null

null null

null null

null null

null null

null null

null null
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.
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 {  
    /* ... */  
}  

<table>
<thead>
<tr>
<th>FIELDS</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td>C</td>
</tr>
<tr>
<td>j</td>
<td>int []</td>
</tr>
<tr>
<td>q</td>
<td>int</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>METHODS</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>int</td>
</tr>
<tr>
<td>stop</td>
<td>bool</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PARAMS</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>int</td>
</tr>
<tr>
<td>q</td>
<td>int</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LOCALS</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ret</td>
<td>int</td>
</tr>
<tr>
<td>a</td>
<td>int</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PARAMS</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>int</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LOCALS</th>
<th></th>
</tr>
</thead>
</table>

**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 methods that is not yet entered into the symbol table.

See Appel, 2nd, page 112.
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'.");
    } else {
        return Type.THE_INTEGER_TYPE;
    }
}
A semantic error in MiniJava.

```java
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
E.java:011.034: Call to method 'Fac.ComputeFac' has the wrong number of
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
    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', formal
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
    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
    arg has type: int[], formal has type int
E.java:000.000: Method 'alice' not found in class 'Bob'
E.java:129.013: Local declaration of 'sz' hides a formal parameter or an argument of a method.
E.java:017.016: A primitive type 'int' has no methods.
E.java:018.016: An array cannot be dereferenced.
E.java:000.000: 'print' for integers only; found type 'boolean'.
E.java:003.014: Unary 'not' operation not applied to exp of type boolean.
E.java:024.007: Two classes with the same name 'C' in the program.
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 to main undeclared.
Overloading

Overloading of methods will not be tested.