CS 565, Jens Palsberg — Aug 19, 1998

Lecture Note 1: Java and Scheme

Contents

1 The Java Language

Java is a programming language of the 1990s.

$$\text{Java} \quad = \quad \text{C++ ++} --$$

If you know C++, good!
If you don't, even better!
Java is a better programming language than C++.

My personal perspective: they took C++, removed the stuff I am uncomfortable with, and put in some stuff I like.

The History of Java

- Java was designed in the 1990s at Sun Microsystems.
- The design of Java was initiated by Bill Joy and completed by Jim Gosling.
- Java was first intended for the smart consumer electronics market. In 1993 the design project was redirected to address the settop box market. In 1994 the Web came along, and the Java designers realized that Java was a good fit for Web programming.
- Java is popular among programming language researchers. At the conference on Principles on Programming Languages in January 1997, at least two papers are on Java.
- Guy Steele has done a good job on writing the specification of Java.
- Java runs on many platforms.
- Java is compiled to byte-code. The byte-code is interpreted by a byte-code interpreter. Thus, to port Java programs to a new platform, simply port the interpreter!

Java's Type System

$$
\begin{array}{rcl}
\mathsf{Type} & ::= & \mathsf{PrimitiveType} \;|\; \mathsf{ClassName} \;|\; \mathsf{InterfaceName} \\
& | & \mathsf{Type}\,[\,] \qquad ;;\ \text{array type}
\end{array}
$$

$$
\mathsf{ReturnType} \;::=\; \mathsf{Type} \;|\; \text{void}
$$

$\mathsf{PrimitiveType}$

$$
\begin{array}{clcl}
::= & \text{boolean} & ;; & \text{1-bit} \\
| & \text{char} & ;; & \text{2 unsigned bytes, Unicode} \\
| & \text{byte} & ;; & \text{1 signed byte (two's complement)} \\
| & \text{short} & ;; & \text{2 bytes signed (two's complement)} \\
| & \text{int} & ;; & \text{4 bytes signed (two's complement)} \\
| & \text{long} & ;; & \text{8 bytes signed (two's complement)} \\
| & \text{float} & ;; & \text{4 bytes, IEEE 754} \\
| & \text{double} & ;; & \text{8 bytes, IEEE 754}
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{ClassName} & ::= & \text{Identifier} \\
\mathsf{InterfaceName} & ::= & \text{Identifier}
\end{array}
$$

- Strings and arrays are objects.
- Casting: only between numeric types, between subclasses and superclasses, and between interfaces.
- Type correctness is checked at compile-time.

Java's Type System

The primitive types form a partial order which is a disjoint union of two trees:

$$\begin{tikzcd}
 & \text{byte} \arrow[r] & \text{short} \arrow[dr] & & & & \\
\text{boolean} & & & \text{int} \arrow[r] & \text{long} \arrow[r] & \text{float} \arrow[r] & \text{double} \\
 & & \text{char} \arrow[ur] & & & &
\end{tikzcd}$$

Subtyping in Java is a partial order of the types which extends the order of the primitive types in two ways:

1. if a class D explicitly extends a class C, then the classname type D is a subtype of the classname type C, and
2. if an interface J explicitly extends I, then the type J is a subtype of the type I.

Thus, an interface can be a subtype of an interface with fewer fields, but only if this is explicitly declared.

Interfaces can be mutually recursive.

Java vs. C++

Here is a list of C++ features which are *not* in Java:

- multiple inheritance
- operator overloading
- templates
- pointers
- undefined or architecture dependent constructs
- goto
- #define
- typedef
- enum
- union
- struct
- global variables (every variable is inside a class)
- global functions (every function must be a method)
- friend functions (in a package, “we are all friends”)
- virtual functions (all methods are by default virtual).

Java vs. C++

Here is a list of Java features which are *not* in C++:

- interfaces
- packages
- multithreading
- garbage collection
- array-bounds checks

The basic control structures of Java have C syntax:

- if, else, for, while, do while, switch

There is exception handling in Java (a la C++).

The expression syntax of Java is essentially that of C.

Parameter passing:

- Objects, including strings and arrays, are passed by reference.
- Data of primitive type are passed by value.

Java is Buzzword Compliant

In “The Java Language: A White Paper,” Sun describes Java as follows:

“Java: A

- simple,
- object-oriented,
- distributed,
- interpreted,
- robust,
- secure,
- architecture neutral,
- portable,
- high-performance,
- multithreaded, and
- dynamic language.”

“Hello World” in Java

```
public class HelloWorld {

   public static void main(String[] args) {
       System.out.println("Hello world!");
   }
}
```

- Store the program in a file with the same name as the public class plus a `.java` extension.
- Compile with `javac`. (It is in `/usr/local/java/`.)
- Example: `javac HelloWorld.java`.
  This produces the file `HelloWorld.class`.
- To execute, run the interpreter `java` (it is also in `/usr/local/java/`).
- Example: `java HelloWorld`.
  This prints `Hello World!`.

Interfaces in Java

- An interface can contain
  - method headers, and
  - constants (static final).
- A class can implement an interface.
- An interface can extend other interfaces.
- Subtyping is by name, not structural.
- No overriding of method headers.

Interface Example

```
interface Driveable {
    boolean startEngine();
    void stopEngine();
    float accelerate( float acc );
    boolean turn( Direction dir );
}
class Automobile implements Driveable() {
    ...
    boolean startEngine() {
        if ( notTooCold )
           engineRunning = true;
        ...
    }
    void stopEngine() {
        engineRunning = false
    }
    float accelerate( float acc ) {
        ...
    }
    boolean turn( Direction dir ) {
        ...
    }
}
```

Interface Example

```
Automobile auto = new Automobile();
Lawnmover mower = new Lawnmover();
Driveable vehicle;

vehicle = auto;
vehicle.startEngine();
vehicle.stopEngine();
...
vehicle = mower;
vehicle.startEngine();
vehicle.stopEngine();
```

Variables in Interfaces

```
interface Scaleable {
    static final int BIG = 0,
                  MEDIUM = 1,
                   SMALL = 2;

    void setScale( int size );
}

class Box implements Scaleable {
    void setScale( int size ) {
        switch( size ) {
            case BIG:
                 ...
            case MEDIUM:
                  ...
            case SMALL:
                  ...
        }
    }
    ...
}
```

Interface Extension

```
interface I extends I1, I2, ..., In {...}
```

- Note: `I1, I2, ..., In` must declare disjoint sets of names.

```
interface DynamicallyScaleable
          extends Scaleable {

    void changeScale( int size );
}
```

Objects and Classes in Java

```
public class Circle {
    public double x,y;    // the center
    public double r;      // the radius
    public double circumference() {
        return 2 * 3.14159 * r;
    }
    public double area() {
        return 3.14159 * r * r;
    }
}
```

We can use this class as follows.

```
Circle c = new Circle();
```

Accessing instance variables:

```
Circle c = new Circle();
c.x = 2.0; c.y = 2.0; c.r = 1.0;
```

Invoking methods:

```
double a;
a = c.area();
```

Inheritance and Super

- Single inheritance!
- One can override methods and shadow variables in superclasses. The old version can be accessed with `super`.

```
class Animal {
    ...
    void eat( FoodItem f ) {
    }
    ...
}

class Carnivore extends Animal {
    ...
    void eat( FoodItem f ) {
        if ( f instanceof Plant )
            // Complain profusely
        else
            super.eat( f );
    }
    ...
}
```

Class Variables, Class Methods

- A class variable is declared with the keyword `static`.
- A class variable is "global" for all instances of the class.
- Same story for class methods.

```
public class Circle {
    static int num_circles = 0;
              // class variable
    public double x,y,r;
              // instance variables
    ...
}
```

Constructors and Finalizers

Constructors:

- To create an object of class `Circle`, we can use the default constructor and write:

  ```
  new Circle();
  ```

- But we can also program constructors ourselves.
- Every constructor must have the name of the class.
- No return type can be specified, the return type is implicitly the class itself.

Finalizers:

- If an object contains a `finalize()` method, then it will be invoked just before the object is garbage collected.

Visibility Modifiers in Java

Modifier	Visibility
`private`	None
`private protected`	Subclasses only, inside or outside the package
none (default)	Classes in the package
`protected`	Classes in package and subclasses inside or outside the package
`public`	All classes

One can *increase* visibility in a subclass. (But one cannot decrease visibility.)

Packages in Java

- A package is a module. It creates a scope.
- The space of packagenames is flat.
- A java program can be divided into files. A file is a sequence of class and interface definitions. One can start a file with

  ```
  package Id
  ```

  This declares that the content of the file belongs to the package `Id`.
- If one omits the package declaration, then the contents becomes part of a default unnamed package.

The Import Statement

- One can write abbreviations for class and interface names with the `import` statement.

  ```
  package mypackage;

  import mytools.text.TextEditor;

  class MyClass {
      TextEditor editor;
      ...
  }
  ```

  The notation with "." is not about nesting.

The Import Statement

- There are three forms of the `import` statement.

  ```
  import packageId;
  import packageId.classId;
  import packageId.*;
  ```

- The first one allows the specified package to be known by the name of its last component (this is where "`.`" plays a role). For example,

  ```
  import java.awt.image;
  ```

  allows the class `java.awt.image.ImageFilter` to be called `image.ImageFilter`.

- The second one allows the specified class in the specified package to be known by its class name alone. For example,

  ```
  import java.util.Hashtable;
  ```

  allows the class `java.util.Hashtable` to be called `Hashtable`.

- The third form makes all classes in a package available by the class name.

Exception Handling in Java

Exceptions = instances of the class `java.lang.Exception`.

An exception is thrown from one point in the code and caught by another.

```
try {
    readFromFile("foo");
    ...
}
catch ( Exception e ) {
    // Handle error
    System.out.println(
      "Exception while reading file: " + e );
    ...
}
```

Exception Handling in Java

A try statement can have multiple catch clauses that specify different specific types (subclasses) of `Exception`. The catch clauses are evaluated in order, and the first possible (assignable) match is taken. At most one catch clause is executed.

```
try {
    readFromFile("foo");
    ...
}
catch ( FileNotFoundException e ) {
    // Handle file not found
    ...
}
catch ( IOException e ) {
    // Handle read error
    ...
}
catch ( Exception e ) {
    // Handle all other errors
    ...
}
```

Exception Handling in Java

Uncaught exceptions are, in turn, thrown from the method in which it appeared.

```
void sendRequest() throws IOException {
  write( ... ); // can throw an IOException
}
```

If an exception never gets caught, the program terminates with a run-time error message.
We can throw our own exceptions:

```
throw new MyException();
```

Exception Handling in Java

Exceptions are divided into two categories:

General exceptions Exceptions a reasonable application should try to handle gracefully. Examples: Missing files, unavailable hosts.

Run-time exceptions Exceptions from which we would not normally expect our software to try to recover. Examples: Out of memory, array index out of bounds.

It is not a compile-time error to ignore the possibility of subclasses of `RuntimeException` to be thrown; methods do not have to declare that they can throw them.

By the way, multithreading in Java is not covered here; take Parallel Programming.

2 MiniJava

```
class Fac {
  public static void main(String [] arg) {
    System.out.println(new A().sumupto(5));
  }
}

interface I {
  int sumupto(int n);
}

class A implements I {
  int a;
  public int sumupto(int n) {
    if (0<n) {
      a=n+((new A()).sumupto(n-1));
    }
    else {
      a=a;
    }
    return a;
  }
}
```

MiniJava Grammar

```
Goal                 ::= MainClass
                           TypeDeclaration*
MainClass            ::= class Identifier {
                           public static void main
                              ( String [ ] Identifier ) {
                              MethodBody
                           }
                         }
TypeDeclaration      ::= InterfaceDeclaration
                      |  InterfaceExtends
                      |  ClassDeclaration
InterfaceDeclaration ::= interface Identifier InterfaceBody
InterfaceExtends     ::= interface Identifier
                           extends Identifier InterfaceBody
InterfaceBody        ::= { InterfaceMember* }
InterfaceMember      ::= ResultType Identifier ( Type Identifier ) ;


ClassDeclaration     ::= class Identifier implements Identifier {
                           ClassBodyDeclaration* }
ClassBodyDeclaration ::= FieldDeclaration
                      |  MethodDeclaration
FieldDeclaration     ::= Type Identifier ;
MethodDeclaration    ::= public ResultType Identifier
                           ( Type Identifier ) { MethodBody }
MethodBody           ::= StatementListReturn
                      |  StatementList
```

MiniJava Grammar

$$
\begin{array}{rcl}
\textit{Type} & ::= & \textit{BooleanType} \\
 & | & \textit{IntegerType} \\
 & | & \textit{Identifier} \\
\textit{BooleanType} & ::= & \texttt{boolean} \\
\textit{IntegerType} & ::= & \texttt{int} \\
\textit{ResultType} & ::= & \textit{VoidType} \\
 & | & \textit{Type} \\
\textit{VoidType} & ::= & \texttt{void}
\end{array}
$$

$$
\begin{array}{rcl}
\textit{Block} & ::= & \texttt{\{}\ \textit{StatementList}\ \texttt{\}} \\
\textit{StatementListReturn} & ::= & \textit{StatementList}\ \texttt{return}\ \textit{Expression}\ \texttt{;} \\
\textit{StatementList} & ::= & \textit{Statement}^{*} \\
\textit{Statement} & ::= & \textit{Block} \\
 & | & \textit{AssignmentStatement} \\
 & | & \textit{IfStatement} \\
 & | & \textit{MessageSendStatement} \\
 & | & \textit{PrintStatement} \\
\textit{AssignmentStatement} & ::= & \textit{Identifier}\ \texttt{=}\ \textit{Expression}\ \texttt{;} \\
\textit{IfStatement} & ::= & \texttt{if (}\ \textit{Expression}\ \texttt{)}\ \textit{Statement} \\
 & & \quad \texttt{else}\ \textit{Statement} \\
\textit{MessageSendStatement} & ::= & \textit{MessageSend}\ \texttt{;} \\
\textit{PrintStatement} & ::= & \texttt{System.out.println(}\ \textit{Expression}\ \texttt{)}\ \texttt{;}
\end{array}
$$

MiniJava Grammar

Expression	::=	*CompareExpression*
	|	*PlusExpression*
	|	*MinusExpression*
	|	*MessageSend*
	|	*PrimaryExpression*
CompareExpression	::=	*PrimaryExpression* `<` *PrimaryExpression*
PlusExpression	::=	*PrimaryExpression* `+` *PrimaryExpression*
MinusExpression	::=	*PrimaryExpression* `-` *PrimaryExpression*
MessageSend	::=	*PrimaryExpression* `.` *Identifier* `(` *Expression* `)`
PrimaryExpression	::=	*IntegerLiteral*
	|	*TrueLiteral*
	|	*FalseLiteral*
	|	*Identifier*
	|	*ThisExpression*
	|	*AllocationExpression*
	|	*BracketExpression*
IntegerLiteral	::=	*INTEGER_LITERAL*
TrueLiteral	::=	`true`
FalseLiteral	::=	`false`
Identifier	::=	*IDENTIFIER*
ThisExpression	::=	`this`
AllocationExpression	::=	`new` *Identifier* `(` `)`
BracketExpression	::=	`(` *Expression* `)`

3 The Java Compiler Compiler

- Can be thought of as “Lex and Yacc for Java.”
- It is based on LL(k) rather than LALR(1).
- Grammars are written in EBNF.
- The Java Compiler Compiler transforms an EBNF grammar into an LL(k) parser.
- The EBNF grammar can have embedded action code written in Java, just like a Yacc grammar can have embedded action code written in C.
- The whole input is given in just one file (not two).
- The lookahead can be changed by writing `LOOKAHEAD(...)`.

Example: MiniJava

Part of a grammar for MiniJava in JavaCC notation:

```
options {
  JAVA_UNICODE_ESCAPE = true;
}

PARSER_BEGIN(MiniJavaParser)
  public class MiniJavaParser {}
PARSER_END(MiniJavaParser)

SKIP : /* WHITE SPACE */
{
  " "
| "\t"
| "\n"
| "\r"
| "\f"
}

SPECIAL_TOKEN : /* COMMENTS */
{
  <SINGLE_LINE_COMMENT: "//" (~["\n","\r"])*
     ("\n"|"\r"|"\r\n")>
| <FORMAL_COMMENT: "/**" (~["*"])* "*"
     ("*" | (~["*","/"] (~["*"])* "*"))* "/">
| <MULTI_LINE_COMMENT: "/*" (~["*"])* "*"
     ("*" | (~["*","/"] (~["*"])* "*"))* "/">
}
```

```
TOKEN :
{
  < LPAREN: "(" >
| < RPAREN: ")" >
| < LSQPAREN: "[" >
| < RSQPAREN: "]" >
| < LBRACE: "{" >
| < RBRACE: "}" >
| < SEMICOLON: ";" >
| < DOT: "." >
| < ASSIGN: "=" >
| < LT: "<" >
| < PLUS: "+" >
| < MINUS: "-" >
| < BOOLEAN: "boolean" >
| < CLASS: "class" >
| < INTERFACE: "interface" >
| < ELSE: "else" >
| < EXTENDS: "extends" >
| < FALSE: "false" >
| < IF: "if" >
| < IMPLEMENTS: "implements" >
| < INTEGER: "int" >
| < MAIN: "main" >
| < NEW: "new" >
| < PUBLIC: "public" >
| < RETURN: "return" >
| < STATIC: "static" >
| < STRING: "String" >
| < THIS: "this" >
```

```
| < TRUE: "true" >
| < PRINT: "System.out.println" >
| < VOID: "void" >
}

TOKEN : /* LITERALS */
{
  < INTEGER_LITERAL: ( ["1"-"9"] (["0"-"9"])* | "0" ) >
}

TOKEN : /* IDENTIFIERS */
{
  < IDENTIFIER: <LETTER> (<LETTER>|<DIGIT>)* >
|
  < #LETTER:
      [
       "\u0024",
       "\u0041"-"\u005a",
       "\u005f",
       "\u0061"-"\u007a",
       "\u00c0"-"\u00d6",
       "\u00d8"-"\u00f6",
       "\u00f8"-"\u00ff",
       "\u0100"-"\u1fff",
       "\u3040"-"\u318f",
       "\u3300"-"\u337f",
       "\u3400"-"\u3d2d",
       "\u4e00"-"\u9fff",
       "\uf900"-"\ufaff"
      ]
```

```
  >
|
  < #DIGIT:
      [
       "\u0030"-"\u0039",
       "\u0660"-"\u0669",
       "\u06f0"-"\u06f9",
       "\u0966"-"\u096f",
       "\u09e6"-"\u09ef",
       "\u0a66"-"\u0a6f",
       "\u0ae6"-"\u0aef",
       "\u0b66"-"\u0b6f",
       "\u0be7"-"\u0bef",
       "\u0c66"-"\u0c6f",
       "\u0ce6"-"\u0cef",
       "\u0d66"-"\u0d6f",
       "\u0e50"-"\u0e59",
       "\u0ed0"-"\u0ed9",
       "\u1040"-"\u1049"
      ]
  >
}
```

```
/*********************************
 * The MiniJava Grammar Starts Here *
 *********************************/


void Goal() :
{}
{
  MainClass()
  ( TypeDeclaration() )*
  <EOF>
}

void MainClass() :
{}
{
  "class" Identifier() "{"
      "public" "static" "void" "main"
          "(" "String" "[" "]" Identifier() ")" "{"
          MethodBody()
      "}"
  "}"
}
```

```
void TypeDeclaration() :
{}
{
  LOOKAHEAD(3)
  InterfaceDeclaration()
|
  InterfaceExtends()
|
  ClassDeclaration()
}

void InterfaceDeclaration() :
{}
{
  "interface" Identifier()
  InterfaceBody()
}

void InterfaceExtends() :
{}
{
  "interface" Identifier() "extends" Identifier()
  InterfaceBody()
}

void InterfaceBody() :
{}
{
  "{" ( InterfaceMember() )* "}"
}
```

```
void MethodBody() :
{}
{
  LOOKAHEAD( StatementList() "return" )
  StatementListReturn()
|
  StatementList()
}

void StatementListReturn() :
{}
{
  StatementList() "return" Expression() ";"
}

void StatementList() :
{}
{
  ( Statement() )*
}
```

4 The Visitor Pattern

For **object-oriented programming**,

the Visitor pattern **enables**

the definition of a **new operation**

on an **object structure**

without changing the classes

of the objects.

Gamma, Helm, Johnson, Vlissides:
Design Patterns, 1995.

A **Visitor** is a specialized version of
a **Walkabout**.

Sneak Preview

When using the **Visitor** pattern,

- the set of classes must be fixed in advance, and
- each class must have an accept method.

Walkabouts do not rely on these restrictions:

- The idea: separate 1) accessing subobjects, and 2) acting on them.
- The implementation of Walkabouts uses **reflection** to access subobjects.

Three steps: 1) Programming without Visitors, 2) Visitors, 3) Walkabouts.

First Approach: Instanceof and Type Casts

The running Java example: summing an integer list.

```
interface List {}

class Nil implements List {}

class Cons implements List {
  int head;
  List tail;
}
```

First Approach: Instanceof and Type Casts

```
List l;      // The List-object
int sum = 0;
boolean proceed = true;
while (proceed) {
  if (l instanceof Nil)
     proceed = false;
  else if (l instanceof Cons) {
     sum = sum + ((Cons) l).head;
     l = ((Cons) l).tail;
     // Notice the two type casts!
  }
}
```

Advantage: The code is written without touching the classes `Nil` and `Cons`.

Drawback: The code constantly uses type casts and `instanceof` to determine what class of object it is considering.

Second Approach: Dedicated Methods

The first approach is **not** object-oriented!

To access parts of an object, the classical approach is to use dedicated methods which both access and act on the subobjects.

```
interface List {
  int sum();
}
```

We can now compute the sum of all components of a given `List`-object `l` by writing `l.sum()`.

Second Approach: Dedicated Methods

```
class Nil implements List {
  public int sum() {
    return 0;
  }
}

class Cons implements List {
  int head;
  List tail;
  public int sum() {
    return head + tail.sum();
  }
}
```

Advantage: The type casts and `instanceof` operations have disappeared, and the code can be written in a systematic way.

Disadvantage: For each new operation on `List`-objects, new dedicated methods have to be written, and all classes must be recompiled.

Third Approach:
The Visitor Pattern

The Idea:

- Divide the code into an object structure and a Visitor (akin to Functional Programming!)
- Insert an `accept` method in each class. Each accept method takes a Visitor as argument.
- A Visitor contains a `visit` method for each class (overloading!) A method for a class C takes an argument of type C.

```
interface List {
  void accept(Visitor v);
}
```

```
interface Visitor {
  void visit(Nil x);
  void visit(Cons x);
}
```

Third Approach: The Visitor Pattern

- The purpose of the `accept` methods is to invoke the `visit` method in the Visitor which can handle the current object.

```
class Nil implements List {
  public void accept(Visitor v) {
    v.visit(this);
  }
}

class Cons implements List {
  int head;
  List tail;
  public void accept(Visitor v) {
    v.visit(this);
  }
}
```

Third Approach: The Visitor Pattern

- The control flow goes back and forth between the `visit` methods in the Visitor and the `accept` methods in the object structure.

```
class SumVisitor implements Visitor {
  int sum = 0;
  public void visit(Nil x) {}
  public void visit(Cons x) {
    sum = sum + x.head;
    x.tail.accept(this);
  }
}
.....
SumVisitor sv = new SumVisitor();
l.accept(sv);
System.out.println(sv.sum);
```

Notice: The `visit` methods describe both 1) actions, and 2) access of subobjects.

Comparison

The Visitor pattern combines the advantages of the two other approaches.

	Frequent type casts?	Frequent recompilation?
Instanceof and type casts	Yes	No
Dedicated methods	No	Yes
The Visitor pattern	No	No

The advantage of Visitors: New methods without recompilation!
Requirement for using Visitors: All classes must have an accept method.

Tools that use the Visitor pattern:

- JJTree (from Sun Microsystems) and the Java Tree Builder (from Purdue University), both frontends for The Java Compiler Compiler from Sun Microsystems.

Visitors: Summary

- **Visitor makes adding new operations easy.** Simply write a new visitor.
- **A visitor gathers related operations.** It also separates unrelated ones.
- **Adding new classes to the object structure is hard.** Key consideration: are you most likely to change the algorithm applied over an object structure, or are you most like to change the classes of objects that make up the structure.
- **Visitors can accumulate state.**
- **Visitor can break encapsulation.** Visitor's approach assumes that the interface of the data structure classes is powerful enough to let visitors do their job. As a result, the pattern often forces you to provide public operations that access internal state, which may compromise its encapsulation.

Fourth Approach: Walkabouts

The Idea:

- One generic class `Walkabout` is the superclass of all visitors.
- `Walkabout` has a just one `visit` method which takes an argument of type `Object`.
- By default, `Walkabout` visits the fields of the argument in sequence.

```
class Walkabout {
  void visit(Object v) {
    if ( v != null )
       if (this has a public visit method
              for the class of v)
          this.visit(v);
       else
       if (v is not of primitive type)
          for (each field f of v)
            this.visit(v.f);
  }
}
```

Walkabout using Reflection

- The fields and the class name are determined by reflection.

```
class Walkabout {
  void visit(Object v) {
    if ( v != null ) {
      Object [] os = {v};
      Class vClass = v.getClass();
      Class [] cs = {vClass};
      try { this.getClass().getMethod("visit",cs)
                           .invoke(this,os); }
      catch (java.lang.NoSuchMethodException e) {
        if (!((v instanceof Short) | (v instanceof Boolean) |
              (v instanceof Character) |
              (v instanceof Byte)  | (v instanceof Number))) {
          java.lang.reflect.Field [] vFields =
                                        vClass.getFields();
          for (int i=0; i<vFields.length; i++) {
              try { this.visit(vFields[i].get(v)); }
              catch (java.lang.IllegalAccessException e1) {
                ; } // Cannot happen.
          }
        }
      }
      catch (java.lang.Exception e) {;}  // Cannot happen.
    }
  }}
```

Visiting while going Walkabout

```
interface List {}

class Nil implements List {}

class Cons implements List {
  int head;
  List tail;
}


class SumWalkabout extends Walkabout {
  int sum = 0;
  public void visit(Cons x) {
    sum = sum + x.head;
    this.visit(x.tail);
  }
}



SumWalkabout sw = new SumWalkabout();
sw.visit(l);
System.out.println(sw.sum);
```

Comparison

- The customized `Visitor` interface has been replaced with the general `Walkabout` class.
- The basic classes (for lists) are here defined exactly as in the first approach since there are no `accept` methods to worry about.
- Even so, the customized walkabout actually is simpler than the corresponding visitor in two ways.
- First, there is no `visit` method for `Nil` in class `SumWalkabout`. Such a method is unnecessary because class `Walkabout` will determine that `Nil` has no fields.
- Second, and more fundamental, is that the indirection caused by `accept` in

  ```
  x.tail.accept(this);
  ```

  has been replaced by the direct recursive call

  ```
  this.visit(x.tail);
  ```

Performance Measurements

The benchmark program: a slightly modified version of the running example.

The following run times were obtained on a Sun Ultra 1, model 170E, 167-MHz UltraSPARC, 2.1-Gbyte 7200 F/W SCSI-2 disk, and 128-Mbyte Memory. The compilation and execution of the Java code was done with Sun's JDK 1.1.3.

Approach	Run-time
Construction of list only	0.68 s
type casts	1.18 s
dedicated methods	0.99 s
`Visitor`	1.17 s
`Walkabout`	4 min 59.93 s

Big performance penalty for using Walkabouts!
The Reason: Reflection!

5 The Java Tree Builder

The Java Tree Builder (JTB) has been developed here at Purdue in my group.

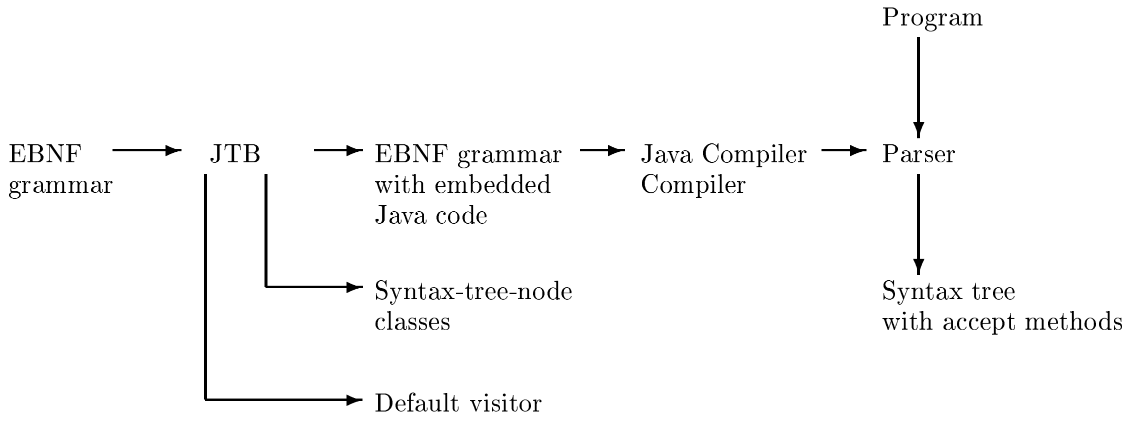
JTB is a frontend for The Java Compiler Compiler.

JTB supports the building of syntax trees which can be traversed using visitors. When all tree traversal and manipulation code is written using visitors, the classes for the syntax tree nodes are generated once by JTB and never touched again. This gives a welcome degree of flexibility and modularity for projects where several programmers, perhaps over time, write code for traversing syntax trees generated from the same grammar. Rather than modifying the classes for the syntax tree nodes, each programmer writes one or more visitors.

The current version of our tool transforms a bare EBNF grammar into three components:

- an EBNF grammar with embedded Java code for building a syntax tree;
- one class for every form of syntax tree node; and
- a default visitor which can do a depth-first traversal of a syntax tree.

The produced EBNF grammar can then be processed by the Java Compiler Compiler to give a parser which produces syntax trees. The produced syntax trees can now be traversed by a Java program by writing subclasses of the default visitor.



Example

For example, consider the Java 1.1 production

```
void Assignment() : {}
   { PrimaryExpression() AssignmentOperator()
     Expression() }
```

JTB produces an annotated grammar where the production for `Assignment` is:

```
Assignment Assignment () :
{ PrimaryExpression n0;
  AssignmentOperator n1;
  Expression n2; {} }
{ n0=PrimaryExpression()
  n1=AssignmentOperator()
  n2=Expression()
  { return new Assignment(n0,n1,n2); }
}
```

Notice that the production returns a syntax tree represented as an `Assignment` object.

The produced syntax-tree-node class for `Assignment` is:

```
public class Assignment implements Node {
  PrimaryExpression f0; AssignmentOperator f1;
  Expression f2;

  public Assignment(PrimaryExpression n0,
                    AssignmentOperator n1,
                    Expression n2)
  { f0 = n0; f1 = n1; f2 = n2; }

  public void accept(visitor.Visitor v) {
      v.visit(this);
  }
}
```

Notice the `accept` method; it invokes the method `visit` for `Assignment` in the default visitor.

The default visitor looks like this:

```
public class Visitor {
  ...
   //
   // f0 -> PrimaryExpression()
   // f1 -> AssignmentOperator()
   // f2 -> Expression()
   //
   public void visit(Assignment n) {
      n.f0.accept(this);
      n.f1.accept(this);
      n.f2.accept(this);
   }
}
```

Notice the body of the method which visits each of the three subtrees of the `Assignment` node.

Here is an example of a program which operates on syntax trees for Java 1.1 programs. The program prints the right-hand side of every assignment. The entire program is six lines:

```
public class VprintAssignRHS extends Visitor {
   void visit(Assignment n) {
      VPrettyPrinter v = new VPrettyPrinter();
      n.f2.accept(v); v.out.println();
      n.f2.accept(this);
   }
}
```

When this visitor is passed to the root of the syntax tree, the depth-first traversal will begin, and when `Assignment` nodes are reached, the method `visit` in `VprintAssignRHS` is executed.

Notice the use of `VPrettyPrinter`. It is a visitor which pretty prints Java 1.1 programs.
JTB is bootstrapped.

6 The Scheme Language

Basics

Interaction loop.

```
3

(+ 1 3)

(a b c)

'(a b c)

(define x 3)

x

(+ x 1)

(define l (a b c))

(define l '(a b c))

(define u (+ x 1))
```

Procedures

Creating procedures with lambda:

```
(lambda (x) body)

(lambda (x) (+ x 1))

((lambda (x) (+ x 1)) 4)

(define mysucc (lambda (x) (+ x 1)))

(mysucc 4)

(define myplus (lambda (x y) (+ x y)))

(+ 3 4)
((lambda (x y) (+ x y)) 3 4)
(myplus 3 4)
```

Procedures can take other procedures as arguments:

```
((lambda (f x) (f x 3))
 myplus 5)
```

Procedures

Procedures can return other procedures; this is called Currying:

```
(define twice
  (lambda (f)
    (lambda (x)
      (f (f x)))))

(define add2 (twice (lambda (x) (+ x 1))))
```

Kinds of Data

Basic values = Symbols ∪ Numbers ∪ Strings ∪ Lists

Symbols: sequence of letters and digits starting with a letter. The sequence can also include other symbols, such as `-`, `$`, `=`, `*`, `/`, `?`, `_`. See the Scheme manual for details.

Numbers: integers, etc.

Strings: `"this is a string"`

Lists:

(1) the empty list is a list `()`

(2) a sequence `(`s_0 ... s_{n-1}`)` where each s_i is a value (either a symbol, number, string, or list

(3) nothing is a list unless it can be shown to be a list by rules (1) and (2).

This is an inductive definition, which will play an important part in our reasoning. We will often solve problems (e.g., write procedures on lists) by following this inductive definition. Examples of lists:

```
(turkey)
(atom)
((turkey) (xyz atom))
xyz -- a symbol, not a list
(xyz)
```

List Processing

Putting list literals in programs:

```
'(a b c)
'()
```

Basic operations on lists:

`car`: if `l` is `(`$s_0 \ \ldots \ s_{n-1}$`)`, then `(car l)` is s_0. The `car` of the empty list is undefined.

```
e.g. (define l '(a b c))    (car l) => a
```

`cdr`: if `l` is `(`$s_0 s_1 \ \ldots \ s_{n-1}$`)`, then `(cdr l)` is `(`$s_1 \ \ldots \ s_{n-1}$`)`. The `cdr` of the empty list is undefined.

```
(cdr l) => (b c)
```

Examples of car and cdr on previous lists.

Combining car and cdr:

```
(car (cdr l))
(cdr (cdr l))
```

etc.

Building Lists

`cons`: if `s` is the value s, and `l` is the list `(`$s_0 \ldots\ \ s_{n-1}$`)`, then `(cons s l)` is the list `(`$s\ \ s_0 \ldots\ \ s_{n-1}$`)`

Cons builds a list whose car is s and whose cdr is l.

```
(car (cons s l)) = s
(cdr (cons s l)) = l
```

```
cons : value * list -> list
car  : list -> value
cdr  : list -> list
```

Booleans

Literals:

`#t, #f`

Predicates:

```
(number? s)
(symbol? s)
(string? s)
(null? s)
(pair? s)
(eq? s1 s2)      --works on symbols
(= n1 n2)        --works on numbers
(zero? n)
(> n1 n2)
etc
```

Conditional:

```
(if bool e1 e2)
```

Recursive Procedures

Imagine we want to define a procedure to find powers, eg.

$$e(n, x) = x^n \ .$$

It is easy to define a bunch of procedures $e_0(x) = x^0$, $e_1(x) = x^1$, $e_2(x) = x^2$:

$$\begin{aligned} e_0(x) &= 1 \\ e_1(x) &= x \times e_0(x) \\ e_2(x) &= x \times e_1(x) \\ e_3(x) &= x \times e_2(x) \end{aligned}$$

and in general, we can define

$$e_n(x) = x \times e_{n-1}(x)$$

How did we do this? At each stage, we used the fact that we had already solved the problem for smaller n. This is the principle of mathematical induction.

How can we parameterize this construction to get a single procedure e?

If `n` is 0, we know the answer: `(e 0 x) = 1`.

If n is greater than 0, assume we can solve the problem for $n - 1$.

Then the answer is `(e n x) = (* x (e (- n 1) x))`.

Recursive Procedures

```
(define e
  (lambda (n x)
    (if (zero? n)
        1
        (* x
           (e (- n 1) x)))))
```

Why does this work? Let's prove it works for any n, by induction on n:

(1) It surely works for n=0.

(2) Now assume (for the moment) that it works when $\mathtt{n} = k$. Then it works when $\mathtt{n} = k + 1$. Why? Because then `(e n x) = (* x (e` k `x))`, and we know e works when its first argument is k. So it gives the right answer when its first argument is $k + 1$.

Structural Induction

- **The Moral:** If we can reduce the problem to a smaller subproblem, then we can call the procedure itself ("recursively") to solve the smaller subproblem.
- Then, as we call the procedure, we ask it to work on smaller and smaller subproblems, so eventually we will ask it about something that it can solve directly (eg n=0, the basis step), and then it will terminate successfully.
- **Principle of structural induction:** If you always recur on smaller problems, then your procedure is sure to work.

```
(e 3 2)
= (* 2 (e 2 2))
= (* 2 (* 2 (e 1 2)))
= (* 2 (* 2 (* 2 (e 0 2))))
= (* 2 (* 2 (* 2 1)))
= 8
```

Recursive Procedures

```
(define fact
  (lambda (n)
    (if (zero? n) 1 (* n (fact (- n 1))))))
```

```
   (fact 4)
= (* 4 (fact 3))
= (* 4 (* 3 (fact 2)))
= (* 4 (* 3 (* 2 (fact 1))))
= (* 4 (* 3 (* 2 (* 1 (fact 0)))))
= (* 4 (* 3 (* 2 (* 1 1))))
= (* 4 (* 3 (* 2 1)))
= (* 4 (* 3 2))
= (* 4 6)
= 24
```

- This is the natural recursive definition of factorial.
- Each call of fact is made with a promise that the value returned will be multiplied by the value of n at the time of the call.
- Thus fact is invoked in larger and larger control contexts as the calculation proceeds.

Recursive Procedures

Let us try a more classical programming style. In Java:

```
int fact(int n) {
  int a=1;
  while (n!=0) {
    a=n*a;
    n=n-1;
  }
  return a;
}
```

In Scheme:

```
(define fact-iter
  (lambda (n)
    (fact-iter-acc n 1)))

(define fact-iter-acc
  (lambda (n a)
    (if (zero? n)
        a
        (fact-iter-acc (- n 1) (* n a)))))
```

Recursive Procedures

```
(fact-iter 4)
= (fact-iter-acc 4 1)
= (fact-iter-acc 3 4)
= (fact-iter-acc 2 12)
= (fact-iter-acc 1 24)
= (fact-iter-acc 0 24)
= 24
```

- `fact-iter-acc` is always invoked in the same context (in this case, no context at all).
- When fact-iter-acc calls itself, it does so at the "tail end" of a call to fact-iter-acc.
- That is, no promise is made to do anything with the returned value other than return it as the result of the call to fact-iter-acc.
- Thus each step in the derivation above has the form `(fact-iter-acc` n a`)`.

Recursive Procedures

Procedure: `nth-elt`

Input: a list l and an integer k.

Output: Returns the k-th element of list l, (counting from 0). If l does not contain sufficiently many elements, an error is signaled.

When k is 0, we know the answer: `(car l)`. If k is greater than 0, then the answer is the $k-1$st element of `(cdr l)`. The only glitch is that we have to *guard* the `car` and `cdr` operations:

```
(define nth-elt
  (lambda (l k)
    (if (null? l)
      (error nth-elt "nth-elt:  ran off end")
      (if (zero? k)
        (car l)
        (nth-elt (cdr l) (- k 1))))))
```

```
(nth-elt '(a b c d e) 3)
= (nth-elt '(b c d e) 2)
= (nth-elt '(c d e) 1)
= (nth-elt '(d e) 0)
= d
```

Recursive Procedures

Procedure: `list-of-symbols?`

Input: a list l

Output: true if l consists entirely of symbols, false if it contains a non-symbol.

```
(define list-of-symbols?
  (lambda (l)
    (if (null? l)
        #t
        (if (symbol? (car l))
            (list-of-symbols? (cdr l))
            #f))))
```

Procedure: `member?`

Input: a symbol i and a list l of atoms.

Output: true if l contains the symbol i as one of its members, false otherwise.

```
(define member
  (lambda (i l)
    (if (null? l) #f
        (if (eqv? i (car l)) #t
            (member i (cdr l))))))
```

Recursive Procedures

Procedure: `remove-first`

Input: a symbol i and a list l of atoms.

Output: A list the same as l, except that the first occurrence of i has been removed. If there is no occurrence of i, the list will be unchanged.

```
(define remove-first
  (lambda (i l)
    (if (null? l) '()
        (if (eqv? i (car l))
            (cdr l)
            (cons (car l)
                  (remove-first i (cdr l)))))))
```

Recursive Procedures

Procedure: `remove`

Input: a symbol i and a list l of atoms.

Output: A list the same as l, except that *all* occurrences of i have been removed. If there is no occurrence of i, the list will be unchanged.

```
(define remove
  (lambda (i l)
    (if (null? l) '()
        (if (eqv? i (car l))
            (remove i (cdr l))
            (cons (car l)
                  (remove i (cdr l)))))))
```

Recursive Procedures

Procedure: `remove-all`

Input: An symbol i and a list l

Output: Like `remove`, except that all occurrences of i in l or any of its sublists (at any level) are to be removed.

```
(define remove-all
  (lambda (i l)
    (if (null? l) '()
        (if (atom? (car l))
            (if (eqv? i (car l))
                (remove-all i (cdr l))
                (cons (car l) (remove-all i (cdr l))))
            (cons (remove-all i (car l))
                  (remove-all i (cdr l)))))))
```

cond

In Scheme, a many-way branch is expressed using `cond`.

```
(cond
  (test1 exp1)
  (test2 exp2)
  ...
  (else exp_n))
```

So we can rewrite `remove-all` as

```
(define remove-all
  (lambda (i l)
    (cond
      ((null? l) '())
      ((atom? (car l))
       (if (eqv? i (car l))
           (remove-all i (cdr l))
           (cons (car l) (remove-all i (cdr l)))))
      (else (cons (remove-all i (car l))
                  (remove-all i (cdr l)))))))
```

let

When we need *local names*, we use the special form `let`:

```
(let ((var1 val1)
      (var2 val2)
      ...)
  exp)
```

For example, we write

```
(let ((n 25))
  (let ((x (sqrt n))
        (y 3))
    (+ x y)))

= (let ((x (sqrt 25))
        (y 3))
    (+ x y))

= (let ((x 5)
        (y 3))
    (+ x y))

= (+ 5 3)
= 8
```

let

```
(let ((x 3))
  (let ((y (+ x 4)))
    (* x y)))

= (let ((y (+ 3 4)))
    (* 3 y))

= (let ((y 7))
    (* 3 y))

= (* 3 7) = 21
```

compare to:

```
(let ((x 3)
      (y (+ x 4)))
  (* x y))
```

Here the occurrence of `x` on the second line refers to whatever the enclosing value of `x` happens to be– maybe set by another `let`.

let

So we can rewrite `remove-all` as

```
(define remove-all
  (lambda (i l)
    (cond
      ((null? l) '())
      ((atom? (car l))
       (let ((rember-cdr (remove-all i (cdr l))))
         (if (eqv? i (car l))
             rember-cdr
             (cons (car l) rember-cdr))))
      (else (cons (remove-all i (car l))
                  (remove-all i (cdr l)))))))
```

`Let`'s can, of course, be nested.

let

Using `let`, we can give names to local procedures and pass them around:

```
(let ((f (lambda (y z) (* y (+ y z))))
      (g (lambda (u) (+ u 5))))
  (f (g 3) 17))

(let ((x 5))
  (let ((f (lambda (y z) (* y (+ x z))))
        (g (lambda (u) (+ u x)))
        (x 38))
    (f (g 3) 17)))

(let ((f (lambda (y) (y (y 5))))
      (g (lambda (z) (+ z 8))))
  (f g))

(let ((f (lambda (y z) (y (y z))))
      (g (lambda (z) (+ z 8))))
  (f g 5))
```

Local Recursive Procedures

Lots of times we'd like to define recursive procedures locally, like `fact-iter` above.

```
(define fact-iter
  (lambda (n)
    (fact-iter-acc n 1)))

(define fact-iter-acc
  (lambda (n a)
    (if (zero? n) a (fact-iter-acc (- n 1)
                                   (* n a)))))
```

You'd like to write

```
(define fact-iter
  (lambda (n)
    (let ((fact-iter-acc
            (lambda (n a)
              (if (zero? n)
                  a
                  (fact-iter-acc (- n 1)
                                 (* n a))))))
      (fact-iter-acc n 1))))
```

Local Recursive Procedures

The scope of `fact-iter-acc` doesn't include it's definition. Instead, we can use `letrec`:

```
(letrec
 ((name1 proc1)
  (name2 proc2)
  ...)
 body)
```

`letrec` creates a set of mutually recursive procedures and makes their names available in the body.

So we can write:

```
(define fact-iter
  (lambda (n)
    (letrec ((fact-iter-acc
               (lambda (n a)
                 (if (zero? n) a
                   (fact-iter-acc (- n 1)
                                  (* n a))))))
      (fact-iter-acc n 1))))
```

Local Recursive Procedures

Here's another example:

```
(define remove
  (lambda (i l)
    (if (null? l) '()
      (if (eq? i (car l))
        (remove i (cdr l))
        (cons (car l)
              (remove i (cdr l)))))))
```

can be replaced by

```
(define remove
  (lambda (i l)
    (letrec
      ((loop
         ;; (loop l) = (remove i l)
         (lambda (l)
           (if (null? l) '()
             (if (eq? i (car l))
               (loop (cdr l))
               (cons (car l)
                     (loop (cdr l))))))))
      (loop l))))
```

Local Recursive Procedures

Another example:

```
(define linear-search
  (lambda (pred list fn)
    (letrec
      ((loop
           ;; (loop list) =
           ;;   (linear-search pred list fn)
         (lambda (list)
               (cond
                 ((null? list) #f)
                 ((pred (car list)) (fn (car list)))
                 (else (loop (cdr list)))))))
      (loop list))))
```

Sequencing

Want to have some finer control over the order in which things are done in Scheme. We need this for worrying about (a) side-effects and (b) termination. In general, there is only one rule about sequencing in Scheme:

Arguments are evaluated before procedure bodies.

So in

```
((lambda (x y z) body) exp1 exp2 exp3)
```

`exp1`, `exp2`, and `exp3` are guaranteed to be evaluated before `body`, but we don't know in what order `exp1`, `exp2`, and `exp3` are going to be evaluated, but they will all be evaluated before `body`.
This is **precisely** the same as

```
(let ((x exp1) (y exp2) (z exp3))
  body)
```

In both cases, we evaluate `exp1`, `exp2`, and `exp3`, and then we evaluate `body` in an environment in which `x`, `y`, and `z` are bound to the values of `exp1`, `exp2`, and `exp3`.

Data Types and their Representations in Scheme

1. Want to define new data types. This comes in two pieces: a *specification* and an *implementation.* The specification tells us *what* data (and what operations on that data) we are trying to represent. The implementation tells us *how* we do it.

2. We want to arrange things so that you can change the implementation without changing the code that uses the data type (user = client; implementation = supplier/server).

3. Both the specification and implementation have to deal with two things: the data and the operations on the data.

4. Vital part of the implementation is the specification of how the data is represented. We will use the notation $\lceil v \rceil$ for "the representation of data v".

Numbers

Data Specification: the non-negative integers.

Specification of Operations:

$$\texttt{zero} = \lceil 0 \rceil$$

$$(\ \texttt{is-zero?}\ \lceil n \rceil\ ) = \begin{cases} \texttt{\#t} & n = 0 \\ \texttt{\#f} & n \neq 0 \end{cases}$$

$$(\ \texttt{succ}\ \lceil n \rceil\ ) = \lceil n+1 \rceil$$

$$(\ \texttt{pred}\ \lceil n+1 \rceil\ ) = \lceil n \rceil$$

Now we can write procedures to do the other arithmetic operations, and these will work no matter what representation of the natural numbers we use.

```
(define plus
  (lambda (x y)
    (if (is-zero? x) y
      (succ (plus (pred x) y)))))
```

will satisfy `( plus` $\lceil x \rceil$ $\lceil y \rceil$ `)` $= \lceil x+y \rceil$, no matter what implementation of the integers we use.

Implementations of Numbers

1. Unary representation:

$$\lceil 0 \rceil = \texttt{()}$$
$$\lceil n+1 \rceil = \texttt{(cons \#t } \lceil n \rceil\texttt{)}$$

So the integer n is represented by a list of n `#t`'s.

It's easy to see that we can satisfy the specification by writing:

```
(define zero '())

(define is-zero? null?)

(define succ
  (lambda (n) (cons #t n)))

(define pred cdr)
```

2. Scheme number representation:

$\lceil n \rceil$ = the Scheme integer `n`

```
(define zero 0)
(define is-zero? zero?)
(define succ (lambda (n) (+ n 1)))
(define prec (lambda (n) (- n 1)))
```

Finite Functions

Data: A function whose domain is a finite set of Scheme symbols, and whose range is unspecified: $\{(s_1, v_1), \ldots, (s_n, v_n)\}$.

Specification of Operations:

Interface:

```
;;; interface finite-function =
;;;   type ff
;;;   empty-ff : ff
;;;   apply-ff : ff * key -> value
;;;   extend-ff : key * val * ff -> ff
;;;
```

$$
\begin{array}{ll}
\texttt{empty-ff} & = \lceil \emptyset \rceil \\
\texttt{(apply-ff } \lceil f \rceil \; s \texttt{)} & = f(s) \\
\texttt{(extend-ff } s \; v \; \lceil f \rceil \texttt{)} & = \lceil g \rceil, \\
 & \text{where } g(s') = \begin{cases} v & s' = s \\ f(s') & \text{otherwise} \end{cases}
\end{array}
$$

This tells us what procedures are available for dealing with finite functions: it defines the *interface*. The interface gives the type of each procedure and a description of the intended behavior of each procedure.

Procedural Representation

```
;;; uses ff = key -> value
```

`f` $= \lceil\{(s_1, v_1), \ldots, (s_n, v_n)\}\rceil$ iff `(f` s_i`)` $= v_i$.
Implement the operations by:

```
(define apply-ff
  (lambda (ff z) (ff z)))

(define empty-ff
  (lambda (z)
    (error 'env-lookup
           (format "couldn't find ~s" z))))

(define extend-ff
  (lambda (key val ff)
    (lambda (z)
      (if (eq? z key)
        val
        (apply-ff ff z)))))
```

Procedural Representation

```
;;; Examples

> (define ff-1 (extend-ff 'a 1 empty-ff))
> (define ff-2 (extend-ff 'b 2 ff-1))
> (define ff-3 (extend-ff 'c 3 ff-2))
> (define ff-4 (extend-ff 'd 4 ff-3))
> (define ff-5 (extend-ff 'e 5 ff-4))
> ff-5
<Procedure>
> (apply-ff ff-5 'd)
4
> (apply-ff empty-ff 'c)
error in env-lookup: couldn't find c.
> (apply-ff ff-3 'd)
error in env-lookup: couldn't find d.
```

Association-list Representation

$\lceil\{(s_1, v_1), \ldots, (s_n, v_n)\}\rceil =$ `((`s_1 `.` v_1`) ... (`s_n `.` v_n`))`

```
(define empty-ff '())

(define extend-ff
  (lambda (key val ff)
    (cons (cons key val) ff)))

(define apply-ff
  (lambda (alist z)
    (if (null? alist)
      (error 'env-lookup
             (format "couldn't find ~s" z))
      (let ((key (caar alist))
            (val (cdar alist))
            (ff (cdr alist)))
        (if (eq? z key) val (apply-ff ff z))))))
```

```
> (define ff-1 (extend-ff 'a 1 empty-ff))
> (define ff-2 (extend-ff 'b 2 ff-1))
> (define ff-3 (extend-ff 'c 3 ff-2))
> (define ff-4 (extend-ff 'd 4 ff-3))
> ff-4
((d . 4) (c . 3) (b . 2) (a . 1))
```

7 An Object-Oriented Style in Scheme

The style is based on three ideas:

- Sharing of bindings.
- Procedures stored in variables.
- A let expression which returns a procedure.

A Java Program

```
interface StackInterface {
  boolean empty();
  void push(Object x);
  void pop();
  Object top();
}

class Stack implements StackInterface {
  Object [] s = new Object[1000];
  int sp = -1;
  public boolean empty() {
    return sp == -1;
  }
  public void push(Object x) {
    sp=sp+1; s[sp]=x;
  }
  public void pop() {
    sp=sp-1;
  }
  public Object top() {
    return s[sp];
  }
}
```

Sharing of Bindings

```
> (define empty? '*)
> (define push! '*)
> (define pop! '*)
> (define top '*)
> (let ((stk '()))
    (set! empty?  (lambda ()  (null? stk)))
    (set! push!   (lambda (x) (set! stk (cons x stk))))
    (set! pop!    (lambda ()  (if (empty?)
                                  (error "Stack empty")
                                  (set! stk (cdr stk)))))
    (set! top     (lambda ()  (if (empty?)
                                  (error "Stack empty")
                                  (car stk)))))

> (push! 1)
> (push! 2)
> (top)
2
> (pop!)
> (top)
1
> (pop!)
```

The four procedures `empty?`, `push!`, `pop!`, and `top` share the same `stk` binding. Thus, the stack is a data abstraction.

Let returns a Procedure

```
(define stack
  (let ((stk '()))
    (lambda (message)
      (case message
        ((empty?) (lambda ()  (null? stk)))
        ((push!)  (lambda (x) (set! stk (cons x stk))))
        ((pop!)   (lambda ()  (if (null? stk)
                                  (error "Stack empty")
                                  (set! stk (cdr stk)))))
        ((top)    (lambda ()  (if (null? stk)
                                  (error "Stack empty")
                                  (car stk))))
        (else (error
                (format "stack: does not understand : ~s"
                        message)))))))

  > ((stack 'push!) 1)
  > ((stack 'push!) 2)
  > ((stack 'top))
  2
  > ((stack 'pop!))
  > ((stack 'top))
  1
  > ((stack 'pop!))
```

Note that the `lambda` is *inside* the `let`. This `stack` is much like an object in object-oriented programming.

Classes and Objects

The following procedure generates stacks.

```
(define make-stack
  (lambda ()
    (let ((stk '()))
      (lambda (message)
        (case message
          ((empty?) (lambda ()  (null? stk)))
          ((push!)  (lambda (x) (set! stk (cons x stk))))
          ((pop!)   (lambda ()  (if (null? stk)
                                    (error "Stack empty")
                                    (set! stk (cdr stk)))))
          ((top)    (lambda ()  (if (null? stk)
                                    (error "Stack empty")
                                    (car stk))))
          (else (error
                  (format "stack: does not understand : "
                          message)))))))))
```

```
> (define s1 (make-stack))
> (define s2 (make-stack))
> ((s1 'push!) 1)
> ((s2 'push!) 2)
> ((s1 'top))
1
> ((s2 'top))
2
```

Each time `make-stack` is invoked, a new stack procedure, with its own `stk` binding, is returned. This gives a simple form of classes, as known from object-oriented programming.