CS 538
Introduction to the Theory and Design of Programming Languages
Charles N. Fischer
Spring 2006

http://www.cs.wisc.edu/~fischer/cs538.html

Class Meets
Mondays, Wednesdays & Fridays,
1:20 — 2:10
1325 Computer Sciences

Instructor
Charles N. Fischer
6367 Computer Sciences
Telephone: 262-6635
E-mail: fischer@cs.wisc.edu
Office Hours:
10:30 - Noon, Tuesdays & Thursdays, or by appointment

Key Dates
- February 22: Homework #1 (tentative)
- March 8: Midterm Exam (tentative)
- March 22: Programming Assignment #1 - Scheme (tentative)
- April 14: Programming Assignment #2 - Standard ML (tentative)
- April 28: Programming Assignment #3 - Prolog (tentative)
- May 5: Programming Assignment #4 - Java, C#, Pizza and Python (optional)
- May 12: Final Exam 7:45am-9:45am

Class Text
- Required text:
  “Foundations of Programming Languages,”
  Seyed Roosta,

- Suggested supplemental Class Text:
  “Modern Programming Languages,”
  Adam Webber,

- Handouts and Web-based reading will also be used.
Reading Assignment
- Roosta: Chapters 1-3 (as background)
- Webber: Chapters 1, 10, 18 (as background)

Class Notes
- Each lecture will be made available prior to, and after, that lecture on the class Web page (under the “Lecture Nodes” link).

Instructional Computers
Departmental Linux Machines (royal1-royal30, emperor01-emperor40) have been assigned to CS 538. All necessary compiler, interpreters and tools will be loaded onto these machines.

You may also use your own PC or workstation. It will be your responsibility to load needed software (instructions on where to find needed software are included on the class web page).

The Systems Lab teaches brief tutorials on Linux if you are unfamiliar with that OS.

Academic Misconduct Policy
- You must do your own assignments — NO copying or sharing of solutions.
- You may discuss general concepts and ideas.
- All cases of misconduct must be reported to the Dean's office.
- Penalties may be severe.

Program & Homework Late Policy
- An assignment may be handed in up to 7 days late, but no later.
- Each day late will be debited 4%, up to a maximum of 28%.
- All students are given 10 “free” late days. That is, the first 40% in late debits will be automatically forgiven.
- Your 10 free late days may be used at any time, and in any combination.
### Approximate Grade Weights

<table>
<thead>
<tr>
<th>Assignment</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>Homework 1</td>
<td>10%</td>
</tr>
<tr>
<td>Program 1 - Scheme</td>
<td>16%</td>
</tr>
<tr>
<td>Program 2 - ML</td>
<td>16%</td>
</tr>
<tr>
<td>Program 3 - Prolog</td>
<td>12%</td>
</tr>
<tr>
<td>Program 4 - Java, C#, Pizza and Python (optional)</td>
<td>10%</td>
</tr>
<tr>
<td>Midterm Exam</td>
<td>23%</td>
</tr>
<tr>
<td>Final Exam (non-cumulative)</td>
<td>23%</td>
</tr>
</tbody>
</table>

### Programming Languages to be Considered in Detail

1. **Scheme**
   - A modern variant of Lisp.
   - A Functional Language: Functions are “first class” data values.
   - Dynamically Typed: A variable’s type may change during execution; no type declarations are needed.
   - All memory allocation and deallocation is automatic.
   - Primary data structures, lists and numbers, are unlimited in size and may grow without bound.
   - Continuations provide a novel way to suspend and “re-execute” computations.

2. **ML (“Meta Language”)**
   - Strong, compile-time type checking.
   - Types are determined by inference rather than declaration.
   - Naturally polymorphic (one function declaration can be used with many different types).
   - Pattern-directed programming (you define patterns that are automatically matched during a call).
   - Typed exceptions are provided.
   - Abstract data types, with constructors, are included.

3. **Prolog (Programming in Logic)**
   - Programs are Facts and Rules.
   - Programmers are concerned with definition, not execution.
   - Execution order is automatically determined.

4. **Pizza**
   - Extends a popular Object-oriented language, Java, to include
     - Parametric polymorphism (similar to C++’s templates).
     - First-class functional objects.
     - Algebraic data types, including patterns.
5. C#
Microsoft’s answer to Java. In most ways it is very similar to Java, with some C++ concepts reintroduced and some useful additions.
- Events and delegates are included to handle asynchronous actions (like keyboard or mouse actions).
- Properties allow user-defined read and write actions for fields.
- Indexers allow objects other than arrays to be indexed.
- Collection classes may be directly enumerated:
  ```csharp
  foreach (int i in array) ...
  ```
- Structs and classes co-exist and may be inter-converted (boxed and unboxed).

6. Java 1.5 (Tiger Java)
Extends current definition of Java to include:
- Parametric polymorphism (collection types may be parameterized).
- Enhanced loop iterators.
- Automatic boxing and unboxing of wrapper classes.
- Typesafe enumerations.
- Static imports (`out.println` rather than `System.out.println`).
- Variable argument methods.
- Formatted output using `printf`:
  ```java
  out.printf("Ans = %3d",a+b);
  ```

7. Python
A simple, efficient scripting language that quickly builds new programs out of existing applications and libraries. It cleanly includes objects. It scales nicely into larger applications.
Evolution of Programming Languages

In the beginning, ...

programs were written in absolute machine code—a sequence of bits that encode machine instructions.

Example:

34020005
0000000c
3c011001
ac220000

This form of programming is

• Very detailed
• Very tedious
• Very error-prone
• Very machine specific

Symbolic Assemblers

Allow use of symbols for operation codes and labels.

Example:

li $v0,5
syscall
sw $v0,a

Far more readable, but still very detailed, tedious and machine-specific.

Types are machine types.

Control structures are conditional branches.

Subprograms are blocks of code called via a “subroutine branch” instruction.

All labels are global.

Fortran (Formula Translator)

Example:

do 10 i=1,100
10 a(i)=0

Developed in the mid-50s.

A major step forward:

• Programming became more “problem oriented” and less “machine oriented.”
• Notions of control structures (ifs and do loops) were introduced.
• Subprograms, calls, and parameters were made available.
• Notions of machine independence were introduced.
• Has evolved into many new variants, including Fortran 77, Fortran 90 and HPF (High Performance Fortran).

Cobol (Common Business Oriented Language)

Example:

multiply i by 3 giving j.
move j to k.
write line1 after advancing 1 lines.

Developed in the early 60s.

The first widely-standardized programming language.

Once dominant in the commercial world; still important.

Wordy in structure; designed for non-scientific users.

Raised the issue of who should program and how important readability and maintainability are.
**Algol 60 (Algorithmic Language)**

Example:

```algol
real procedure cheb(x,n);
value x,n;
real x; integer n;
cheb := if n = 0 then 1
else if n = 1 then x
else 2 \times x \times
    cheb(x,n-1) - cheb(x,n-2);
```

Developed about 1960.
A direct precursor of Pascal, C, C++ and Java.
Introduced many ideas now in wide use:
- Blocks with local declarations and scopes.
- Nested declarations and control structures.

- Parameter passing
- Automatic recursion.

But,
- I/O wasn't standardized.
- IBM promoted Fortran and PL/I.

**Lisp (List Processing Language)**

Example:

```lisp
(((\lambda (x) (* x x)) 10)
```

Developed in the early 60s.
A radical departure from earlier programming languages.
Programs and data are represented in a uniform list format.
Types are a property of data values, not variables or parameters.
A program can build and run new functions as it executes.
Data values were not fixed in size.
Memory management was automatic.
A formal semantics was developed to define precisely what a program means.

**Simula 67 (Simulation Algol)**

Example:

```simula
Class Rectangle (Width, Height);
Real Width, Height;
Boolean Procedure IsSquare;
    IsSquare := Width=Height;
End of Rectangle;
```

Developed about 1967.
Introduced the notion of a class (for simulation purposes).
Included objects, a garbage collector, and notions of extending a class.
C++ was originally C with classes (as Simula was Algol with classes).
C and C++

C was developed in the early 70's; C++ in the mid-80s.
These languages have a concise, expressive syntax; they generate high quality code sufficient for performance-critical applications.
C, along with Unix, provided the viability of platform-independent languages and applications.
C and C++ allow programmers a great deal of freedom in bending and breaking rules.
Raises the issue of whether one language can span both novice and expert programmers.

Interesting issue—if most statements and expressions are meaningful, can errors be readily detected?

```c
if (a=b)
a=0;
else a = 1;
```

Java

Developed in the late 90s.
Cleaner object-oriented language than C++.
Introduced notions of dynamic loading of class definitions across the Web.
Much stronger emphasis on secure execution and detection of run-time errors.
Extended notions of platform independence to system independence.

What Drives Research into New Programming Languages?

Why isn't C or C++ or C++ enough?
1. Curiosity
   What other forms can a programming language take?
   What other notions of programming are possible?
2. Productivity
   Procedural languages, including C, C++ and Java, are very detailed.
   Many source lines imply significant development and maintenance expenses.
3. Reliability
Too much low-level detail in programs greatly enhances the chance of minor errors. Even minor errors can raise significant problems in applications.

4. Security
Computers are entrusted with ever-increasing responsibilities. How can we know that a program is safe and reliable enough to trust?

5. Execution speed
Procedural languages are closely tied to the standard sequential model of instruction execution. We may need radically different programming models to fully exploit parallel and distributed computers.

Desirable Qualities in a Programming Language
Theoretically, all programming languages are equivalent (Why?)
If that is so, what properties are desirable in a programming language?

• It should be easy to use.
  Programs should be easy to read and understand.
  Programs should be simple to write, without unexpected pitfalls.
  It should be orthogonal, providing only one way to do each step or computation.
  Its notation should be natural for the application being programmed.

• The language should support abstraction.
  You can’t anticipate all needed data structures and operations, so adding new definitions easily and efficiently should be allowed.

• The language should support testing, debugging and verification.

• The language should have a good development environment.
  Integrated editors, compilers, debuggers, and version control are a big plus.

• The language should be portable, spanning many platforms and operating systems.
The language should be inexpensive to use:
Execution should be fast.
Memory needs should be modest.
Translation should be fast and modular.
Program creation and testing should be easy and cheap.
Maintenance should not be unduly cumbersome.
Components should be reusable.

Programming Paradigms

Programming languages naturally fall into a number of fundamental styles or paradigms.

Procedural Languages
Most of the widely-known and widely-used programming languages (C, Fortran, Pascal, Ada, etc.) are procedural in nature.
Programs execute statement by statement, reading and modifying a shared memory.
This programming style closely models conventional sequential processors linked to a random access memory (RAM).

Functional Languages
Lisp, Scheme and ML are functional in nature.
Programs are expressions to be evaluated.
Language design aims to minimize side-effects, including assignment.
Alternative evaluation mechanisms are possible, including
- Lazy (Demand Driven)
- Eager (Data Driven or Speculative)

Question:
Given
```plaintext
a = a + 1;
if (a > 10)
  b = 10;
else
  b = 15;
a = a * b;
```
Why can't 5 processors each execute one line to make the program run 5 times faster?
Object-Oriented Languages

C++, Java, Smalltalk, Pizza and Python are object-oriented. Data and functions are encapsulated into Objects. Objects are active, have persistent state, and uniform interfaces (messages or methods). Notions of inheritance and common interfaces are central. All objects that provide the same interface can be treated uniformly. In Java you can print any object that provides the `toString` method. You can iterate through the elements of any Java object that implements the `Enumeration` interface.

Subclassing allows you to extend or redefine part of an object’s behavior without reprogramming all of the object’s definition. Thus in Java, you can take a `Hashtable` class (which is fairly elaborate) and create a subclass in which an existing method (like `toString`) is redefined, or new operations are added.

Logic Programming Languages

Prolog notes that most programming languages address both the logic of a program (what is to be done) and the control flow of a program (how to do what you want).

A logic programming language, like Prolog, lets programmers focus on a program’s logic without concern for control issue. These languages have no real control structures, and little notion of “flow of control.” What results are programs that are unusually succinct and focused.

Example:

```
inOrder([ ]).
inOrder([ _ ]).
inOrder([a, b|c]) :- (a\<b),
inOrder([b|c]).
```

This is a complete, executable function that determines if a list is in order. It is naturally polymorphic, and is not cluttered with declarations, variables or explicit loops.
Review of Concepts from Procedural Programming Languages

- Declarations/Scope/Lifetime/Binding
  - Static/Dynamic
- Identifiers are declared, either explicitly or implicitly (from context of first use).
- Declarations bind type and kind information to an identifier. Kind specifies the grouping of an identifier (variable, label, function, type name, etc.)
- Each identifier has a scope (or range) in a program—that part of the program in which the identifier is visible (i.e., may be used).

- Data objects have a lifetime—the span of time, during program execution, during which the object exists and may be used.
- Lifetimes of data objects are often tied to the scope of the identifier that denotes them. The objects are created when its identifier’s scope is entered, and they may be deleted when the identifier’s scope is exited. For example, memory for local variables within a function is usually allocated when the function is called (activated) and released when the call terminates.
  In Java, a method may be loaded into memory when the object it is a member of is first accessed.

Properties of an identifier (and the object it represents) may be set at
- Compile-time
  - These are static properties as they do not change during execution.
  - Examples include the type of a variable, the value of a constant, the initial value of a variable, or the body of a function.
- Run-time
  - These are dynamic properties.
  - Examples include the value of a variable, the lifetime of a heap object, the value of a function’s parameter, the number of times a while loop iterates, etc.

Example: In Fortran
- The scope of an identifier is the whole program or subprogram.
- Each identifier may be declared only once.
- Variable declarations may be implicit. (Using an identifier implicitly declares it as a variable.)
- The lifetime of data objects is the whole program.
Block Structured Languages

- Include Algol 60, Pascal, C and Java.
- Identifiers may have a non-global scope. Declarations may be local to a class, subprogram or block.
- Scopes may nest, with declarations propagating to inner (contained) scopes.
- The lexically nearest declaration of an identifier is bound to uses of that identifier.

Binding of an identifier to its corresponding declaration is usually static (also called lexical), though dynamic binding is also possible. Static binding is done prior to execution—at compile-time.

Example (drawn from C):

```c
int x, z;
void A() {
    float x, y;
    print(x, y, z);
}
void B() {
    print(x, y, z);
}
```

Variations in these rules of name scoping are possible. For example, in Java, the lifetime of all class objects is from the time of their creation (via new) to the last visible reference to them.

Thus

```java
... Object o; ...
```

creates an object reference but does not allocate any memory space for \( o \). You need

```java
... Object o = new Object(); ...
```
to actually create memory space for \( o \).

Block Structure Concepts

- Nested Visibility
  No access to identifiers outside their scope.
- Nearest Declaration Applies
  Static name scoping.
- Automatic Allocation and Deallocation of Locals
  Lifetime of data objects is bound to the scope of the identifiers that denote them.
Dynamic Scoping

An alternative to static scoping is dynamic scoping, which was used in early Lisp dialects (but not in Scheme, which is statically scoped).

Under dynamic scoping, identifiers are bound to the dynamically closest declaration of the identifier. Thus if an identifier is not locally declared, the call chain (sequence of callers) is examined to find a matching declaration.

Example:

```java
int x;
void print() {
    write(x);
}
main () {
    bool x;
    print();
}
```

Under static scoping the `x` written in `print` is the lexically closest declaration of `x`, which is as an `int`.

Under dynamic scoping, since `print` has no local declaration of `x`, `print`'s caller is examined. Since `main` calls `print`, and it has a declaration of `x` as a `bool`, that declaration is used.

Dynamic scoping makes type checking and variable access harder and more costly than static scoping. (Why?)

However, dynamic scoping does allow a notion of an “extended scope” in which declarations extend to subprograms called within that scope.

Though dynamic scoping may seem a bit bizarre, it is closely related to virtual functions used in C++ and Java.

Virtual Functions

A function declared in a class, C, may be redeclared in a class derived from C. Moreover, for uniformity of redeclaration, it is important that all calls, including those in methods within C, use the new declaration.

Example:

```java
class C {
    void DoIt() {PrintIt();}
    void PrintIt() {
        println("C rules!");
    }
}
class D extends C {
    void PrintIt() {
        println("D rules!");
    }
    void TestIt() {DoIt();}
}
D dvar = new D();
dvar.TestIt();
D rules! is printed.
```
Scope vs. Lifetime

It is usually required that the lifetime of a run-time object at least cover the scope of the identifier. That is, whenever you can access an identifier, the run-time object it denotes better exist.

But,

it is possible to have a run-time object's lifetime exceed the scope of its identifier. An example of this is static or own variables.

In C:

```c
void p() {
    static int i = 0;
    print(i++);
}
```

Each call to `p` prints a different value of `i` (0, 1, ...) Variable `i` retains its value across calls.

Some languages allow an explicit binding of an identifier for a fixed scope:

```c
let
    id = val { type id = val;
    in
    statements
    end;
```

A declaration may appear wherever a statement or expression is allowed. Limited scopes enhance readability.

Structs vs. Blocks

Many programming languages, including C, C++, C#, Pascal and Ada, have a notion of grouping data together into structs or records.

For example:

```c
struct complex { float re, im; }
```

There is also the notion of grouping statements and declarations into blocks:

```c
{ float re, im;
  re = 0.0; im = 1.0;
}
```

Blocks and structs look similar, but there are significant differences:

- As originally designed, structs contain only data (no functions or methods).
- Structs can be dynamically created, in any number, and included in other data structures (e.g., in an array of structs).
- All fields in a struct are visible outside the struct.
Blocks are code,
- They can contain both code and data.
- Blocks can't be dynamically created during execution; they are "built into" a program.
- Locals in a block aren't visible outside the block.

By adding functions and initialization code to structs, we get classes—a nice blend of structs and blocks.
For example:

```cpp
class complex{
    float re, im;
    complex (float v1, float v2){
        re = v1; im = v2; }
}
```

Classes
- Class objects can be created as needed, in any number, and included in other data structure.
- They include both data (fields) and functions (methods).
- They include mechanisms to initialize themselves (constructors) and to finalize themselves (destructors).
- They allow controlled access to members (private and public declarations).

Type Equivalence in Classes
In C, C++ and Java, instances of the same struct or class are type-equivalent, and mutually assignable.
For example:

```cpp
class MyClass { ... }
MyClass v1, v2;
v1 = v2; // Assignment is OK
```

We expect to be able to assign values of the same type, including class objects.
However, sometimes a class models a data object whose size or shape is set upon creation (in a constructor).
Then we may not want assignment to be allowed.

```cpp
class Point {
    int dimensions;
    float coordinates[];
    Point () {
        dimensions = 2;
        coordinates = new float[2];
    }
    Point (int d) {
        dimensions = d;
        coordinates = new float[d];
    }
}
Point plane = new Point();
Point solid = new Point(3);
plane = solid; //OK in Java
```

This assignment is allowed, even though the two objects represent points in different dimensions.
Subtypes

In C++, C# and Java we can create subclasses—new classes derived from an existing class.

We can use subclasses to create new data objects that are similar (since they are based on a common parent), but still type-inequivalent.

Example:

```java
class Point2 extends Point {
    Point2() {super(2);}
}
class Point3 extends Point {
    Point3() {super(3);}
}
Point2 plane = new Point2();
Point3 solid = new Point3();
plane = solid; //Illegal in Java
```

Parametric Polymorphism

We can create distinct subclasses based on the values passed to constructors. But sometimes we want to create subclasses based on distinct types, and types can’t be passed as parameters. (Types are not values, but rather a property of values.)

We see this problem in Java, which tries to create general purpose data structures by basing them on the class `Object`. Since any object can be assigned to `Object` (all classes must be a subclass of `Object`), this works—at least partially.

```java
class LinkedList {
    Object value;
    LinkedList next;
    Object head() {return value;}
    LinkedList tail() {return next;}
    LinkedList(Object O) {
        value = O; next = null;
    }
    LinkedList(Object O, LinkedList L) {
        value = O; next = L;
    }
}
Using this class, we can create a linked list of any subtype of `Object`.
But,

- We can't guarantee that linked lists are type homogeneous (contain only a single type).
- We must cast `Object` types back into their “real” types when we extract list values.

We must use wrapper classes like `Integer` rather than `int` (because primitive types like `int` aren't objects, and aren't subclass of `Object`).

For example, to use `LinkedList` to build a linked list of ints we do the following:

```java
LinkedList l = new LinkedList(new Integer(123));
int i = ((Integer) l.head()).intValue();
```
This is pretty clumsy code. We'd prefer a mechanism that allows us to create a “custom version” of `LinkedList`, based on the type we want the list to contain.
We can’t just call something like `LinkedList(int)` or `LinkedList(Integer)` because types can’t be passed as parameters. Parametric polymorphism is the solution. Using this mechanism, we can use type parameters to build a “custom version” of a class from a general purpose class.

C++ allows this using its template mechanism. Tiger Java, the newest version of Java, also allows type parameters.

In both languages, type parameters are enclosed in “angle brackets” (e.g., `LinkedList<T>` passes `T`, a type, to the `LinkedList` class).

Thus we have

```java
class LinkedList<T> {
    T value; LinkedList<T> next;
    T head() {return value;}
    LinkedList<T> tail() {
        return next;
    }
    LinkedList(T O) {
        value = O; next = null;
    }
    LinkedList(T O, LinkedList<T> L) {
        value = O; next = L;
    }
}
LinkedList<int> l = new LinkedList(123);
int i = l.head();
```

Overloading and Ad-hoc Polymorphism

Classes usually allow overloading of method names, if only to support multiple constructors.
That is, more than one method definition with the same name is allowed within a class, as long as the method definitions differ in the number and/or types of the parameters they take.
For example,
```java
class MyClass {
    int f(int i) { ... }
    int f(float g) { ... }
    int f(int i, int j) { ... }
}
```

Overloading is sometimes called “ad hoc” polymorphism, because, to the programmer, it appears that one method can take a variety of different parameter types. This isn’t true polymorphism because the methods have different bodies; there is no sharing of one definition among different parameter types. There is no guarantee that the different definitions do the same thing, even though they share a common name.
Issues in Overloading

Though many languages allow overloading, few allow overloaded methods to differ only on their result types. (Neither C++ nor Java allow this kind of overloading, though Ada does). For example,

```cpp
class MyClass {
    int f() { ... }
    float f() { ... }
}
```

is illegal. This is unfortunate, since methods with the same name and parameters, but different result types, could be used to automatically convert result values to the type demanded by the context of call.

Why is this form of overloading usually disallowed?

It’s because overload resolution (deciding which definition to use) becomes much harder. Consider

```cpp
class MyClass {
    int f(int i, int j) { ... }
    float f(float i, float j) { ... }
    float f(int i, int j) { ... }
}
in
int a = f( f(1,2), f(3,4) );
```

which definitions of `f` do we use in each of the three calls? Getting the correctly answer can be tricky, though solution algorithms do exist.

Operator Overloading

Some languages, like C++ and C#, allow operators to be overloaded. You may add new definitions to existing operators, and use them on your own types. For example,

```cpp
class MyClass {
    int i;
    public:
        int operator+(int j) {
            return i+j; }
}
```

MyClass c;
int i = c+10;
int j = c.operator+(10);
int k = 10+c; // Illegal!

The expression 10+c is illegal because there is no definition of `+` for the types `int` and `MyClass&`. We can create one by using C++’s friend mechanism to insert a definition into `MyClass` that will have access to `MyClass`’s private data:

```cpp
class MyClass {
    int i;
    public:
        int operator+(int j) {
            return i+j; }
        friend int operator+ (int j, MyClass& v){
            return j+v.i; }
}
```

MyClass c;
int k = 10+c; // Now OK!
C++ limits operator overloading to existing predefined operators. A few languages, like Algol 68 (a successor to Algol 60, developed in 1968), allow programmers to define brand new operators.

In addition to defining the operator itself, it is also necessary to specify the operator's precedence (which operator is to be applied first) and its associativity (does the operator associate from left to right, or right to left, or not at all). Given this extra detail, it is possible to specify something like:

```cpp
op +++ prec = 8;
int op +++(int& i, int& j) {
    return (i++)+(j++);
}
```

(Why is int& used as the parameter type rather than int?)

Parameter Binding

Almost all programming languages have some notion of binding an actual parameter (provided at the point of call) to a formal parameter (used in the body of a subprogram).

There are many different, and inequivalent, methods of parameter binding. Exactly which is used depends upon the programming language in question.

Parameter Binding Modes include:

- **Value**: The formal parameter represents a local variable initialized to the value of the corresponding actual parameter.

- **Result**: The formal parameter represents a local variable. Its final value, at the point of return, is copied into the corresponding actual parameter.

- **Value/Result**: A combination of the value and results modes. The formal parameter is a local variable initialized to the value of the corresponding actual parameter. The formal's final value, at the point of return, is copied into the corresponding actual parameter.

- **Reference**: The formal parameter is a pointer to the corresponding actual parameter. All references to the formal parameter indirectly access the corresponding actual parameter through the pointer.

- **Name**: The formal parameter represents a block of code (sometimes called a thunk) that is evaluated to obtain the value or address of the corresponding actual parameter. Each reference to the formal parameter causes the thunk to be reevaluated.

- ** Readonly (sometimes called Const)**: Only reads of the formal parameter are allowed. Either a copy of the actual parameter's value, or its address, may be used.
What Parameter Modes do Programming Languages Use?

- C: Value mode except for arrays which pass a pointer to the start of the array.
- C++: Allows reference as well as value modes. E.g.,
  ```
  int f(int a, int& b)
  ```
- C#: Allows result (out) as well as reference and value modes. E.g.,
  ```
  int g(int a, out int b)
  ```
- Java: Scalar types (int, float, char, etc.) are passed by value; objects are passed by reference (references to objects are passed by value).
- Fortran: Reference (even for constants!)
- Ada: Value/result, reference, and readonly are used.

Example

```
void p(value int a,
      reference int b,
      name int c) {
  a=1; b=2; print(c)
}
int i=3, j=3, k[10][10];
p(i, j, k[i][j]);
```

What element of \( k \) is printed?

- The assignment to \( a \) does not affect \( i \), since \( a \) is a value parameter.
- The assignment to \( b \) does affect \( j \), since \( b \) is a reference parameter.
- \( c \) is a name parameter, so it is evaluated whenever it is used. In the print statement \( k[i][j] \) is printed. At that point \( i=3 \) and \( j=2 \), so \( k[3][2] \) is printed.

Why are there so Many Different Parameter Modes?

Parameter modes reflect different views on how parameters are to be accessed, as well as different degrees of efficiency in accessing and using parameters.

- Call by value protects the actual parameter value. No matter what the subprogram does, the parameter can’t be changed.
- Call by reference allows immediate updates to the actual parameter.
- Call by readonly protects the actual parameter and emphasizes the “constant” nature of the formal parameter.

- Call by value/result allows actual parameters to change, but treats a call as a single step (assign parameter values, execute the subprogram’s body, update parameter values).
- Call by name delays evaluation of an actual parameter until it is actually needed (which may be never).
**Call by Name**

Call by name is a special kind of parameter passing mode. It allows some calls to complete that otherwise would fail. Consider

$$f(i, j/0)$$

Normally, when \( j/0 \) is evaluated, a divide fault terminates execution. If \( j/0 \) is passed by name, the division is delayed until the parameter is needed, which may be never.

Call by name also allows programmers to create some interesting solutions to hard programming problems.

Consider the conditional expression found in C, C++, and Java:

$$(\text{cond} ? \text{value1} : \text{value2})$$

What if we want to implement this as a function call:

```c
condExpr(cond, value1, value2) {
    if (cond)
        return value1;
    else return value2;
}
```

With most parameter passing modes, this implementation won't work! (Why?)

But if \( \text{value1} \) and \( \text{value2} \) are passed by name, the implementation is correct.

---

**Call by Name and Lazy Evaluation**

Call by name has much of the flavor of lazy evaluation. With lazy evaluation, you don’t compute a value but rather a suspension—a function that will provide a value when called.

This can be useful when we need to control how much of a computation is actually performed.

Consider an infinite list of integers. Mathematically it is represented as

$$1, 2, 3, ...$$

How do we compute a data structure that represents an infinite list?

The obvious computation

```c
infList(int start) {
    return list(start, infList(start+1));
}
```

doesn’t work. (Why?)

A less obvious implementation, using suspensions, does work:

```c
infList(int start) {
    return list(start, function() {
        return infList(start+1);
    });
}
```

Now, whenever we are given an infinite list, we get two things: the first integer in the list and a suspension function. When called, this function will give you the rest of the infinite list (again, one more
value and another suspension function).
The whole list is there, but only as much as you care to access is actually computed.

**Eager Parameter Evaluation**

Sometimes we want parameters evaluated eagerly—as soon as they are known.

Consider a sorting routine that breaks an array in half, sorts each half, and then merges together the two sorted halves (this is a merge sort).

In outline form it is:

```java
sort(inputArray) {
  ... 
  merge(sort(leftHalf(inputArray)), sort(rightHalf(inputArray))); }
```

This definition lends itself nicely to parallel evaluation: The two halves of an input array can be sorted in parallel. Each of these two halves can again be split in two, allowing parallel sorting of four quarter-sized arrays,

then leading to 8 sorts of 1/8 sized arrays, etc.

But, to make this all work, the two parameters to merge must be evaluated eagerly, rather than in sequence.

**Type Equivalence**

Programming languages use types to describe the values a data object may hold and the operations that may be performed.

By checking the types of values, potential errors in expressions, assignments and calls may be automatically detected. For example, type checking tells us that

```
123 + "123"
```

is illegal because addition is not defined for an integer, string combination.

Type checking is usually done at compile-time; this is static typing.

Type checking may also be done at run-time; this is dynamic typing.
A program is type-safe if it is impossible to apply an operation to a value of the wrong type. In a type-safe language, plus is never told to add an integer to a string, because its definition does not allow that combination of operands. In type-safe programs an operator can still see an illegal value (e.g., a division by zero), but it can't see operands of the wrong type.

A strongly-typed programming language forbids the execution of type-unsafe programs.

Weakly-typed programming languages allow the execution of potentially type-unsafe programs.

The question reduces to whether the programming language allows programmers to “break” the type rules, either knowingly or unknowingly.

Java is strongly typed; type errors preclude execution. C and C++ are weakly typed; you can break the rules if you wish. For example:

```
int i; int* p;
p = (int *) i * i;
```

Now `p` may be used as an integer pointer though multiplication need not produce valid integer pointers.

If we are going to do type checking in a program, we must decide whether two types, T1 and T2 are equivalent; that is, whether they be used interchangeably.

There are two major approaches to type equivalence:

Name Equivalence:

Two types are equivalent if and only if they refer to exactly the same type declaration.

For example,

```
type PackerSalaries = int[100];
type AssemblySizes = int[100];
PackerSalaries salary;
AssemblySizes size;
```

Is

```
sal = size;
```

allowed?

Using name equivalence, no. That is, salary \( \not\equiv_N \) size since these two variables have different type declarations (that happen to be identical in structure).

Formally, we define \( \equiv_N \) (name type equivalence) as:

(a) \( T \equiv_N T \)

(b) Given the declaration

```
Type T1 = T2;
```

\( T1 \equiv_N T2 \)
We treat anonymous types (types not given a name) as an abbreviation for an implicit declaration of a new and unique type name.

Thus

```
int A[10];
```

is an abbreviation for

```
Type T_new = int[10];
T_new A;
```

---

**Structural Equivalence**

An alternative notion of type equivalence is structural equivalence (denoted $\equiv_S$). Roughly, two types are structurally equivalent if the two types have the same definition, independent of where the definitions are located. That is, the two types have the same definitional structure.

---

Formally,

(a) $T \equiv_S T$

(b) Given the declaration

```
Type T = Q;
T \equiv_S Q
```

(c) If $T$ and $Q$ are defined using the same type constructor and corresponding parameters in the two definitions are equal or structurally equivalent

then $T \equiv_S Q$

---

Returning to our previous example,

```
type PackerSalaries = int[100];
type AssemblySizes = int[100];
PackerSalaries salary;
AssemblySizes size;
```

salary $\equiv_S$ size since both are arrays and 100 = 100 and int $\equiv_S$ int.
Which notion of Equivalence do Programming Languages Use?

C and C++ use structural equivalence except for structs and classes (where name equivalence is used). For arrays, size is ignored.

Java uses structural equivalence for scalars. For arrays, it requires name equivalence for the component type, ignoring size. For classes it uses name equivalence except that a subtype may be used where a parent type is expected. Thus given

```java
void subr(Object O) { ... };
the call
subr(new Integer(100));
is OK since Integer is a subclass of Object.
```

Automatic Type Conversions

C, C++ and Java also allow various kinds of automatic type conversions.

In C, C++ and Java, a float will be automatically created from an int:

```c
float f = 10; // No type error
```

Also, an integer type (char, short, int, long) will be widened:

```c
int i = 'x';
```

In C and C++ (but not Java), an integer value can also be narrowed, possibly with the loss of significant bits:

```c
char c = 1000000;
```

Reading Assignment

- Roosta: Sections 13.1, 13.2
- The Scheme Language Definition (linked from class web page)

Lisp & Scheme

Lisp (List Processing Language) is one of the oldest programming languages still in wide use.

It was developed in the late 50s and early 60s by John McCarthy.

Its innovations include:

- Support of symbolic computations.
- A functional programming style without emphasis on assignments and side-effects.
- A naturally recursive programming style.
- Dynamic (run-time) type checking.
- Dynamic data structures (lists, binary trees) that grow without limit.
• Automatic garbage collection to manage memory.
• Functions are treated as “first class” values; they may be passed as arguments, returned as result values, stored in data structures, and created during execution.
• A formal semantics (written in Lisp) that defines the meaning of all valid programs.
• An Integrated Programming Environment to create, edit and test Lisp programs.

**Scheme**

Scheme is a recent dialect of Lisp.
It uses lexical (static) scoping.
It supports true first-class functions.
It provides program-level access to control flow via continuation functions.

**Atomic (Primitive) Data Types**

Symbols:
Essentially the same form as identifiers. Similar to enumeration values in C and C++.
Very flexible in structure; essentially any sequence of printable characters is allowed; anything that starts a valid number (except + or -) may not start a symbol.
Valid symbols include:
```abc hello-world + <=!```

Integers:
Any sequence of digits, optionally prefixed with a + or -. Usually unlimited in length.

**Reals:**
A floating point number in a decimal format (123.456) or in exponential format (1.23e45). A leading sign and a signed exponent are allowed (−12.3, 10.0e−20).

**Rationals:**
Rational numbers of the form integer/integer (e.g., 1/3 or 9/7) with an optional leading sign (−1/2, +7/8).

**Complex:**
Complex numbers of the form num+num i or num-num i, where num is an integer or real number. Example include 1+3i, −1.5−2.5i, 0+1i).
String:
A sequence of characters delimited by double quotes. Double quotes and backslashes must be escaped using a backslash. For example
"Hello World" "\Wow!"

Character:
A single character prefixed by #\. For example, #\a, #\0, #\, #\#. Two special characters are #\space and #\newline.

Boolean:
True is represented as #t and false is represented as #f.

Binary Trees
Binary trees are also called S-Expressions in Lisp and Scheme.
They are of the form ( item . item )
where item is any atomic value or any S-Expression. For example:
(A . B)
(1.2 . "xyz")
((A . B) . C)
(A . (B . C))
S-Expressions are linearizations of binary trees:

Lists
In Lisp and Scheme lists are a special, widely-used form of S-Expressions.
() represents the empty or null list
(A) represents the list containing A. By definition, (A) ≡ (A . () )
(A B) represents the list containing A and B. By definition,
(A B) ≡ (A . (B . ( ) ) )
In general, (A B C . . . Z) ≡
(A . (B . (C . . . (Z . ( ) . . . ) ) . . . ))
(A B C) ≡

A
B
C
()
**Function Calls**

In List and Scheme, function calls are represented as lists.

\[(A \ B \ C)\] means:
- Evaluate \(A\) (to a function)
- Evaluate \(B\) and \(C\) (as parameters)
- Call \(A\) with \(B\) and \(C\) as its parameters
- Then use the value returned by the call as the “meaning” of \((A \ B \ C)\).

\(\text{cons, car and cdr are predefined symbols bound to built-in functions that build and access lists and S-Expressions.}\)

\(\text{Literals (of type integer, real, rational, complex, string, character and boolean) evaluate to themselves.}\)

**For example** (⇒ means “evaluates to”)

\[
\begin{align*}
\text{(cons 1 2)} & \Rightarrow (1 . 2) \\
\text{(cons 1 () )} & \Rightarrow (1) \\
\text{(car (cons 1 2))} & \Rightarrow 1 \\
\text{(cdr (cons 1 ()) )} & \Rightarrow ()
\end{align*}
\]

But,

\[
\text{(car (1 2)) fails during execution!}
\]

Why?

The expression \((1 \ 2)\) looks like a call, but 1 isn’t a function! We need some way to “quote” symbols and lists we don’t want evaluated.

\(\text{(quote arg)}\)

is a special function that returns its argument unevaluated.

**User-defined Functions**

The list

\[
\text{(lambda (args) (body))}
\]

evaluates to a function with (args) as its argument list and (body) as the function body.

No quotes are needed for (args) or (body).

Thus

\[
\text{(lambda (x) (+ x 1)) evaluates to the increment function.}
\]

Similarly,

\[
\text{((lambda (x) (+ x 1)) 10) \Rightarrow 11}
\]

Thus

\[
\text{(quote (1 2)) doesn’t try to evaluate the list (1 2); it just returns it.}
\]

Since quotation is so often used, it may be abbreviated using a single quote. That is

\[
\text{(quote arg) \equiv ’arg}
\]

Thus

\[
\begin{align*}
\text{(car ’(a b c))} & \Rightarrow a \\
\text{(cdr ’( (A) (B) (C)) )} & \Rightarrow \\
& ( (B) (C) ) \\
\text{(cons ’a ’l)} & \Rightarrow (a . 1)
\end{align*}
\]

But,

\[
\text{’(cdr ’(A B)) fails!}
\]

Why?
We can bind values and functions to global symbols using the `define` function.

The general form is

```
(define id object)
```

`id` is not evaluated but `object` is. `id` is bound to the value `object` evaluates to.

For example,

```
(define pi 3.1415926535)
(define plus1 (lambda (x) (+ x 1)))
(define pi*2 (* pi 2))
```

Once a symbol is defined, it evaluates to the value it is bound to:

```
(plus1 12) => 13
```

Since functions are frequently defined, we may abbreviate

```
(define id (lambda (args) (body)))
```

as

```
(define (id args) (body))
```

Thus

```
(define (plus1 x) (+ x 1))
```

### Conditional Expressions in Scheme

A **predicate** is a function that returns a boolean value. By convention, in Scheme, predicate names end with “?”

For example,

```
number?  symbol?  equal?  null?  list?
```

In conditionals, `#f` is false, and everything else, including `#t`, is true.

The **if** expression is

```
(if pred E1 E2)
```

First `pred` is evaluated. Depending on its value (`#f` or `not`), either `E1` or `E2` is evaluated (but not both) and returned as the value of the **if** expression.

For example,

```
(if (= 1 (+ 0 1))
   'Yes
   'No
)
```

```
(define
  (fact n)
  (if (= n 0)
    1
    (* n (fact (- n 1)))
  )
)
```
Generalized Conditional

This is similar to a switch or case:

```
(cond
  (p1  e1)
  (p2  e2)
  ...
  (else en))
```

Each of the predicates \(p_1, p_2, \ldots\) is evaluated until one is true (\(\neq \#f\)). Then the corresponding expression \(e_1, e_2, \ldots\) is evaluated and returned as the value of the `cond`. `else` acts like a predicate that is always true.

Example:

```
(cond
  ((= a 1) 2)
  ((= a 2) 3)
  (else 4)
)
```

Recursion in Scheme

Recursion is widely used in Scheme and most other functional programming languages.

Rather than using a loop to step through the elements of a list or array, recursion breaks a problem on a large data structure into a simpler problem on a smaller data structure.

A good example of this approach is the `append` function, which joins (or appends) two lists into one larger list containing all the elements of the two input lists (in the correct order).

Note that `cons` is not `append`. `cons` adds one element to the head of an existing list.

Thus

```
(cons '(a b) '(c d)) \Rightarrow
((a b) c d)
(append '(a b) '(c d)) \Rightarrow
(a b c d)
```

The `append` function is predefined in Scheme, as are many other useful list-manipulating functions (consult the Scheme definition for what’s available).

It is instructive to define `append` directly to see its recursive approach:

```
(define
  (append L1 L2)
  (if  (null? L1)
    L2
    (cons (car L1)
      (append (cdr L1) L2))
  )
)
```

Let’s trace `(append '(a b) '(c d))`

Our definition is

```
(define
  (append L1 L2)
  (if  (null? L1)
    L2
    (cons (car L1)
      (append (cdr L1) L2))
  )
)
```

Now \(L1 = (a b)\) and \(L2 = (c d)\).

`(null? L1)` is false, so we evaluate

```
(cons (car L1) (append (cdr L1) L2))
= (cons (car '(a b))
  (append (cdr '(a b)) '(c d)))
= (cons 'a (append '(b) '(c d)))
```

We need to evaluate

```
(append '(b) '(c d))
```

In this call, \(L1 = (b)\) and \(L2 = (c d)\).

\(L1\) is not null, so we evaluate

```
(cons (car L1) (append (cdr L1) L2))
= (cons (car '(b))
  (append (cdr '(b)) '(c d)))
```
We need to evaluate

\[(\text{append} \ '() \ '(c d))\]

In this call, \(L_1 = ()\) and \(L_2 = (c d)\).

Therefore

\[\text{(cons 'b (append '() '(c d)) = (cons 'b '(c d)) = (b c d) = (append 'b '(c d))}\]

Finally,

\[(\text{append} \ '(a b) \ '(c d)) =
(\text{cons} \ 'a \ (\text{append} \ '(b) \ '(c d)) =
(\text{cons} \ 'a \ '(b c d)) = (a b c d)\]

Note:
Source files for append, and other Scheme examples, may be found in
~cs538-1/public/scheme/example1.scm,
~cs538-1/public/scheme/example2.scm, etc.

Reversing a List

Another useful list-manipulation function is \(\text{rev}\), which reverses the members of a list. That is, the last element becomes the first element, the next-to-last element becomes the second element, etc.

For example,

\[(\text{rev} \ '(1 2 3)) \implies (3 2 1)\]

The definition of \(\text{rev}\) is straightforward:

\[
\begin{align*}
(\text{define} \ (\text{rev} L) & \quad \text{if} \ (\text{null?} L) \quad L \\
& \quad \text{else} \quad \text{(append} \ (\text{rev} \ (\text{cdr} L)) \text{(list} \ (\text{car} L)) \text{)} \quad \\
\end{align*}
\]

As an example, consider

\[(\text{rev} \ '(1 2))\]

Here \(L = (1 2)\). \(L\) is not null so we evaluate

\[(\text{append} \ (\text{rev} \ (\text{cadr} L)) \text{(list} \ (\text{car} L))\)]

\[(\text{append} \ (\text{rev} \ (\text{cadr} '1 2)) \text{(list} \ (\text{car} '1 2))\)]

\[(\text{append} \ (\text{rev} '2) \text{(list} 1)\)
(\text{append} \ (\text{rev} '2) \text{'1} \)]

We must evaluate \((\text{rev} '2)\)

Here \(L = (2)\). \(L\) is not null so we evaluate

\[(\text{append} \ (\text{rev} \ (\text{cadr} L)) \text{(list} \ (\text{car} L))\)]

\[(\text{append} \ (\text{rev} \ (\text{cadr} '2)) \text{(list} \ (\text{car} '2))\)]

\[(\text{append} \ (\text{rev} '2) \text{'(1} \)]

We must evaluate \((\text{rev} '2)\)

Here \(L = ()\). \(L\) is null so \((\text{rev} '()) = ()\)

Thus \(\text{(append} \ (\text{rev} '()) '2)\) =

\(\text{(append} () '2) = (2) = (\text{rev} '2)\)

Finally, recall \((\text{rev} '1 2)) =

\(\text{(append} \ (\text{rev} '1 2) '1) =
(\text{append} '2) '1) = (2 1)\)

As constructed, \(\text{rev}\) only reverses the “top level” elements of a list. That is, members of a list that themselves are lists aren’t reversed.

For example,

\[(\text{rev} \ '1 2 3 4) =
((3 4) (1 2))\]

We can generalize \(\text{rev}\) to also reverse list members that happen to be lists.

To do this, it will be convenient to use Scheme’s \text{let} construct.
The Let Construct

Scheme allows us to create local names, bound to values, for use in an expression.

The structure is

\[(let \ ( \ (id1 \ val1) \ (id2 \ val2) \ldots ) \ expr \)\]

In this construct, \(val1\) is evaluated and bound to \(id1\), which will exist only within this let expression. If \(id1\) is already defined (as a global or parameter name) the existing definition is hidden and the local definition, bound to \(val1\), is used. Then \(val2\) is evaluated and bound to \(id2\), ... Finally, \(expr\) is evaluated in a scope that includes \(id1, id2, \ldots\)

For example,

\[
(let \ ( \ (a \ 10) \ (b \ 20) ) \ (+ \ a \ b)) \Rightarrow 30
\]

Using a let, the definition of \(\text{revall}\), a version of \(\text{rev}\) that reverses all levels of a list, is easy:

\[
(\text{define} \ (\text{revall} \ L) \\
(\text{if} \ (\text{null?} \ L) \ L \\
(\text{let} \ ((E \ (\text{if} \ (\text{list?} \ (\text{car} \ L)) \ \\
(\text{revall} \ (\text{car} \ L)) \\
(\text{car} \ L)))) \\
(\text{append} \ (\text{revall} \ (\text{cdr} \ L)) \\
(\text{list} \ E)) \\
))
\)

\((\text{revall} \ '\ (\ 1 \ 2 \ ) \ (3 \ 4)) \Rightarrow \ ((4 \ 3) \ (2 \ 1))\)

Subsets

Another good example of Scheme's recursive style of programming is subset computation.

Given a list of distinct atoms, we want to compute a list of all subsets of the list values.

For example,

\[
(\text{subsets} \ '\ (1 \ 2 \ 3)) \Rightarrow \\
( () \ (1) \ (2) \ (3) \ (1 \ 2) \ (1 \ 3) \ (2 \ 3) \ (1 \ 2 \ 3))
\]

The order of atoms and sublists is unimportant, but all possible subsets of the list values must be included.

Given Scheme's recursive style of programming, we need a recursive definition of subsets.

That is, if we have a list of all subsets of \(n\) atoms, how do we extend this list to one containing all subsets of \(n+1\) values?

First, we note that the number of subsets of \(n+1\) values is exactly twice the number of subsets of \(n\) values.

For example,

\[
(\text{subsets} \ '\ (1 \ 2)) \Rightarrow \\
( () \ (1) \ (2) \ (1 \ 2)), \text{which contains 4 subsets.}
\]

\((\text{subsets} \ '\ (1 \ 2 \ 3)) \text{ contains 8 subsets (as we saw earlier).}
\]

Moreover, the extended list (of subsets for \(n+1\) values) is simply the list of subsets for \(n\) values plus the result of "distributing" the new value into each of the original subsets.
Thus \((\text{subsets } '(1 2 3)) \Rightarrow (()\ (1)\ (2)\ (3)\ (1 2)\ (1 3)\ (2 3)\ (1 2 3)) = ( ()\ (1)\ (2)\ (1 2) ) \text{ plus} (3)\ (1 3)\ (2 3)\ (1 2 3)\)

This insight leads to a concise program for subsets.

We will let \((\text{distrib } L\ E)\) be a function that “distributes” \(E\) into each list in \(L\).

For example,
\[
\text{(distrib '(() (1) (2) (1 2)) 3) = ( (3) (3 1) (3 2) (3 1 2) )}
\]

\[
\text{(define (distrib L E)}
\]
\[
\text{ (if (null? L)}
\]
\[
\text{ ()
}\]
\[
\text{ (cons (cons E (car L))}
\]
\[
\text{ (distrib (cdr L) E))}
\]
\]

We will let \((\text{extend } L\ E)\) extend a list \(L\) by distributing element \(E\) through \(L\) and then appending this result to \(L\).

\[
\text{(define (extend L E)}
\]
\[
\text{ (append L (distrib L E))}
\]
\]

Now subsets is easy:

\[
\text{(define (subsets L)}
\]
\[
\text{ (if (null? L)}
\]
\[
\text{ ()
}\]
\[
\text{ (list ()}
\]
\[
\text{ (extend (subsets (cdr L))}
\]
\[
\text{ (car L))}
\]
\]

Data Structures in Scheme

In Scheme, lists and S-expressions are basic. Arrays can be simulated using lists, but access to elements “deep” in the list can be slow (since a list is a linked structure).

To access an element deep within a list we can use:

- \((\text{list-tail } L\ k)\)
  This returns list \(L\) after removing the first \(k\) elements. For example,
  \[
  (\text{list-tail } '(1 2 3 4 5) 2) \Rightarrow (3 4 5)
  \]
- \((\text{list-ref } L\ k)\)
  This returns the \(k\)-th element in \(L\) (counting from 0). For example,
  \[
  (\text{list-ref } '(1 2 3 4 5) 2) \Rightarrow 3
  \]

Vectors in Scheme

Scheme provides a vector type that directly implements one dimensional arrays.

Literals are of the form \(#(\ldots)\)

For example, \(#(1 2 3)\) or \(#(1 2.0 "three")\)

The function \((\text{vector? } val)\) tests whether \(val\) is a vector or not.

\[
\text{(vector? 'abc) \Rightarrow #f}
\]
\[
\text{(vector? '(a b c)) \Rightarrow #f}
\]
\[
\text{(vector? #(a b c)) \Rightarrow #t}
\]

The function \((\text{vector v1 v2 \ldots})\) evaluates \(v1, v2, \ldots\) and puts them into a vector.

\[
\text{(vector 1 2 3) \Rightarrow #(1 2 3)}
\]
The function \((\text{make-vector } k \text{ val})\) creates a vector composed of \(k\) copies of \(\text{val}\). Thus
\[
(\text{make-vector} \ 4 \ (/ \ 1 \ 2)) \Rightarrow \hspace{1cm} \begin{array}{ccc}
\text{(1/2)} & \text{(1/2)} & \text{(1/2)} & \text{(1/2)}
\end{array}
\]
The function \((\text{vector-ref } \text{vect} \ k)\) returns the \(k\)-th element of \(\text{vect}\), starting at position 0. It is essentially the same as \(\text{vect}[k]\) in C or Java. For example,
\[
(\text{vector-ref} \ \#(2 \ 4 \ 6 \ 8 \ 10) \ 3) \Rightarrow 8
\]
The function \((\text{vector-set! } \text{vect} \ k \ \text{val})\) sets the \(k\)-th element of \(\text{vect}\), starting at position 0, to be \(\text{val}\). It is essentially the same as \(\text{vect}[k]=\text{val}\) in C or Java. The value returned by the function is unspecified. The suffix “!” in \text{set!} indicates that the function has a side-effect. For example,
\[
\begin{align*}
\text{(define v #((1 2 3 4 5))}
\text{(vector-set! v 2 0)}
\text{v} & \Rightarrow #((1 2 0 4 5))
\end{align*}
\]
Vectors aren’t lists (and lists aren’t vectors). Thus \((\text{car} \ #((1 2 3))\) doesn’t work.
There are conversion routines:
\[\begin{align*}
\text{• (vector->list } V)\ &\text{ converts vector } \text{vect} \text{ to a list containing the same values as } \text{v}. \text{ For example,} \\
(\text{vector->list} \ \#((1 2 3))) & \Rightarrow \ (1 \ 2 \ 3)
\end{align*}\]
\[\begin{align*}
\text{• (list->vector L) \ &\text{ converts list } L \text{ to a vector containing the same values as } L. \text{ For example,} \\
(\text{list->vector} \ '((1 2 3))) & \Rightarrow \ #((1 2 3))
\end{align*}\]

- In general Scheme names a conversion function from type \(T\) to type \(Q\) as \(T\rightarrow Q\). For example, \(\text{string->list}\) converts a string into a list containing the characters in the string.

**Records and Structs**

In Scheme we can represent a record, struct, or class object as an association list of the form
\[
\begin{align*}
\text{(obj1 val1) (obj2 val2) ...}
\end{align*}
\]
In the association list, which is a list of \((\text{object value})\) sublists, object serves as a “key” to locate the desired sublist.
For example, the association list
\[
\begin{align*}
\text{(A 10) (B 20) (C 30)}
\end{align*}
\]
serves the same role as
\[
\text{struct}\\
\begin{align*}
\text{(int a = 10;}
\text{int b = 20;}
\text{int c = 30;)}
\end{align*}
\]
The predefined Scheme function
(assoc obj alist)
checks alist (an association list) to see if it contains a sublist with obj as its head. If it does, the list starting with obj is returned; otherwise #f (indicating failure) is returned.
For example,
(define L
  '( (a 10) (b 20) (c 30) )) (assoc 'a L) ⇒ (a 10) (assoc 'b L) ⇒ (b 20) (assoc 'x L) ⇒ #f

We can use non-atomic objects as keys too!
(define price-list
  ' ( ((bmw m5)     71095) ((bmw z4)     40495) ((jag xj8)   56975) ((mb sl500)   86655) )
) (assoc '(bmw z4) price-list) ⇒ ((bmw z4) 40495)

Using assoc, we can easily define a structure function:
(structure key alist) will return the value associated with key in alist; in C or Java notation, it returns alist.key.
(define
  (structure key alist)
  (if (assoc key alist)
      (cadr (assoc key alist))
      #f
  ))
We can improve this function in two ways:
• The same call to assoc is made twice; we can save the value computed by using a let expression.
• Often combinations of car and cdr are needed to extract a value. Scheme has a number of predefined functions that combine several calls to car and cdr into one function. For example,
  (caar x) ≡ (car (car x))
  (cadr x) ≡ (car (cdr x))
  (cdar x) ≡ (cdr (car x))
  (cddr x) ≡ (cdr (cdr x))

Using these two insights we can now define a better version of structure
(define
  (structure key alist)
  (let ((p (assoc key alist)))
    (if p
        (cadr p)
        #f
    ))
)
What does \texttt{assoc} do if more than one sublist with the same key exists? It returns the first sublist with a matching key. In fact, this property can be used to make a simple and fast function that updates association lists:

\begin{verbatim}
(define (set-structure key alist val)
  (set-structure! key alist val)
)
\end{verbatim}

If we want to be more space-efficient, we can create a version that updates the internal structure of an association list, using \texttt{set-cdr!} which changes the \texttt{cdr} value of a list:

\begin{verbatim}
(define (set-structure! key alist val)
  (let ((p (assoc key alist)))
    (if p
      (begin
        (set-cdr! p (list val))
        alist
      )
      (cons (list key val) alist)
    )
  )
)
\end{verbatim}

\textbf{Functions are First-class Objects}

Functions may be passed as parameters, returned as the value of a function call, stored in data objects, etc.

This is a consequence of the fact that \texttt{(lambda (args) (body))} evaluates to a function just as \texttt{(+ 1 1)} evaluates to an integer.

\textbf{Scoping}

In Scheme scoping is static (lexical). This means that non-local identifiers are bound to containing lambda parameters, or let values, or globally defined values. For example,

\begin{verbatim}
(define (f x)
  (lambda (y) (+ x y))
)
\end{verbatim}

Function \texttt{f} takes one parameter, \texttt{x}. It returns a function of \texttt{y}, with \texttt{x} in the returned function bound to the value of \texttt{x} used when \texttt{f} was called.

Thus

\begin{verbatim}
(f 10) \equiv (lambda (y) (+ 10 y))
((f 10) 12) \Rightarrow 22
\end{verbatim}
Unbound symbols are assumed to be globals; there is a run-time error if an unbound global is referenced. For example,

(define (p y) (+ x y))
(p 20) ; error -- x is unbound
(define x 10)
(p 20) → 30

We can use let bindings to create private local variables for functions:

(define F
  (let ((X 1))
    (lambda () X))
)

F is a function (of no arguments).
(F) calls F.
(define X 22)
(F) → 1; X used in F is private

We can encapsulate internal state with a function by using private, let-bound variables:

(define cnt
  (let ((I 0))
    (lambda ()
      (set! I (+ I 1)) I))
)

Now,
(cnt) → 1
(cnt) → 2
(cnt) → 3
etc.

Let Bindings can be Subtle

You must check to see if the let-bound value is created when the function is created or when it is called.

Compare

(define cnt
  (let ((I 0))
    (lambda ()
      (set! I (+ I 1)) I))
)

VS.

(define reset
  (lambda ()
    (let ((I 0))
      (set! I (+ I 1)) I))
)
(reset) → 1, (reset) → 1, etc.

Simulating Class Objects

Using association lists and private bound values, we can encapsulate data and functions. This gives us the effect of class objects.

(define (point x y)
  (list
    (list 'rect (lambda () (list x y)))
    (list 'polar (lambda ()
      (list
        (sqrt (+ (* x x) (* y y)))
        (atan (/ x y))))
    )))

A call (point 1 1) creates an association list of the form
  ( (rect funct) (polar funct) )
We can use `structure` to access components:

```
(define p (point 1 1))
( (structure 'rect p) ) ⇒ (1 1)
( (structure 'polar p) ) ⇒
(\sqrt{2} \frac{\pi}{4})
```

We can add new functionality by just adding new `(id function)` pairs to the association list.

```
(define (point x y)
  (list
    (list 'rect
      (lambda () (list x y)))
    (list 'polar
      (lambda ()
        (list
          (sqrt (+ (* x x) (* y y)))
          (atan (/ x y))))
    )))
```

Now we have

```
(define p (point 1 1))
( (structure 'rect p) ) ⇒ (1 1)
( (structure 'polar p) ) ⇒
(\sqrt{2} \frac{\pi}{4})
```

```
((structure 'set-polar! p) 1 \pi/4)
⇒ (1 \pi/4)
( (structure 'rect p) ) ⇒
\frac{1}{\sqrt{2}} \frac{1}{\sqrt{2}}
```

Limiting Access to Internal Structure

We can improve upon our association list approach by returning a single function (similar to a C++ or Java object) rather than an explicit list of `(id function)` pairs.

The function will take the name of the desired operation as one of its arguments.
First, let's differentiate between
(define def1
  (let ( (I 0) )
    (lambda () (set! I (+ I 1)) I))
)
and
(define def2
  (let ( (I 0) )
    (lambda () (set! I (+ I 1)) I))
)
def1 is a function of zero arguments that increments a local variable and returns its updated value.
def2 is a function (of zero arguments) that generates a function of zero arguments (that increments a local variable and returns its updated value). Each call to def2 creates a different function.

Stack Implemented as a Function

(define (stack )
  (let ((s () ))
    (lambda (op . args) ; var # args
      (cond
        ((equal? op 'push!)
          (set! s (cons (car args) s))
          (car s))
        ((equal? op 'pop!)
          (if (null? s)
            #f
            (let ((top (car s)))
              (set! s (cdr s))
              top )))
        ((equal? op 'empty?)
          (null? s))
        (else #f)
      ))
    ))
)

(define stk (stack));new empty stack
(stk 'push! 1) ⇒ 1 ;s = (1)
(stk 'push! 3) ⇒ 3 ;s = (3 1)
(stk 'push! 'x) ⇒ x ;s = (x 3 1)
(stk 'pop!) ⇒ x ;s = (3 1)
(stk 'empty?) ⇒ #f ;s = (3 1)
(stk 'dump) ⇒ #f ;s = (3 1)

Higher-Order Functions

A higher-order function is a function that takes a function as a parameter or one that returns a function as its result.

A very important (and useful) higher-order function is map, which applies a function to a list of values and produces a list or results:
(define (map f L)
  (if (null? L)
      ()
      (cons (f (car L))
            (map f (cdr L)))))
)

Note: In Scheme's built-in implementation of map, the order of function application is unspecified.
(map sqrt '(1 2 3 4 5)) ⇒
(1 1.414 1.732 2 2.236)
(map (lambda(x) (* x x)) '(1 2 3 4 5)) ⇒
(1 4 9 16 25)

Map may also be used with multiple argument functions by supplying more than one list of arguments:

(map + '(1 2 3) '(4 5 6)) ⇒
(5 7 9)

The Reduce Function

Another useful higher-order function is reduce, which reduces a list of values to a single value by repeatedly applying a binary function to the list values.

This function takes a binary function, a list of data values, and an identity value for the binary function:

(define (reduce f L id)
  (if (null? L)
      id
      (f (car L)
         (reduce f (cdr L) id))))

(reduce + '(1 2 3 4 5) 0) ⇒ 15
(reduce * '(1 2 4 6 8 10) 1) ⇒ 3840

(reduce append '((1 2 3) (4 5 6) (7 8)) ()) ⇒ (1 2 3 4 5 6 7 8)
(reduce expt '(2 2 2 2) 1) ⇒ \[2^{2^2} = 65536\]

(reduce expt '(2 2 2 2) 1) ⇒ \[2^{\text{65536}}\]

(string-length (number->string (reduce expt '(2 2 2 2) 1))) ⇒ 19729 ; digits in \[2^{65536}\]

Sharing vs. Copying

In languages without side-effects an object can be copied by copying a pointer (reference) to the object; a complete new copy of the object isn’t needed. Hence in Scheme (define A B) normally means

In Scheme (define A B) normally means

![A B](image)

But, if side-effects are possible we may need to force a physical copy of an object or structure:
(define (copy obj)
  (if (pair? obj)
      (cons (copy (car obj))
            (copy (cdr obj)))
      obj))

For example,
(define A '(1 2))
(define B (cons A A))
B = ( (1 2) 1 2)

(set-car! (car B) 10)
B = ( (10 2) 10 2)

(define C (cons (copy A) (copy A)))

(similar concerns apply to strings and vectors, because their internal structure can be changed.)

(set-car! (car C) 20)
C = ((20 2) 10 2)

Shallow & Deep Copying

A copy operation that copies a pointer (or reference) rather than the object itself is a shallow copy.
For example, In Java,
Object O1 = new Object();
Object O2 = new Object();
O1 = O2; // shallow copy

If the structure within an object is physically copied, the operation is a deep copy.
In Java, for objects that support the clone operation,
O1 = O2.clone(); // deep copy

Even in Java’s deep copy (via the clone() operation), objects referenced from within an object are shallow copied. Thus given
class List {
    int value;
    List next;
}
List L, M;
M = L.clone();
L.value and M.value are independent, but L.next and M.next refer to the same List object.
A complete deep copy, that copies all objects linked directly or indirectly, is expensive and tricky to implement.
(Consider a complete copy of a circular linked list).

Equality Checking in Scheme
In Scheme = is used to test for numeric equality (including comparison of different numeric types). Non-numeric arguments cause a run-time error. Thus

\[ (= \ 1 \ 1) \Rightarrow \#t \]
\[ (= \ 1 \ 1.0) \Rightarrow \#t \]
\[ (= \ 1 \ 2/2) \Rightarrow \#t \]
\[ (= \ 1 \ 1+0.0i) \Rightarrow \#t \]

To compare non-numeric values, we can use either:
pointer equivalence (do the two operands point to the same address in memory)
structural equivalence (do the two operands point to structures with the same size, shape and components, even if they are in different memory locations)
In general pointer equivalence is faster but less accurate.

Scheme implements both kinds of equivalence tests.
(eqv? obj1 obj2)
This tests if obj1 and obj2 are the exact same object. This works for atoms and pointers to the same structure.
(equal? obj1 obj2)
This tests if obj1 and obj2 are the same, component by component. This works for atoms, lists, vectors and strings.

(eqv? 1 1) \Rightarrow \#t
(eqv? 1 (+ 0 1)) \Rightarrow \#t
(eqv? 1/2 (- 1 1/2)) \Rightarrow \#t
(eqv? (cons 1 2) (cons 1 2)) \Rightarrow \#f
(eqv? "abc" "abc") \Rightarrow \#f
In general it is wise to use `equal?` unless speed is a critical factor.

I/O in Scheme

Scheme provides simple read and write functions, directed to the "standard in" and "standard out" files.

(read)
Reads a single Scheme object (an atom, string, vector or list) from the standard in file. No quoting is needed.

(write obj)
Writes a single object, obj, to the standard out file.

(display obj)
Writes obj to the standard out file in a more readable format. (Strings aren't quoted, and characters aren't escaped.)

(newline)
Forces a new line on standard out file.

Ports

Ports are Scheme objects that interface with systems files. I/O to files is normally done through a port object bound to a system file.

(open-input-file "path to file")
This returns an input port associated with the "path to file" string (which is system dependent). A runtime error is signalled if "path to file" specifies an illegal or inaccessible file.

(read port)
Reads a single Scheme object (an atom, string, vector or list) from port, which must be an input port object.

(eof-object? obj)
When the end of an input file is reached, a special eof-object is returned. eof-object? tests whether an object is this special end-of-file marker.

(open-output-file "path to file")
This returns an output port associated with the "path to file" string (which is system dependent). A runtime error is signalled if "path to file" specifies an illegal or inaccessible file.

(write obj port)
Writes a single object, obj, to the output port specified by port.
(display obj port)
Writes obj to the output port specified by port. display uses a more readable format than write does. (Strings aren’t quoted, and characters aren’t escaped.)

(close-input-port port)
This closes the input port specified by port.

(close-output-port port)
This closes the output port specified by port.

Example—Reading & Echoing a File

We will iterate through a file, reading and echoing its contents. We need a good way to do iteration; recursion is neither natural nor efficient here.

Scheme provides a nice generalization of the let expression that is similar to C’s for loop.

(let X ( (id1 val1) (id2 val2) ... )
...         (X v1 v2 ...)
)
A name for the let (x in the example) is provided. As usual, val1 is evaluated and bound to id1, val2 is evaluated and bound to id2, etc. In the body of the let, the let may be “called” (using its name) with a fresh set of values for the let variables. Thus (X v1 v2 ...) starts the next iteration of the let with id1 bound to v1, id2, bound to v2, etc.

The calls look like recursion, but they are implemented as loop iterations.

For example, in

(let loop ( (x 1) (sum 0) )
  (if (<= x 10)
    (loop (+ x 1) (+ sum x))
    sum
  )
)
we sum the values of x from 1 to 10. Compare it to

for (x=1,sum=0; x <= 10;
    sum+=x,x+=1)
{}
Control Flow in Scheme

Normally, Scheme's control flow is simple and recursive:

- The first argument is evaluated to get a function.
- Remaining arguments are evaluated to get actual parameters.
- Actual parameters are bound to the function's formal parameters.
- The functions' body is evaluated to obtain the value of the function call.

This approach routinely leads to deeply nested expression evaluation.

Let's try

(define (*list0 L)
  (cond
   (((null? L) 1)
    ((= 0 (car L)) 0)
    (else (* (car L) (*list0 (cdr L)))))
  )
)

This helps a bit—we never go past a zero in L, but we still unnecessarily do a sequence of pending multiplies, all of which must yield zero!

Can we escape from a sequence of nested calls once we know they're unnecessary?

As an example, consider a simple function that multiplies a list of integers:

(define (*list L)
  (if (null? L)
      1
      (* (car L)(*list (cdr L)))
  )
)

The call (*list '(1 2 3 4 5)) expands to
(* 1 (* 2 (* 3 (* 4 (* 5 1)))))

But,
what if we get clever and decide to improve this function by noting that if 0 appears anywhere in list L, the product must be 0?

We can create an alternative to echo that uses

(call-with-input-file filename function)

This function opens filename, creates an input port from it, and then calls function with that port as an argument:

(define (echo2 filename)
  (call-with-input-file filename
    (lambda(port)
        (let loop ( (obj (read port)))
          (if (eof-object? obj) #t
              (begin
                (write obj)
                (newline)
                (loop (read port)))
          )
        )
    )
  )
)
Exceptions

In languages like Java, a statement may throw an exception that's caught by an enclosing exception handler. Code between the statement that throws the exception and the handler that catches it is abandoned.

Let's solve the problem of avoiding multiplication of zero in Java, using its exception mechanism:

```java
class Node {
    int val;
    Node next;
}
class Zero extends Throwable {
}
int mult (Node L) {
    try {
        return multNode(L);
    } catch (Zero z) {
        return 0;
    }
}
int multNode(Node L) throws Zero {
    if (L == null)
        return 1;
    else if (L.val == 0)
        throw new Zero();
    else return L.val * multNode(L.next);
}
```

In this implementation, no multiplies by zero are ever done.

Continuations

In our Scheme implementation of *list, we'd like a way to delay doing any multiplies until we know no zeros appear in the list. One approach is to build a continuation—a function that represents the context in which a function's return value will be used:

```scheme
(define (*listC L con)
  (cond
   ((null? L) (con 1))
   ((= 0 (car L)) 0)
   (else
    (*listC (cdr L)
     (lambda (n)
      (* n (con (car L)))))))
)
```

The top-level call is

```scheme
(*listC L (lambda (x) x))
```

For ordinary lists *listC expands to a series of multiplies, just like *list did.

```scheme
(define (id x) x)
(*listC '(1 2 3) id) ⇒
(*listC '(2 3)
     (lambda (n) (* n (id 1)))) ≡
(*listC '(2 3)
     (lambda (n) (* n 1))) ⇒
(*listC '(3)
     (lambda (n) (* n (* 2 1)))) ≡
(*listC '(3)
     (lambda (n) (* n 2))) ⇒
(*listC ()
     (lambda (n) (* n (* 3 2)))) ≡
(*listC () (lambda (n) (* n 6))) ⇒ (* 1 6) ⇒ 6
```
But for a list with a zero in it, we get a different execution path:

\[(\ast\text{listC} \ ('1\ 0\ 3)\ \text{id}) \Rightarrow\]
\[(\ast\text{listC} \ ('0\ 3)\]
\[
\quad\text{(lambda (n) (* n (\text{id} 1)))) \Rightarrow 0\]

No multiplies are done!

**Another Example of Continuations**

Let’s redo our list multiply example so that if a zero is seen in the list we return a function that computes the product of all the non-zero values and a parameter that is the “replacement value” for the unwanted zero value. The function gives the caller a chance to correct a probable error in the input data.

We create

\[(\ast\text{list2} \ L) \equiv\]

Product of all integers in \(L\) if no zero appears
else
\[
\text{(lambda (n) (* n product-of-all-nonzeros-in-L))}\]

\[
\text{(define \((\ast\text{list2} \ L) \equiv \text{(*listE L id)})\)}
\]

\[
\text{(define \((\ast\text{listE L con}) \equiv)}
\]

\[
\text{cond}
\]
\[
\text{((null? L) \ (con 1))}
\]
\[
\text{((= 0 (car L))}
\]
\[
\quad\text{(lambda (n)}
\]
\[
\quad\text{(* \ (con n)}
\]
\[
\quad\text{(*listE (cdr L) id)))}
\]
\[
\text{else}
\]
\[
\text{(*listE (cdr L))}
\]
\[
\quad\text{(lambda (m)}
\]
\[
\quad\text{(* m (con (car L)))))}
\]
\[
\quad\}
\]
\[
\}
\]

In the following, we check to see if \((\ast\text{list2} \ L)\) returns a number or a function. If a function is returned, we call it with 1, effectively removing 0 from the list

\[
\text{(let \ ((V (*\text{list2} \ L)) \)}
\]
\[
\text{(if \ (number? \ V) \}
\]
\[
\quad \ V \}
\]
\[
\quad (V 1) \}
\]
\[
\}
\]
For ordinary lists \(*\text{list2}\) expands to a series of multiplies, just like \(*\text{list}\) did.

\(*\text{listE \'(1 2 3) id} \Rightarrow \)
\(*\text{listE \'(2 3)} \)
(\lambda (m) (* m (id 1)))) \equiv
\(*\text{listE \'(2 3)} \)
(\lambda (m) (* m 1))) \Rightarrow
\(*\text{listE \'(3)} \)
(\lambda (m) (* m (* 2 1)))) \equiv
\(*\text{listE \'(3)} \)
(\lambda (m) (* m 2))) \Rightarrow
\(*\text{listE ()} \)
(\lambda (m) (* m (* 3 2)))) \equiv
\(*\text{listE ()} (\lambda (n) (* n 6))) \Rightarrow (* 1 6) \Rightarrow 6

But for a list with a zero in it, we get a different execution path:

\(*\text{listE \'(1 0 3) id} \Rightarrow \)
\(*\text{listE \'(0 3)} \)
(\lambda (m) (* m (id 1)))) \Rightarrow
\(*\text{listE \'(0 3)} \)
(\lambda (n) (* (con n) \(*\text{listE \'(3) id}\)) \equiv
\(*\text{listE \'(3)} \)
(\lambda (n) (* (* n 1) \(*\text{listE \'(3) id}\)) \equiv
\(*\text{listE \'(3)} \)
(\lambda (n) (* (* n 1) 3)) \Rightarrow
\(*\text{listE ()} (\lambda (n) (* n 6))) \Rightarrow (* 1 6) \Rightarrow 6

But note that only one zero value in the list is handled correctly!
Why?

\(\text{(define \(*\text{listE L con})\)}\)
\(\text{(cond}\)
\((= 0 (\text{car L}))\)
\(\text{(lambda (n)}\)
\((* (\text{con n}}\)
\(\text{(*listE (cdr L) id)})\))
\(\text{else}\)
\(\text{(*listE (cdr L}}\)
\(\text{(lambda (m)}\)
\(\text{(* m (con (car L))))}\))
\(\text{)}\)

Continuations in Scheme

Scheme provides a built-in mechanism for creating continuations. It has a long name: \text{call-with-current-continuation}

This name is usually abbreviated as \text{call/cc}

(\text{perhaps using define}).

\text{call/cc} takes a single function as its argument. That function also takes a single argument. That is, we use \text{call/cc} as

\(\text{(call/cc funct) where}\)
funct \equiv (\text{lambda (con) (body)})

\text{call/cc} calls the function that it is given with the “current continuation” as the function’s argument.
Current Continuations

What is the current continuation?
It is itself a function of one argument. The current continuation function represents the execution context within which the call/cc appears. The argument to the continuation is a value to be substituted as the return value of call/cc in that execution context.

For example, given

\( (+ \text{(fct n)} 3) \)

the current continuation for \((\text{fct n})\)
\(\text{is (lambda (x) (+ x 3))}\)

Given \((* 2 (+ \text{(fct z)} 10))\)
the current continuation for \((\text{fct z})\)
\(\text{is (lambda (m) (* 2 (+ m 10))})\)

To use call/cc to grab a continuation in (say) \((+ \text{(fct n)} 3)\)
we make \((\text{fct n})\) the body of a function of one argument. Let’s call that argument return. We therefore create

\((\lambda \text{return} \text{(fct n)})\)

Then

\((\text{call/cc})\)

\((\lambda \text{return} \text{(fct n)})\)

binds the current continuation to return and executes \((\text{fct n})\).

We can ignore the current continuation bound to return and do a normal return

Or

we can use return to force a return to the calling context of the call/cc.

The call \((\text{return value})\) forces value to be returned as the value of call/cc in its context of call.

Example:

\( (* \text{(call/cc (lambda(return))}) \) \)

\( (/ \text{(g return) 0)}) \)

\(10\)

\(\text{return}\)

\(\text{(define (g con) (con 5))}\)

Now during evaluation no divide by zero error occurs. Rather, when \((\text{g return})\) is called, 5 is passed to con, which is bound to return. Therefore 5 is used as the value of the call to call/cc, and 50 is computed.

Continuations are Just Functions

Continuations may be saved in variables or data structures and called in the future to “reactive” a completed or suspended computation.

\(\text{(define CC ()})\)

\(\text{(define (F)}\)

\(\text{let (}\)

\(\text{(v (call/cc}\)

\(\text{(lambda (here)}\)

\(\text{(set! CC here)\}

\(1}))\))\)

\(\text{(display "The ans is: ")}\)

\(\text{(display v)\}

\(\text{(newline)\})\)

\(\text{)}\)

This displays The ans is: 1
At any time in the future, \((\text{CC 10})\) will display The ans is: 10
List Multiplication Revisited

We can use call/cc to reimplement the original *list to force an immediate return of 0 (much like a throw in Java):

\[
\text{(define (*listc L return)}
\text{(cond}
\text{((null? L) 1)}
\text{((= 0 (car L)) (return 0))}
\text{(else (* (car L)}
\text{(*listc (cdr L) return))))}
\text{))}
\]

A 0 in L forces a call of (return 0) which makes 0 the value of call/cc.

Interactive Replacement of Error Values

Using continuations, we can also redo *listE so that zeroes can be replaced interactively! Multiple zeroes (in both original and replacement values) are correctly handled.

\[
\text{(define (*list L)}
\text{(let (}
\text{V (call/cc}
\text{(lambda (here)}
\text{(*liste L here))))}
\text{)}
\text{(if (number? V)}
\text{V}
\text{(begin}
\text{(display}
\text{"Enter new value for 0")}
\text{newline) (newline)}
\text{(V (read))}
\text{)}
\text{)}
\text{))}
\]

Implementing Coroutines with call/cc

Coroutines are a very handy generalization of subroutines. A coroutine may suspend its execution and later resume from the point of suspension. Unlike subroutines, coroutines do not have to complete their execution before they return.

Coroutines are useful for computation of long or infinite streams of data, where we wish to compute some data, use it, compute additional data, use it, etc.

Subroutines aren't always able to handle this, as we may need to save a lot of internal state to resume with the correct next value.
Producer/Consumer using Coroutines

The example we will use is one of a consumer of a potentially infinite stream of data. The next integer in the stream (represented as an unbounded list) is read. Call this value \( n \). Then the next \( n \) integers are read and summed together. The answer is printed, and the user is asked whether another sum is required. Since we don’t know in advance how many integers will be needed, we’ll use a coroutine to produce the data list in segments, requesting another segment as necessary.

Next, we’ll consider \( \text{sum-n-elems} \), which adds the first element of list (a running sum) to the next \( n \) elements on the list. We’ll use \( \text{moredata} \) to extend the data list as needed.

```scheme
(define (sum-n-elems n list)
  (cond
   ((= 0 n)  list)
   ((null? (cdr list))
    (sum-n-elems n
     (cons (car list) (moredata))))
   (else
    (sum-n-elems (- n 1)
     (cons (+ (car list)
              (cadr list))
          (cddr list))))
   )
)
```

The function \( \text{moredata} \) is called whenever we need more data. Initially a producer function is called to get the initial segment of data. \( \text{producer} \) actually returns the next data segment plus a continuation (stored in \( \text{producer-cc} \)) used to resume execution of \( \text{producer} \) when the next data segment is required.

```scheme
(define (consumer)
  (next 0); reset next function
  (let loop ((data (moredata)))
    (let ((
          (sum+restoflist
           (sum-n-elems (car data)
            (cons 0 (cddr data))))))
      (display (car sum+restoflist))
      (newline)
      (display "more? ")
      (if (equal? (read) 'y)
       (if (= 1
            (length sum+restoflist))
        (loop (moredata))
        (loop (cdr sum+restoflist))
       )
       #t ; Normal completion
     )
   )
)
```
(define moredata
  (let ( (producer-cc () ) )
    (lambda ()
      (let ( (data+cont)
            (if (null? producer-cc)
                (call/cc (lambda (here)
                           (producer here)))
                (call/cc (lambda (here)
                           (producer-cc here))))
                      ))
      (set! producer-cc
            (cdr data+cont))
      (car data+cont)
    ))
)

Function (next z) returns the next z integers in an infinite sequence that starts at 1. A value z=0 is a special flag indicating that the sequence should be reset to start at 1.

(define next
  (let ( (i 1))
    (lambda (z)
      (if (= 0 z)
          (set! i 1)
          (let loop
            ((cnt z) (val i) (ints ()))
              (if (> cnt 0)
                  (loop (- cnt 1)
                        (+ val 1)
                        (append ints
                                     (list val)))
                  (begin
                    (set! i val)
                    ints
                  ))
          )
          ))
))

The function producer generates an infinite sequence of integers (1,2,3,...). It suspends every 5/10/15/25 elements and returns control to moredata.

(define (producer initial-return)
  (let loop
    ((return initial-return) )
    (set! return
          (call/cc (lambda (here)
                     (return (cons (next 5)
                                   here))))))
  (set! return
        (call/cc (lambda (here)
                   (return (cons (next 10)
                                 here))))))
  (set! return
        (call/cc (lambda (here)
                   (return (cons (next 15)
                                 here))))))
  (loop
   (call/cc (lambda (here)
             (return (cons (next 25)
                           here))))))
)

Reading Assignment

- MULTILISP: a language for concurrent symbolic computation, by Robert H. Halstead (linked from class web page)
Lazy Evaluation

Lazy evaluation is sometimes called “call by need.” We do an evaluation when a value is used; not when it is defined.

Scheme provides for lazy evaluation:

(delay expression)

Evaluation of expression is delayed. The call returns a “promise” that is essentially a lambda expression.

(force promise)

A promise, created by a call to delay, is evaluated. If the promise has already been evaluated, the value computed by the first call to force is reused.

Example:

Though and is predefined, writing a correct implementation for it is a bit tricky.

The obvious program

(define (and A B)
  (if A
    B
    #f)
)

is incorrect since B is always evaluated whether it is needed or not. In a call like

(and (not (= i 0)) (> (/ j i) 10))

unnecessary evaluation might be fatal.

An argument to a function is strict if it is always used. Non-strict arguments may cause failure if evaluated unnecessarily.

With lazy evaluation, we can define a more robust and function:

(define (and A B)
  (if A
    (force B)
    #f)
)

This is called as:

(and (not (= i 0))
  (delay (> (/ j i) 10)))

Note that making the programmer remember to add a call to delay is unappealing.

Delayed evaluation also allows us a neat implementation of suspensions.

The following definition of an infinite list of integers clearly fails:

(define (inflist i)
  (cons i (inflist (+ i 1))))

But with use of delays we get the desired effect in finite time:

(define (inflist i)
  (cons i
    (delay (inflist (+ i 1)))))

Now a call like (inflist 1) creates

1 promise for (inflist 2)
We need to slightly modify how we explore suspended infinite lists. We can't redefine \texttt{car} and \texttt{cdr} as these are far too fundamental to tamper with.

Instead we'll define \texttt{head} and \texttt{tail} to do much the same job:

\begin{verbatim}
(define head car)
(define (tail L)
  (force (cdr L)))
\end{verbatim}

\texttt{head} looks at \texttt{car} values which are fully evaluated.

\texttt{tail} forces one level of evaluation of a delayed \texttt{cdr} and saves the evaluated value in place of the suspension (promise).

---

**Exploiting Parallelism**

Conventional procedural programming languages are difficult to compile for multiprocessors. Frequent assignments make it difficult to find independent computations.

Consider (in Fortran):

\begin{verbatim}
  do 10  I = 1,1000
  X(I) = 0
  A(I) = A(I+1)+1
  B(I) = B(I-1) - 1
  C(I) = (C(I-2) + C(I+2))/2
10  continue
\end{verbatim}

This loop defines 1000 values for arrays \texttt{X}, \texttt{A}, \texttt{B} and \texttt{C}.

---

Which computations can be done in parallel, partitioning parts of an array to several processors, each operating independently?

- \texttt{X(I) = 0}
  Assignments to \texttt{x} can be readily parallelized.

- \texttt{A(I) = A(I+1)+1}
  Note that each computation of \texttt{A(I)} uses an \texttt{A(I+1)} value that is yet to be changed. Thus a whole array of new \texttt{A} values can be computed from an array of "old" \texttt{A} values in parallel.

- \texttt{B(I) = B(I-1)-1}
  This is less obvious. Each \texttt{B(I)} uses \texttt{B(I-1)} which is defined in terms of \texttt{B(I-2)}, etc. Ultimately all new \texttt{B} values depend only on \texttt{B(0)} and \texttt{I}. That is, \texttt{B(I) = B(0) - I}. So this...
computation can be parallelized, but it takes a fair amount of insight to realize it.

\[ C(I) = \frac{C(I-2) + C(I+2)}{2} \]

It is clear that even and odd elements of \( C \) don't interact. Hence two processors could compute even and odd elements of \( C \) in parallel. Beyond this, since both earlier and later \( C \) values are used in each computation of an element, no further means of parallel evaluation is evident. Serial evaluation will probably be needed for even or odd values.

### How is Parallelism Found?

There are two approaches:

- We can use a “smart” compiler that is able to find parallelism in existing programs written in standard serial programming languages.
- We can add features to an existing programming language that allows a programmer to show where parallel evaluation is desired.

### Exploiting Parallelism in Scheme

Assume we have a shared-memory multiprocessor. We might be able to assign different processors to evaluate various independent subexpressions.

For example, consider

\[
\text{map (lambda (x) (* 2 x)) '(1 2 3 4 5))
\]

We might assign a processor to each list element and compute the lambda function on each concurrently:

![Diagram of parallel evaluation]

### Concurrentization

Concurrentization (often called parallelization) is process of automatically finding potential concurrent execution in a serial program.

Automatically finding current execution is complicated by a number of factors:

- **Data Dependence**
  
  Not all expressions are independent. We may need to delay evaluation of an operator or subprogram until its operands are available.
  
  Thus in
  
  \[
  (+ (* x y) (* y z))
  \]
  
  we can't start the addition until both multiplications are done.
• Control Dependence
   Not all expressions need be (or should be) evaluated.
   In
   \[
   (\text{if } (= a 0) \\
   \quad 0 \\
   \quad (/ b a))
   \]
   we don’t want to do the division until we know \(a \neq 0\).

• Side Effects
   If one expression can write a value that another expression might read, we probably will need to serialize their execution.
   Consider
   \[
   (\text{define rand!} \\
   \quad (\text{let } ((\text{seed } 99)) \\
   \quad \quad (\text{lambda } () \\
   \quad \quad \quad (\text{set! } \text{seed} \quad \text{(mod } (* \text{seed } 1001) \text{ } 101101)) \\
   \quad \quad \quad \text{seed} \\
   \quad \quad )))
   \]

Now in
\[
(+ (f \text{ (rand!)) } (g \text{ (rand!))})
\]
we can’t evaluate \(f \text{ (rand!)) and } (g \text{ (rand!)) in parallel, because of the side effect of } \text{set! in rand!}. In fact if we did, \(f \) and \(g \) might see exactly the same “random” number! (Why?)

• Granularity
   Evaluating an expression concurrently has an overhead (to setup a concurrent computation). Evaluating very simple expressions (like \((\text{car } x)\) or \((+ x 1)\)) in parallel isn’t worth the overhead cost.
   Estimating where the “break even” threshold is may be tricky.

Utility of Concurrentization
Concurrentization has been most successful in engineering and scientific programs that are very regular in structure, evaluating large multidimensional arrays in simple nested loops. Many very complex simulations (weather, fluid dynamics, astrophysics) are run on multiprocessors after extensive concurrentization.
Concurrentization has been far less successful on non-scientific programs that don’t use large arrays manipulated in nested for loops. A compiler, for example, is difficult to run (in parallel) on a multiprocessor.

Concurrentization within Processors
Concurrentization is used extensively within many modern uniprocessors. Pentium and PowerPC processors routinely execute several instructions in parallel if they are independent (e.g., read and write distinct registers). This are superscalar processors.
These processors also routinely speculate on execution paths, “guessing” that a branch will (or won’t) be taken even before the branch is executed! This allows for more concurrent execution than if strictly “in order” execution is done. These processors are called “out of order” processors.
Adding Parallel Features to Programming Languages.

It is common to take an existing serial programming language and add features that support concurrent or parallel execution. For example versions for Fortran (like HPF—High Performance Fortran) add a parallel do loop that executes individual iterations in parallel. Java supports threads, which may be executed in parallel. Synchronization and mutual exclusion are provided to avoid unintended interactions.

Multilisp

Multilisp is a version of Scheme augmented with three parallel evaluation mechanisms:

- Pcall
  Arguments to a call are evaluated in parallel.
- Future
  Evaluation of an expression starts immediately. Rather than waiting for completion of the computation, a “future” is returned. This future will eventually transform itself into the result value (when the computation completes)
- Delay
  Evaluation is delayed until the result value is really needed.

The Pcall Mechanism

Pcall is an extension to Scheme’s function call mechanism that causes the function and its arguments to be all computed in parallel.

Thus

\[ (pcall F \, X \, Y \, Z) \]

causes \( F, X, Y \) and \( Z \) to all be evaluated in parallel. When all evaluations are done, \( F \) is called with \( X, Y \) and \( Z \) as its parameters (just as in ordinary Scheme).

Compare

\[ (+ \, (* \, X \, Y) \, (* \, Y \, Z)) \]

with

\[ (pcall + \, (* \, X \, Y) \, (* \, Y \, Z)) \]

It may not look like \texttt{pcall} can give you that much parallel execution, but in the context of recursive definitions, the effect can be dramatic.

Consider \texttt{treemap}, a version of map that operates on binary trees (S-expressions).

\[
\begin{align*}
\text{(define (treemap fct tree)} \\
& \text{(if (pair? tree)} \\
& \quad \text{(pcall cons)} \\
& \quad \text{(treemap fct (car tree)))} \\
& \quad \text{(treemap fct (cdr tree)))} \\
& \quad \text{(fct tree)} \\
& \text{)})
\end{align*}
\]

Look at the execution of \texttt{treemap} on the tree

\[
(((1 \, 2) \, . \, (3 \, 4)) \, . \, ((5 \, 6) \, . \, (7 \, 8)))
\]
We start with one call that uses the whole tree. This splits into two parallel calls, one operating on

\[(1 \cdot 2) \cdot (3 \cdot 4)\]

and the other operating on

\[(5 \cdot 6) \cdot (7 \cdot 8)\]

Each of these calls splits into 2 calls, and finally we have 8 independent calls, each operating on the values 1 to 8.

---

**Futures**

Evaluation of an expression as a future is the most interesting feature of Multilisp.

The call

\[(future \ expr)\]

begins the evaluation of \(expr\). But rather than waiting for \(expr\)'s evaluation to complete, the call to \(future\) returns immediately with a new kind of data object—a future. This future is actually an “IOU.” When you try to use the value of the future, the computation of \(expr\) may or may not be completed. If it is, you see the value computed instead of the future—it automatically transforms itself. Thus evaluation of \(expr\) appears instantaneous.

---

If the computation of \(expr\) is not yet completed, you are forced to wait until computation is completed. Then you may use the value and resume execution.

But this is exactly what ordinary evaluation does anyway—you begin evaluation of \(expr\) and wait until evaluation completes and returns a value to you!

---

To see the usefulness of futures, consider the usual definition of Scheme's map function:

\[
\begin{align*}
&\text{(define (map } f \ L) \\nonumber \\
&\quad \text{(if (null? } L) \\nonumber \\
&\quad \quad () \\nonumber \\
&\quad \quad (\text{cons } (f \ (\text{car } L)) \\nonumber \\
&\quad \quad \quad (\text{map } f \ (\text{cdr } L)))\nonumber \\
&\quad ) \nonumber \\
\end{align*}
\]

If we have a call

\[(\text{map } slow-function \ long-list)\]

where \(slow-function\) executes slowly and \(long-list\) is a large data structure, we can expect to wait quite a while for computation of the result list to complete.
Now consider fastmap, a version of map that uses futures:

\[
\text{(define (fastmap } f \text{ } L) \nonumber \\
\text{ (if (null? } L) \nonumber \\
\text{ () } \nonumber \\
\text{ (cons } \nonumber \\
\text{ (future } (f \text{ (car } L)) \text{) } \nonumber \\
\text{ (fastmap } f \text{ (cdr } L) \text{) } \nonumber \\
\text{ ) }) \nonumber \\
\text{) }) \nonumber \\
\]

Now look at the call

\[
\text{(fastmap slow-function long-list) } \nonumber \\
\]

We will exploit a useful aspect of futures—they can be cons'ed together without delay, even if the computation isn't completed yet. Why? Well a cons just stores a pair of pointers, and it really doesn't matter what the pointers reference (a future or an actual result value).

The call to fastmap can actually return before any of the call to slow-function have completed:

Eventually all the futures automatically transform themselves into data values:

Note that pcall can be implemented using futures. That is, instead of

\[
\text{(pcall } F \text{ } X \text{ } Y \text{ } Z) \nonumber \\
\text{we can use} \nonumber \\
\text{((future } F \text{) (future } X \text{) (future } Y \text{) (future } Z)) \nonumber \\
\]

In fact the latter version is actually more parallel—execution of \( F \) can begin even if all the parameters aren't completely evaluated.

Another Example of Futures

The following function, partition, will take a list and a data value (called pivot). partition will partition the list into two sublists:

(a) Those elements \( \leq \) pivot
(b) Those elements \( > \) pivot

\[
\text{(define (partition pivot } L) \nonumber \\
\text{ (if (null? } L) \nonumber \\
\text{ (cons () () ) } \nonumber \\
\text{ (let ((tail-part } \nonumber \\
\text{ (partition pivot (cdr } L))) \nonumber \\
\text{ (if (\leq (car } L \text{) pivot) } \nonumber \\
\text{ (cons } \nonumber \\
\text{ (cons (car } L \text{) (car tail-part)) } \nonumber \\
\text{ (cdr tail-part)) } \nonumber \\
\text{ (cons } \nonumber \\
\text{ (car tail-part)) } \nonumber \\
\text{ (cons (car } L \text{) (cdr tail-part)) } \nonumber \\
\text{ ) ) ) ) } \nonumber \\
\]
We want to add futures to partition, but where? It makes sense to use a future when a computation may be lengthy and we may not need to use the value computed immediately. What computation fits that pattern? The computation of tail-part. We'll mark it in a blue box to show we plan to evaluate it using a future:

```
(define (partition pivot L)
  (if (null? L)
      (cons () () )
      (let ((tail-part
            (partition pivot (cdr L))))
        (if (<= (car L) pivot)
            (cons
             (cons (car L) (car tail-part))
             (cdr tail-part))
            (cons
             (car tail-part))
            (cons (car L) (cdr tail-part))
      ) ) )
)
```

But this one change isn't enough! We soon access the car and cdr of tail-part, which forces us to wait for its computation to complete. To avoid this delay, we can place the four references to car or cdr of tail-part into futures too:

```
(define (partition pivot L)
  (if (null? L)
      (cons () () )
      (let ((tail-part
            (partition pivot (cdr L))))
        (if (<= (car L) pivot)
            (cons
             (cons (car L) (car tail-part))
             (cdr tail-part))
            (cons
             (car tail-part))
            (cons (car L) (cdr tail-part))
      ) ) )
)
```

Now we can build the initial part of the partitioned list (that involving pivot and (car L) independently of the recursive call of partition, which completes the rest of the list. For example,

```
(partition 17 '(5 3 8 ...))
```

creates a future (call it future1) to compute

```
(partition 17 '(3 8 ...))
```

It also creates future2 to compute (car tail-part) and future3 to compute (cdr tail-part). The call builds
Reading Assignment

- Roosta: Section 13.3
- Introduction to Standard ML (linked from class web page)
- Webber: Chapters 5, 7, 9, 11

ML — Meta Language

SML is Standard ML, a popular ML variant.

ML is a functional language that is designed to be efficient and type-safe. It demonstrates that a functional language need not use Scheme’s odd syntax and need not bear the overhead of dynamic typing.

SML’s features and innovations include:

1. Strong, compile-time typing.
2. Automatic type inference rather than user-supplied type declarations.
3. Polymorphism, including “type variables.”

4. Pattern-directed Programming

   fun len([]) = 0
   | len(a::b) = 1+len(b);

5. Exceptions

6. First-class functions

7. Abstract Data Types

   coin of int |
   bill of int |
   check of string*real;
   val dime = coin(10);

A good ML reference is
“Elements of ML Programming,”
by Jeffrey Ullman
(Prentice Hall, 1998)

SML is Interactive

You enter a definition or expression, and SML returns a result with an inferred type.

The command
use "file name";
loads a set of ML definitions from a file.

For example (SML responses are in blue):

21;
val it = 21 : int
(2 div 3);
val it = 0 : int
true;
val it = true : bool
"xyz";
val it = "xyz" : string
Basic SML Predefined Types

- **Unit**
  Its only value is (). Type `unit` is similar to `void` in C; it is used where a type is needed, but no "real" type is appropriate. For example, a call to a write function may return `unit` as its result.

- **Integer**
  Constants are sequences of digits. Negative values are prefixed with a `-` rather than a `-` (a `~` is a binary subtraction operator). For example, `-123` is negative 123.
  Standard operators include `+ - * div mod
  `< > `<=` `>=` `=` `<>`

For example, `real(3)` returns 3.0,
`floor(3.1)` returns 3,
`ceiling(3.3)` returns 4,
`round(~3.6)` returns ~4,
`trunc(3.9)` returns 3.
Mixed mode expressions, like `1 + 2.5` aren't allowed; you must do explicit conversion, like `real(1) + 2.5`

- **Strings**
  Strings are delimited by double quotes. Newlines are `, tabs are `, and `, escape double quotes and backslashes. E.g. "Bye now\n"
The `^` operator is concatenation.
"abc" ^ "def" = "abcdef"
The usual relational operators are provided: `< > `<=` `>=` `=` `<>`

- **Real**
  Both fractional (123.456) and exponent forms (10e7) are allowed. Negative signs and exponents use `~` rather than `-` (`-10.0e-12`).
  Standard operators include `+ - * /
  `< > `<=` `>=` `=` `<>`
  Note that `=` and `<>` aren't allowed! (Why?)
  Conversion routines include `real(int)` to convert an `int` to a `real`,
  `floor(real)` to take the floor of a `real`,
  `ceil(real)` to take the ceiling of a `real`.
  `round(real)` to round a `real`,
  `trunc(real)` to truncate a `real`.

- **Characters**
  Single characters are delimited by double quotes and prefixed by a `#`. For example, `"a"` or `"\t"`. A character is not a string of length one.
The `str` function may be used to convert a character into a string. Thus `str(#"a") = "a"

- **Boolean**
  Constants are `true` and `false`. Operators include `andalso` (short-circuit and), `orelse` (short-circuit or), `not`, `=`, and `<=`.
  A conditional expression,
  `(if boolval v1 else v2)` is available.
Tuples

A tuple type, composed of two or more values of any type is available. Tuples are delimited by parentheses, and values are separated by commas. Examples include:

(1, 2);
val it = (1, 2) : int * int

("xyz", 1=2);
val it = ("xyz", false) : string * bool

(1, 3.0, false);
val it = (1, 3.0, false) : int * real * bool

(1, 2, (3, 4));
val it = (1, 2, (3, 4)) : int * int * (int * int)

Equality is checked componentwise:

(1, 2) = (0+1, 1+1);
val it = true : bool

(1, 2, 3) = (1, 2) causes a compile-time type error (tuples must be of the same length and have corresponding types to be compared).

#i selects the i-th component of a tuple (counting from 1). Hence

#2 (1, 2, 3);
val it = 2 : int

Lists

Lists are required to have a single element type for all their elements; their length is unbounded. Lists are delimited by [ and ] and elements are separated by commas. Thus [1, 2, 3] is an integer list. The empty (or null) list is [] or nil.

The cons operator is ::

Hence [1, 2, 3] = 1::2::3::[]

Lists are automatically typed by ML:

[1, 2];
val it = [1, 2] : int list

Cons

Cons is an infix operator represented as ::

The left operand of :: is any value of type T.

The right operand of :: is any list of type T list.

The result of :: is a list of type T list.

Hence :: is polymorphic.

[] is the empty list. It has a type 'a list. The symbol 'a, read as "alpha" or "tic a" is a type variable.

Thus [] is a polymorphic constant.
**List Equality**

Two lists may be compared for equality if they are of the same type. Lists \( L_1 \) and \( L_2 \) are considered equal if:

1. They have the same number of elements
2. Corresponding members of the two lists are equal.

**List Operators**

- \( \text{hd} \equiv \text{head of list operator} \approx \text{car} \)
- \( \text{tl} \equiv \text{tail of list operator} \approx \text{cdr} \)
- \( \text{null} \equiv \text{null list predicate} \approx \text{null?} \)
- \( @ \equiv \text{infix list append operator} \approx \text{append} \)

**Records**

Their general form is

\[
\{ \text{name}_1 = \text{val}_1, \text{name}_2 = \text{val}_2, \ldots \}
\]

Field selector names are local to a record.

For example:

\[
\{a=1,b=2\};
\]

\[
\text{val it} = \{a=1,b=2\} : (a:\text{int}, b:\text{int})
\]

\[
\{a=1,b="xyz"\};
\]

\[
\text{val it} = \{a=1,b="xyz"\} : (a:\text{int}, b:\text{string})
\]

\[
\{a=1.0,b=\{c=[1,2]\}\};
\]

\[
\text{val it} = \{a=1.0,b=\{c=[1,2]\}\} : (a:\text{real}, b:\{c:\text{int list}\})
\]

The order of fields is irrelevant; equality is tested using field names.

\[
\{a=1,b=2\} = \{b=2,a=2-1\};
\]

\[
\text{val it} = \text{true} : \text{bool}
\]

\#id extracts the field named id from a record.

\[
\#b \{a=1,b=2\};
\]

\[
\text{val it} = 2 : \text{int}
\]

**Identifiers**

There are two forms:

- Alphanumeric (excluding reserved words)
    - Any sequence of letters, digits, single quotes and underscores; must begin with a letter or single quote. Case is significant. Identifiers that begin with a single quote are type variables.
    - Examples include:
    
    \[
    \text{abc a10 'polar sum_of_20}
    \]

- Symbolic
    - Any sequence (except predefined operators) of
    
    \[
    ! \& \# ^ | \_ \* \\
    \]
    
    Usually used for user-defined operators.
    - Examples include: ++ <-> !=
Comments

Of form
(* text *)
May cross line boundaries.

Declaration of Values

The basic form is
val id = expression;
This defines id to be bound to expression; ML answers with the name and value defined and the inferred type.
For example
val x = 10*10;
val x = 100 : int

Redefinition of an identifier is OK, but this is redefinition not assignment;
Thus
val x = 100;
val x = (x=100);
is fine; there is no type error even though the first x is an integer and then it is a boolean.
val x = 100 : int
val x = true : bool

Examples

val x = 1;
val x = 1 : int
val z = (x,x,x);
val z = (1,1,1) : int * int * int
val L = [z,z];
val L = [(1,1,1),(1,1,1)] : (int * int * int) list
val r = {a=L};
val r = {a=[(1,1,1),(1,1,1)]} : {a:(int * int * int) list}
After rebinding, the “nearest” (most recent) binding is used.

The and symbol (not boolean and) is used for simultaneous binding:
val x = 10;
val x = 10 : int
val x = true and y = x;
val x = true : bool
val y = 10 : int

Local definitions are temporary value definitions:
local
  val x = 10
in
  val u = x*x;
end;
val u = 100 : int
Let bindings are used in expressions:

```ml
let
  val x = 10
in
  5 * x
end;
val it = 50 : int
```

Patterns

Scheme (and most other languages) use access or decomposition functions to access the components of a structured object. Thus we might write

```ml
(let ( (h (car L)) (t (cdr L)))
  body)
```

Here `car` and `cdr` are used as access functions to locate the parts of `L` we want to access.

In ML we can access components of lists (or tuples, or records) directly by using patterns. The context in which the identifier appears tells us the part of the structure it references.

```ml
val x = (1,2);
val x = (1,2) : int * int
val (h,t) = x;
val h = 1 : int
val t = 2 : int
val L = [1,2,3];
val L = [1,2,3] : int list
val [v1,v2,v3] = L;
val v1 = 1 : int
val v2 = 2 : int
val v3 = 3 : int
val [1,x,3] = L;
(* This is illegal. Why? *)
val yy::rest = L;
val yy = 1 : int
val rest = [2,3] : int list
```

Wildcards

An underscore (`_`) may be used as a "wildcard" or "don't care" symbol. It matches part of a structure without defining a new binding.

```ml
val zz::_ = L;
val zz = 1 : int
```

Pattern matching works in records too.

```ml
val r = {a=1,b=2};
val r = {a=1,b=2} :
  {a:int, b:int}
val {a=va,b=vb} = r;
val va = 1 : int
val vb = 2 : int
val {a=wa,b=_}=r;
val wa = 1 : int
val {a=za, ...}=r;
val za = 1 : int
```
Patterns can be nested too.

```ml
val x = ((1,3.0),5);
val x = ((1,3.0),5) : (int * real) * int
val ((1,y),_) = x;
val y = 3.0 : real
```

### Functions

Functions take a single argument (which can be a tuple).

Function calls are of the form

```
function_name argument;
```

For example

```
size "xyz";
cos 3.14159;
```

The more conventional form

```
size("xyz"); or cos(3.14159);
```

is OK (the parentheses around the argument are allowed, but unnecessary).

The form `size "xyz"` or `cos 3.14159` is OK too.

---

Note that the call

```
plus(1,2);
```

passes one argument, the tuple `(1,2)` to `plus`.

The call `dummy();`

passes one argument, the unit value, to `dummy`.

All parameters are passed by value.

---

### Function Types

The type of a function in ML is denoted as `T1 -> T2`. This says that a parameter of type `T1` is mapped to a result of type `T2`.

The symbol `fn` denotes a value that is a function.

Thus

```
size;
val it = fn : string -> int
not;
val it = fn : bool -> bool
Math.cos;
val it = fn : real -> real
```

(`Math` is an ML structure—an external library member that contains separately compiled definitions).
User-Defined Functions

The general form is

fun name arg = expression;

ML answers back with the name defined, the fact that it is a function (the \texttt{fn} symbol) and its inferred type.

For example,

\begin{verbatim}
fun twice x = 2*x;
val twice = fn : int -> int

fun twotimes(x) = 2*x;
val twotimes = fn : int -> int

fun fact n = 
  if n=0 
  then 1 
  else n*fact(n-1);
val fact = fn : int -> int
\end{verbatim}

The \texttt{:\texttt{int}} suffix is a type constraint. It is needed to help ML decide that \texttt{+} is integer plus rather than real plus.

Patterns In Function Definitions

The following defines a predicate that tests whether a list, \(L\) is null (the predefined \texttt{null} function already does this).

\begin{verbatim}
fun isNull L = 
  if L=[] then true else false;
val isNull = fn : 'a list -> bool
\end{verbatim}

However, we can decompose the definition using patterns to get a simpler and more elegant definition:

\begin{verbatim}
fun isNull [] = true
  | isNull(_::_) = false;
val isNull = fn : 'a list -> bool
\end{verbatim}

The "|" divides the function definition into different argument patterns; no explicit conditional logic is needed. The definition that matches a particular actual parameter is automatically selected.

\begin{verbatim}
fun fact(1) = 1 
  | fact(n) = n*fact(n-1);
val fact = fn : int -> int
\end{verbatim}

If patterns that cover all possible arguments aren't specified, you may get a run-time \texttt{Match} exception. If patterns overlap you may get a warning from the compiler.
fun append([],L) = L
| append(hd::tl,L) = hd::append(tl,L);

val append = fn :
  'a list * 'a list -> 'a list

If we add the pattern
append(L,[]) = L

we get a redundant pattern warning (Why?)

fun append([],L) = L
| append(hd::tl,L) = hd::append(tl,L)
| append(L,[]) = L;

stdIn:151.1-153.20 Error: match redundant
  (nil,L) => ...
  (hd :: tl,L) => ...
  --> (L,nil) => ...

But a more precise decomposition is fine:

fun append([],L) = L
| append(hd::tl,hd2::tl2) = hd::append(tl,hd2::tl2)
| append(hd::tl,[]) = hd::tl;

val append = fn :
  'a list * 'a list -> 'a list

Function Types Can be Polytypes

Recall that 'a, 'b, ... represent type variables. That is, any valid type may be substituted for them when checking type correctness.

ML said the type of append is

val append = fn :
  'a list * 'a list -> 'a list

Why does 'a appear in three places?

We can define eitherNull, a predicate that determines whether either of two lists is null as

fun eitherNull(L1,L2) = null(L1) orelse null(L2);

val eitherNull =
  fn : 'a list * 'b list -> bool

Why are both 'a and 'b used in eitherNull's type?

Currying

ML chooses the most general (least-restrictive) type possible for user-defined functions.

Functions are first-class objects, as in Scheme.

The function definition

fun f x y = expression;

defines a function f (of x) that returns a function (of y).

Reducing multiple argument functions to a sequence of one argument functions is called currying (after Haskell Curry, a mathematician who popularized the approach).
Thus

fun f x y = x :: [y];
val f = fn : 'a -> 'a -> 'a list

says that $f$ takes a parameter $x$, of type $'a$, and returns a function (of $y$, whose type is $'a$) that returns a list of $'a$.

Contrast this with the more conventional

fun g(x,y) = x :: [y];
val g = fn : 'a * 'a -> 'a list

Here $g$ takes a pair of arguments (each of type $'a$) and returns a value of type $'a$ list.

The advantage of currying is that we can bind one argument and leave the remaining argument(s) free.

For example

$f(1)$;
is a legal call. It returns a function of type

fn : int -> int list

The function returned is equivalent to

fun h b = 1 :: [b];
val h = fn : int -> int list

Map Revisited

ML supports the map function, which can be defined as

fun map(f,[]) = []
| map(f,x::y) = (f x) :: map(f,y);
val map = fn : ('a -> 'b) * 'a list -> 'b list

This type says that $\text{map}$ takes a pair of arguments. One is a function from type $'a$ to type $'b$. The second argument is a list of type $'a$. The result is a list of type $'b$.

In curried form $\text{map}$ is defined as

fun map f [] = []
| map f (x::y) = (f x) :: map f y;
val map = fn : ('a -> 'b) -> 'a list -> 'b list

This type says that $\text{map}$ takes one argument that is a function from type $'a$ to type $'b$. It returns a function that takes an argument that is a list of type $'a$ and returns a list of type $'b$.

The advantage of the curried form of $\text{map}$ is that we can now use $\text{map}$ to create “specialized” functions in which the function that is mapped is fixed.

For example,

val neg = map not;
val neg = fn : bool list -> bool list
neg [true,false,true];
val it = [false,true,false] : bool list
Power Sets Revisited

Let's compute power sets in ML.

We want a function `pow` that takes a list of values, viewed as a set, and which returns a list of lists. Each sublist will be one of the possible subsets of the original argument.

For example,

\[ \text{pow } [1,2] = \text{ [[1,2], [1], [2], []]} \]

We first define a version of `cons` in curried form:

```
fun cons h t = h::t;
val cons = fn : 'a -> 'a list -> 'a list
```

Now we define `pow`. We define the powerset of the empty list, [], to be [[]]. That is, the power set of the empty set is set that contains only the empty set.

For a non-empty list, consisting of `h::t`, we compute the power set of `t`, which we call `pset`. Then the power set for `h::t` is just `h` distributed through `pset` appended to `pset`.

We distribute `h` through `pset` very elegantly: we just map the function `(cons h)` to `pset`. `(cons h)` adds `h` to the head of any list it is given. Thus mapping `(cons h)` to `pset` adds `h` to all lists in `pset`.

The complete definition is simply

```
fun pow [] = [[]]
|  pow (h::t) = 
  let
    val pset = pow t
  in
    (map (cons h) pset) @ pset
  end;
val pow = 
  fn : 'a list -> 'a list list
```

Let's trace the computation of `pow [1,2]`.


Now `h = 2` and `t = []`.

We know `pow [1] = [[]]`, so `pow [2] = (map (cons 2) [[]])@[[]] = ([[2]])@[[]] = [[2], []]`

Therefore `pow [1,2] = (map (cons 1) [[2], [1]]) @ [[2], [1]] = [[1,2], [1], [2], []]`. 

Therefore `pow [1,2] = ([1,2], [1], [2], [])`. 

```
Composing Functions

We can define a composition function that composes two functions into one:
fun comp \( (f, g)(x) = f(g(x)) \);
val comp = fn :
('a -> 'b) * ('c -> 'a) ->
'c -> 'b
In curried form we have
fun comp f g x = f(g(x));
val comp = fn :
('a -> 'b) ->
('c -> 'a) -> 'c -> 'b
For example,
fun sqr x:int = x*x;
val sqr = fn : int -> int
comp sqr sqr;
val it = fn : int -> int

Comp sqrs sqrs 3;
val it = 81 : int
In SML \(\circ\) (lower-case O) is the infix composition operator.
Hence
sqr \(\circ\) sqr \(\equiv\) comp sqr sqs

Lambda Terms

ML needs a notation to write down unnamed (anonymous) functions, similar to the lambda expressions Scheme uses.
That notation is
fn arg => body;
For example,
val sqr = fn x:int => x*x;
val sqr = fn : int -> int
comp sqr sqr;
val it = fn : int -> int

An anonymous function can be used wherever a function value is needed.
For example,
map (fn x => [x]) [1,2,3];
val it =
[[1],[2],[3]] : int list list
We can use patterns too:
(fn [] => []
| (h::t) => h::h::t);
val it = fn : 'a list -> 'a list
(What does this function do?)
Polymorphism vs. Overloading

ML supports polymorphism.
A function may accept a polytype (a set of types) rather than a single fixed type.
In all cases, the same function definition is used. Details of the supplied type are irrelevant and may be ignored.
For example,

```ml
fun id x = x;
val id = fn : 'a -> 'a
fun toList x = [x];
val toList = fn : 'a -> 'a list
```

Overloading, as in C++ and Java, allows alternative definitions of the same method or operator, with selection based on type.
Thus in Java + may represent integer addition, floating point addition or string concatenation, even though these are really rather different operations.
In ML +, −, * and = are overloaded.
When = is used (to test equality), ML deduces that an equality type is required. (Most, but not all, types can be compared for equality).
When ML decides an equality type is needed, it uses a type variable that begins with two tics rather than one.

```ml
fun eq(x,y) = (x=y);
val eq = fn : ''a * ''a -> bool
```

Defining New Types in ML

We can create new names for existing types (type abbreviations) using
`type id = def;`
For example,

```ml
type triple = int*real*string;
type triple = int * real * string
type rec1 =
  {a:int,b:real,c:string};
type rec1 =
  {a:int, b:real, c:string}
type 'a triple3  = 'a*'a*'a;
type 'a triple3 = 'a * 'a * 'a
type intTriple = int triple3;
type intTriple = int triple3
```

These type definitions are essentially macro-like name substitutions.

The Datatype Mechanism

New types are defined using the `datatype` mechanism, which specifies new data value constructors.
For example,

```ml
datatype color =
  red|blue|green;
datatype color =
  blue | green | red
```

Pattern matching works on user-defined types using their constructors:

```ml
fun translate red = "rot"
  | translate blue = "blau"
  | translate green = "gruen";
val translate =
  fn : color -> string
```
fun jumble red = blue
  |  jumble blue = green
  |  jumble green = red;
val jumble = fn : color -> color
translate (jumble green);
val it = "rot" : string

SML Examples
Source code for most of the SML examples presented here may be found in
~cs538-1/public/sml/class.sml

Parameterized Constructors
The constructors used to define data types may be parameterized:
datatype money =
  none
  |  coin of int
  |  bill of int
  |  iou of real * string;
datatype money =
  bill of int | coin of int
  |  iou of real * string | none

Now expressions like coin(25) or bill(5) of iou(10.25,"Lisa") represent valid values of type money.

We can also define values and functions of type money:
val dime = coin(10);
val dime = coin 10 : money
val deadbeat =
  iou(25.00,"Homer Simpson");
val deadbeat =
  iou (25.0,"Homer Simpson") : money
fun amount(none) = 0.0
  |  amount(coin(cents)) =
    real(cents)/100.0
  |  amount(bill(dollars)) =
    real(dollars)
  |  amount(iou(amt,_)) =
    0.5*amt;
val amount = fn : money -> real

Polymorphic Datatypes
A user-defined data type may be polymorphic. An excellent example is
datatype 'a option =
  none | some of 'a;
datatype 'a option =
  none | some of 'a
val zilch = none;
val zilch = none : 'a option
val mucho =some(10e10);
val mucho =
  some 100000000000.0 : real option
type studentInfo =
  {name:string,
   ssNumber:int option};
type studentInfo = {name:string,
   ssNumber:int option}
val newStudent = {name="Mystery Man", ssNumber=none} : studentInfo;
val newStudent = {name="Mystery Man", ssNumber=none} : studentInfo

Datatypes may be Recursive

Recursive datatypes allow linked structures without explicit pointers.

datatype binTree =
  null
| leaf
| node of binTree * binTree;
datatype binTree =
  leaf | node of binTree * binTree | null
fun size(null) = 0
| size(leaf) = 1
| size(node(t1,t2)) =
  size(t1)+size(t2) + 1
val size = fn : binTree -> int

Recursive Datatypes may be Polymorphic

datatype 'a binTree =
  null
| leaf of 'a
| node of 'a binTree * 'a binTree
datatype 'a binTree =
  leaf of 'a | node of 'a binTree * 'a binTree | null
fun frontier(null) = []
| frontier(leaf(v)) = [v]
| frontier(node(t1,t2)) =
  frontier(t1) @ frontier(t2)
val frontier =
  fn : 'a binTree -> 'a list

We can model n-ary trees by using lists of subtrees:

datatype 'a Tree =
  null
| leaf of 'a
| node of 'a Tree list;
datatype 'a Tree = leaf of 'a | node of 'a Tree list | null
fun frontier(null) = []
| frontier(leaf(v)) = [v]
| frontier(node(h::t)) =
  frontier(h) @
  frontier(node(t))
| frontier(node([])) = []
val frontier = fn :
  'a Tree -> 'a list
Abstract Data Types

ML also provides abstract data types in which the implementation of the type is hidden from users. The general form is

abstype name = implementation with
val and fun definitions end;

Users may access the name of the abstract type and the val and fun definitions that follow the with, but the implementation may be used only with the body of the abstype definition.

Example

abstype 'a stack =
  stk of 'a list
with
  val Null = stk([], [])
  fun empty(stk([])) = true
  | empty(stk(_::_)) = false
  fun top(stk(h::_)) = h
  fun pop(stk(_::t)) = stk(t)
  fun push(v, stk(L)) = stk(v :: L)
end

type 'a stack
val Null = - : 'a stack
val empty = fn : 'a stack -> bool
val top = fn : 'a stack -> 'a
val pop =
  fn : 'a stack -> 'a stack
val push = fn :
  'a * 'a stack -> 'a stack

Local value and function definitions, not to be exported to users of the type can be created using the local definition mechanism described earlier:

local
val and fun definitions in
exported definitions end;

abstype 'a stack =
  stk of 'a list
with
  local
    fun size(stk(L)) = length(L);
  in
    val Null = stk([], [])
    fun empty(s) =
      (size(s) = 0)
    fun top(stk(h::_)) = h
    fun pop(stk(_::t)) = stk(t)
    fun push(v, stk(L)) = stk(v :: L)
  end
end
type 'a stack
val Null = - : 'a stack
val empty = fn : 'a stack -> bool
val top = fn : 'a stack -> 'a
val pop = fn :
  'a stack -> 'a stack
val push = fn :
  'a * 'a stack -> 'a stack
Why are abstract data types useful? Because they hide an implementation of a type from a user, allowing implementation changes without any impact on user programs.

Consider a simple implementation of queues:

```plaintext
abstype 'a queue =
  q of 'a list
with
  val Null = q([],)
  fun front(q(h::_)) = h
  fun rm(q(_::t)) = q(t)
  fun enter(v,q(L)) =
    q(rev(v::rev(L)))
end

type 'a queue
val Null = - : 'a queue
val front = fn : 'a queue -> 'a
```

This implementation of queues is valid, but somewhat inefficient. In particular to enter a new value onto the rear end of a queue, we do the following:

```plaintext
fun enter(v,q(L)) =
  q(rev(v::rev(L)))
```

We reverse the list that implements the queue, add the new value to the head of the reversed queue then reverse the list a second time.

A more efficient (but less obvious) implementation of a queue is to store it as two lists. One list represents the “front” of the queue. It is from this list that we extract the front value, and from which we remove elements.

The other list represents the “back” of the queue (in reversed order). We add elements to the rear of the queue by adding elements to the front of the list. From time to time, when the front list becomes null, we “promote” the rear list into the front list (by reversing it). Now access to both the front and the back of the queue is fast and direct. The new implementation is:

```plaintext
abstype 'a queue =
  q of 'a list * 'a list
with
  val Null = q([],[])
  fun front(q(h::_,_)) = h
  |  front(q([],L)) =
      front(q(rev(L),[]))
  fun rm(q(_::t,L)) = q(t,L)
  |  rm(q([],L)) =
      rm(q(rev(L),[]))
  fun enter(v,q(L1,L2)) =
    q(L1,v::L2)
end

type 'a queue
val Null = - : 'a queue
val front = fn : 'a queue -> 'a
val rm = fn : 'a queue -> 'a
val enter = fn : 'a * 'a queue -> 'a queue
```
From the user’s point of view, the two implementations are identical (they export exactly the same set of values and functions). Hence the new implementation can replace the old implementation without any impact at all to the user (except, of course, performance!).

Exception Handling

Our definitions of stacks and queues are incomplete. Reconsider our definition of stack:

```
abstype 'a stack = stk of 'a list
with
  val Null = stk([])
  fun empty(stk([])) = true
    | empty(stk(_::_)) = false
  fun top(stk(h::_)) = h
  fun pop(stk(_::t)) = stk(t)
  fun push(v, stk(L)) = stk(v::L)
end
```

What happens if we evaluate `top(Null);`

We get a “match failure” because our definition of `top` is incomplete!

In ML we can raise an exception if an illegal or unexpected operation occurs. Asking for the top of an empty stack ought to raise an exception since the requested value does not exist.

ML contains a number of predefined exceptions, including

Match Empty Div Overflow (exception names by convention begin with a capital letter).

Predefined exceptions are raised by illegal values or operations. If they are not caught, the run-time system prints an error message.

```
fun f(1) = 2;
val f = fn : int -> int
f(2);
uncaught exception nonexhaustive match failure
hd[];
uncaught exception Empty
1000000*1000000;
uncaught exception overflow
(1 div 0);
uncaught exception divide by zero
1.0/0.0;
val it = inf : real
(inf is the IEEE floating-point standard “infinity” value)
```
User Defined Exceptions

New exceptions may be defined as
exception name;
or
exception name of type;
For example
exception IsZero;
exception IsZero
exception NegValue of real;
exception NegValue of real

Exceptions May be Raised

The raise statement raises (throws) an exception:
raise exceptionName;
or
raise exceptionName(expr);
For example
fun divide(a,0) = raise IsZero
| divide(a,b) = a div b;
val divide = fn : int * int -> int
divide(10,3);
val it = 3 : int
divide(10,0);
uncaught exception IsZero

val sqrt = Real.Math.sqrt;
val sqrt = fn : real -> real
fun sqroot(x) = 
  if x < 0.0
  then raise NegValue(x)
  else sqrt(x);
val sqroot = fn : real -> real
sqroot(2.0);
val it = 1.41421356237 : real
sqroot(~2.0);
uncaught exception NegValue

Exception Handlers

You may catch an exception by defining a handler for it:
(expr) handle exception1 => val1
|| exception2 => val2
|| ... ;

For example,
(sqroot ~100.0)
| handle NegValue(v) =>
  (sqrt (~v));
val it = 10.0 : real
Stacks Revisited

We can add an exception, `EmptyStk`, to our earlier stack type to handle `top` or `pop` operations on an empty stack:

```sml
abstype 'a stack = stk of 'a list with
  val Null = stk([])
  exception EmptyStk
  fun empty(stk([])) = true |
    empty(stk(_::_)) = false
  fun top(stk(h::_)) = h |
    top(stk([])) = raise EmptyStk
  fun pop(stk(_::t)) = stk(t) |
    pop(stk([])) = raise EmptyStk
  fun push(v, stk(L)) = stk(v::L)
end
```

User-Defined Operators

SML allows users to define symbolic operators composed of non-alphanumeric characters. This means operator-like symbols can be created and used. Care must be taken to avoid predefined operators (like `+`, `-`, `^`, `@`, etc.).

If we wish, we can redo our stack definition using symbols rather than identifiers.

We might choose the following symbols:

- `top |=`
- `pop <=>`
- `push ==>`
- `null <&>`
- `empty <?>`

Now we can have expressions like

```sml```
```sml```
```sml```
```sml```
```sml```
```sml```
```sml```
```sml```
level is an integer representing the “precedence” level of the infix operator. 0 is the lowest precedence level; higher precedence operators are applied before lower precedence operators (in the absence of explicit parentheses).

infix defines a left-associative operator (groups from left to right). infixr defines a right-associative operator (groups from right to left).

Thus

fun cat(L1, L2) = L1 @ L2;
infix 5 cat

makes cat a left-assicative infix operator at the same precedence level as @. We can now write

[1,2] cat [3,4,5] cat [6,7];

val it = [1,2,3,4,5,6,7] : int list

The standard predefined operators have the following precedence levels:

<table>
<thead>
<tr>
<th>Level</th>
<th>Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>o</td>
</tr>
<tr>
<td>4</td>
<td>= &lt;&gt; &lt; &gt; &lt;= &gt;=</td>
</tr>
<tr>
<td>5</td>
<td>:: @</td>
</tr>
<tr>
<td>6</td>
<td>+ - ^</td>
</tr>
<tr>
<td>7</td>
<td>* / div mod</td>
</tr>
</tbody>
</table>

If we define ==> (push) as

infixr 2 ==> then

1 ==> 2+3 ==> <@

will work as expected, evaluating expressions like 2+3 before doing any pushes, with pushes done right to left.

abstype 'a stack =

stk of 'a list

with

val <@> = stk([])

exception emptyStk
fun <?>(stk([])) = true
| <?>(stk(_::_)) = false

fun |==(stk(h:_)) = h
| |==(stk([])) = raise emptyStk

fun <==(stk(_::t)) = stk(t)
| <==(stk([])) = raise emptyStk

fun ==>=(v, stk(L)) =

stk(v::L)
infixr 2 ==>=

end

type 'a stack

val <@> = - : 'a stack

exception emptyStk
val ?<> = fn : 'a stack -> bool
val |= = fn : 'a stack -> 'a
val <== = fn :

'a stack -> 'a stack
val ==> = fn : 'a * 'a stack -> 'a stack

infixr 2 ==>=

Now we can write

val myStack =

1 ==> 2+3 ==> <@

val myStack = - : int stack

|= myStack;
val it = 1 : int
|= (<== myStack);
val it = 5 : int
Using Infix Operators as Values

Sometimes we simply want to use an infix operator as a symbol whose value is a function.
For example, given
fun dupl f v = f(v,v);
val dupl = fn : ('a * 'a -> 'b) -> 'a -> 'b
we might try the call
dupl ^ "abc";
This fails because SML tries to parse
dupl and "abc" as the operands of ^.
To pass an operator as an ordinary function value, we prefix it with op
which tells the SML compiler that the following symbol is an infix operator.

Thus
dupl op ^ "abc";
val it = "abcabc" : string
works fine.

The Case Expression

ML contains a case expression patterned on switch and case statements found in other languages.
As in function definitions, patterns are used to choose among a variety of values.
The general form of the case is
case expr of
  pattern1 => expr1 |
  pattern2 => expr2 |
  ...
  patternn => exprn;
If no pattern matches, a Match exception is thrown.
It is common to use _ (the wildcard) as the last pattern in a case.

Examples include
case c of
  red   => "rot" |
  blue  => "blau" |
  green => "gruen";

case pair of
  (1,_) => "win" |
  (2,_) => "place" |
  (3,_) => "show" |
  (_,_) => "loser";

case intOption of
  none => 0 |
  some(v) => v;
**Imperative Features of ML**

ML provides references to heap locations that may be updated. This is essentially the same as access to heap objects via references (Java) or pointers (C and C++).

The expression

\[ \text{ref } \text{val} \]

creates a reference to a heap location initialized to val. For example,

\[ \text{ref } 0; \]

\[ \text{val } \text{it} = \text{ref } 0 : \text{int ref} \]

The prefix operator \(!\) fetches the value contained in a heap location (just as \(*\) dereferences a pointer in C or C++).

Thus

\[ ! (\text{ref } 0); \]

\[ \text{val } \text{it} = 0 : \text{int} \]

**Sequential Composition**

Expressions or statements are sequenced using \(;\). Hence

\[ \text{val } \text{a} = (1+2;3+4); \]

\[ \text{val } \text{a} = 7 : \text{int} \]

\[ (\text{x}:=1;!\text{x}); \]

\[ \text{val } \text{it} = 1 : \text{int} \]

**Iteration**

while expr1 do expr2

implements iteration (and returns unit); Thus

\[ (\text{while false do 10}); \]

\[ \text{val } \text{it} = () : \text{unit} \]

while \(!\text{x} > 0\) do \text{x}:= \!\text{x}-1;

\[ \text{val } \text{it} = () : \text{unit} \]

\[ \text{!x}; \]

\[ \text{val } \text{it} = 0 : \text{int} \]

**Simple I/O**

The function

\[ \text{print}; \]

\[ \text{val } \text{it} = \text{fn} : \text{string }\rightarrow \text{unit} \]

prints a string onto standard output. For example,

\[ \text{print("Hello World\n")}; \]

Hello World

The conversion routines

\[ \text{Real.toString}; \]

\[ \text{val } \text{it} = \text{fn} : \text{real }\rightarrow \text{string} \]

\[ \text{Int.toString}; \]

\[ \text{val } \text{it} = \text{fn} : \text{int }\rightarrow \text{string} \]

\[ \text{Bool.toString}; \]

\[ \text{val } \text{it} = \text{fn} : \text{bool }\rightarrow \text{string} \]

convert a value (real, int or bool) into a string. Unlike Java, the call must be explicit.
For example,
\[
\text{print(Int.toString(123))};
\]
123
Also available are
\[
\text{Real.fromString};
\]
\[
\text{val it = fn : string -> real option}
\]
\[
\text{Int.fromString};
\]
\[
\text{val it = fn : string -> int option}
\]
\[
\text{Bool.fromString};
\]
\[
\text{val it = fn : string -> bool option}
\]
which convert from a string to a real or int or bool if possible.
(That's why the option type is used).

For example,
\[
\text{case (Int.fromString("123")) of}
\]
\[
\text{SOME(i) => i | NONE => 0;}
\]
\[
\text{val it = 123 : int}
\]
\[
\text{case (Int.fromString("One two three")) of}
\]
\[
\text{SOME(i) => i | NONE => 0;}
\]
\[
\text{val it = 0 : int}
\]

Text I/O

The structure TextIO contains a wide variety of I/O types, values and functions. You load these by entering:
\[
\text{open TextIO;}
\]
Among the values loaded are
- type instream
  This is the type that represents input text files.
- type outstream
  This is the type that represents output text files.
- type vector = string
  Makes vector a synonym for string.
- type elem = char
  Makes elem a synonym for char.
- \[
\text{val stdIn : instream}
\]
- \[
\text{val stdOut : outstream}
\]
- \[
\text{val stdErr : outstream}
\]
  Predefined input and output streams.
- \[
\text{val openIn : string -> instream}
\]
- \[
\text{val openOut : string -> outstream}
\]
  Open an input or output stream. For example,
- \[
\text{val out = openOut("/tmp/test1");}
\]
- \[
\text{val out = - : outstream}
\]
- \[
\text{val input : instream -> vector}
\]
  Read a line of input into a string (vector is defined as equivalent to string). For example (user input is in red):
val s = input(stdIn);
Hello!
val s = "Hello!\n" : vector
• val inputN :
  instream * int -> vector
  Read the next N input characters into
  a string. For example,
  val t = inputN(stdIn, 3);
  abcde
  val t = "abc" : vector
• val inputAll :
  instream -> vector
  Read the rest of the input file into a
  string (with newlines separating
  lines). For example,
  val u = inputAll(stdIn);
  Four score and
  seven years ago ...
  val u = "Four score and\n\nseven years ago ...\n" : vector

String Operations

ML provides a wide variety of string manipulation routines. Included are:
• The string concatenation operator, ^
  "abc" ^ "def" = "abcdef"
• The standard 6 relational operators:
  < > <= >= = <>
• The string size operator:
  val size : string -> int
  size ("abcd");
  val it = 4 : int
• The string subscripting operator
  (indexing from 0):
  val sub =
    fn : string * int -> char
    sub("abcde", 2);
  val it = #"c" : char

• The substring function
  val substring :
    string * int * int -> string
  This function is called as
  substring(string, start, len)
  start is the starting position,
  counting from 0.
  len is the length of the desired
  substring. For example,
  substring("abcdefghij", 3, 4)
  val it = "defg" : string
• Concatenation of a list of strings into
  a single string:
  concat :
    string list -> string
  For example,
  concat ["What’s", " up", "?"];
  val it = "What’s up?" : string
Convert a character into a string:

\[ \text{str : char -> string} \]

For example,

\[ \text{str(#"x")}; \]
\[ \text{val it = "x" : string} \]

“Explode” a string into a list of characters:

\[ \text{explode : string -> char list} \]

For example,

\[ \text{explode("abcde")}; \]
\[ \text{val it = ["a","b","c","d","e"] : char list} \]

“Implode” a list of characters into a string.

\[ \text{implode : char list -> string} \]

For example,

\[ \text{implode ["a","b","c","d","e"]}; \]
\[ \text{val it = "abcde" : string} \]

---

**Structures and Signatures**

In C++ and Java you can group variable and function definitions into classes. In Java you can also group classes into packages.

In ML you can group value, exception and function definitions into structures.

You can then import selected definitions from the structure (using the notation \text{structure.name}) or you can open the structure, thereby importing all the definitions within the structure.

(Examples used in this section may be found at \texttt{~cs538-1/public/sml/struct.sml})

---

The general form of a structure definition is

\[ \text{structure name = struct} \]
\[ \text{val, exception and} \]
\[ \text{fun definitions} \]
\[ \text{end} \]

For example,

\[ \text{structure Mapping = struct} \]
\[ \text{exception NotFound;} \]
\[ \text{val create = [];} \]
\[ \text{fun lookup(key,[])} = \]
\[ \text{raise NotFound} \]
\[ | \text{lookup(key, (key1,value1)::rest)} = \]
\[ \text{if key = key1} \]
\[ \text{then value1} \]
\[ \text{else lookup(key,rest);} \]

---

fun insert(key,value,[]) =
{} [(key,value)]
| insert(key,value,
 (key1,value1)::rest) =
 if key = key1
 then (key1,value)::rest
 else (key1,value1)::
 insert(key,value,rest);
end;

We can access members of this structure as Mapping.name. Thus

\[ \text{Mapping.insert(538,"languages",[]);} \]
\[ \text{val it = [(538,"languages")]} : \]
\[ \text{(int * string) list} \]
\[ \text{open Mapping;} \]
\[ \text{exception NotFound} \]
\[ \text{val create : 'a list} \]
\[ \text{val insert : 'a * 'b * ('a * 'b) list -> ('a * 'b) list} \]
\[ \text{val lookup : 'a * ('a * 'b) list -> 'b} \]
Signatures
Each structure has a signature, which is its type.
For example, Mapping's signature is

```ml
structure Mapping : 
  sig 
    exception NotFound 
    val create : 'a list 
    val insert : ''a * 'b * 
       (''a * 'b) list -> 
       (''a * 'b) list 
    val lookup : ''a * 
       (''a * 'b) list -> 'b 
  end 
```

You can define a signature as

```ml
signature name = sig 
  type definitions for values, 
  functions and exceptions 
end 
```

For example,

```ml
signature Str2IntMapping = 
  sig 
    exception NotFound; 
    val lookup: 
      string * (string*int) list 
      -> int; 
  end; 
```

Signatures can be used to
- Restrict the type of a value or function in a structure.
- Hide selected definitions that appear in a structure.
For example

```ml
structure Str2IntMap : 
  Str2IntMapping = Mapping; 
defines a new structure, Str2IntMap, 
created by restricting Mapping to the 
Str2IntMapping signature. When 
we do this we get 
open Str2IntMap; 
  exception NotFound 
  val lookup : string * 
     (string * int) list -> int 
Only lookup and NotFound are 
created, and lookup is limited to keys 
that are strings.
```

Extending ML's Polymorphism
In languages like C++ and Java we must use types like void* or Object to simulate the polymorphism that ML provides. In ML whenever possible a general type (a polytype) is used rather than a fixed type. Thus in

```ml
fun len([]) = 0 
| len(a::b) = 1 + len(b); 
```
we get a type of

```
'a list -> int 
```
because this is the most general type possible that is consistent with len's definition.

Is this form of polymorphism general enough to capture the general idea of making program definitions as type-independent as possible?
It isn’t, and to see why consider the following ML definition of a merge sort. A merge sort operates by first splitting a list into two equal length sublists. The following function does this:

```ml
fun split [] = ([],[])
| split [a] = ([a],[])
| split (a::b::rest) = 
  let val (left,right) = 
    split(rest) in 
    (a::left, b::right)
  end;
```

After the input list is split into two halves, each half is recursively sorted, then the sorted halves are merged together into a single list.

The following ML function merges two sorted lists into one:

```ml
fun merge([],[]) = []
| merge([],hd::tl) = hd::tl
| merge(hd::tl,[],) = hd::tl
| merge(hd::tl,h::t) = 
  if hd <= h 
  then hd::merge(tl,h::t)
  else h::merge(hd::tl,t)
```

With these two subroutines, a definition of a sort is easy:

```ml
fun sort [] = []
| sort([a]) = [a]
| sort(a::b::rest) = 
  let val (left,right) = 
    split(a::b::rest) in 
    merge(sort(left),sort(right))
  end;
```

This definition looks very general—it should work for a list of any type. Unfortunately, when ML types the functions we get a surprise:

```ml
val split = fn : 'a list -> 'a list * 'a list
val merge = fn : int list * int list -> int list
val sort = fn : int list -> int list
```

split is polymorphic, but merge and sort are limited to integer lists!

Where did this restriction come from?

The problem is that we did a comparison in merge using the <= operator, and ML typed this as an integer comparison.

We can make our definition of sort more general by adding a comparison function, le(a,b) as a parameter to merge and sort. If we curry this parameter we may be able to hide it from end users. Our updated definitions are:

```ml
fun merge(le, [], []) = []
| merge(le, [], hd::tl) = hd::tl
| merge(le, hd::tl, []) = hd::tl
| merge(le, hd::tl, h::t) = 
  if le(hd,h) 
  then hd::merge(le,tl,h::t)
  else h::merge(le,hd::tl,t)
```
fun sort le [] = []
|    sort le [a] = [a]
|    sort le (a::b::rest) =
        let val (left,right) = split(a::b::rest) in
            merge(le, sort le left, sort le right)
        end;

Now the types of merge and sort are:

val merge = fn :
 ('a * 'a -> bool) *
   'a list * 'a list -> 'a list
val sort = fn : ('a * 'a -> bool) -> 'a list

We can now “customize” sort by choosing a particular definition for the le parameter:

fun le(a,b) = a <= b;
val le = fn : int * int -> bool

fun intsort L = sort le L;
val intsort =
    fn : int list -> int list
intsort( [4,9,0,2,111,~22,8,~123]);
val it = [-123, ~22, 0, 2, 4, 8, 9, 111] : int list

fun strle(a:string,b) =
    a <= b;
val strle =
    fn : string * string -> bool

fun strsort L = sort strle L;
val strsort =
    fn : string list -> string list
strsort( ["aac","aaa","ABC","123"]);
val it = ["123","ABC","aaa","aac"] :
    string list

Making the comparison relation an explicit parameter works, but it is a bit ugly and inefficient. Moreover, if we have several functions that depend on the comparison relation, we need to ensure that they all use the same relation. Thus if we wish to define a predicate inOrder that tests if a list is already sorted, we can use:

fun inOrder le [] = true
|    inOrder le [a] = true
|    inOrder le (a::b::rest) =
        le(a,b) andalso
        inOrder le (b::rest);
val inOrder = fn :
    ('a * 'a -> bool) -> 'a list ->
    bool

Now sort and inOrder need to use the same definition of le. But how can we enforce this?

fun split [] = ([],[])
|    split [a] = ([a],[])
|    split (a::b::rest) =
        let val (left,right) = split rest in
            (a::left,b::right)
        end;
val split = fn :
    'a list * 'a list -> 'a list

fun merge([],[]) = []
|    merge([],hd::tl) = hd::tl
|    merge(hd::tl,[]) = hd::tl
|    merge(hd::tl,h::t) =
        if le(hd,h)
            then hd::merge(tl,h::t)
            else h::merge(hd::tl,t)

fun merge([],[a]) = [a]
|    merge([a],[]) = [a]
|    merge([a],hd::tl) = a::merge([a],tl)

fun merge ([a],[b]) =
    if le(a,b)
        then a::merge([b],tl)
        else b::merge([a],tl)

fun merge ([a],[b],[c]) =
    if le(a,b)
        then a::merge([b],[c])
        else b::merge([a],[c])

fun merge ([a],[b],[c],[d]) =
    if le(a,b)
        then a::merge([b],[c],[d])
        else b::merge([a],[c],[d])
fun sort [] = []
| sort([a]) = [a]
| sort(a::b::rest) = 
  let val (left, right) = 
    split(a::b::rest) in 
    merge(sort(left), sort(right)) 
  end;

fun inOrder [] = true
| inOrder [a] = true 
| inOrder (a::b::rest) = 
  le(a,b) andalso 
  inOrder (b::rest);
end;
structure Sorting : 
sig
  val inOrder : int list -> bool
  val le : int * int -> bool
  val merge : int list * 
    int list -> int list
  val sort :
    int list -> int list
end

val split : 'a list -> 
  'a list * 'a list
end

To sort a type other than integers, we replace the definition of le in the structure.
But rather than actually edit that definition, ML gives us a powerful mechanism to parameterize a structure. This is the functor, which allows us to use one or more structures as parameters in the definition of a structure.

Functors

The general form of a functor is
functor name
  (structName:signature) = 
structure definition;

This functor will create a specific version of the structure definition using the structure parameter passed to it.

For our purposes this is ideal—we pass in a structure defining an ordering relation (the le function). This then creates a custom version of all the functions defined in the structure body, using the specific le definition provided.

We first define
signature Order = 
sig
  type elem
  val le : elem*elem -> bool
end;

This defines the type of a structure that defines a le predicate defined on a pair of types called elem.

An example of such a structure is
structure IntOrder:Order = 
struct
  type elem = int;
  fun le(a,b) = a <= b;
end;

Now we just define a functor that creates a Sorting structure based on an Order structure:
functor MakeSorting(O:Order) =
struct
  open O; (* makes le available*)
  fun split [] = ([],[])
      | split [a] = ([a],[])
      | split (a::b::rest) =
        let val (left,right) = split rest in
        (a::left,b::right)
          end;
  fun merge([],[]) = []
      | merge([],hd::tl) = hd::tl
      | merge(hd::tl,[]) = hd::tl
      | merge(hd::tl,h::t) =
        if le(hd,h)
          then hd::merge(tl,h::t)
          else h::merge(hd::tl,t)
        end;
  end;

fun sort [] = []
  | sort([a]) = [a]
  | sort(a::b::rest) =
    let val (left,right) = split(a::b::rest) in
    merge(sort(left),
         sort(right))
      end;

fun inOrder [] = true
  | inOrder [a] = true
  | inOrder (a::b::rest) =
    le(a,b) andalso
    inOrder (b::rest);
end;

structure IntSorting = MakeSorting(IntOrder);
creates a custom structure for sorting integers:
  IntSorting.sort [3,0,-22,8];
val it = [-22,0,3,8] : elem list
To sort strings, we just define a
structure containing an le defined
for strings with Order as its
signature (i.e., type) and pass it to
MakeSorting:
structure StrOrder:Order =
struct
  type elem = string
  fun le(a:string,b) = a <= b;
end;

structure StrSorting =
  MakeSorting(StrOrder);
StrSorting.sort ("cc","abc","xyz");
val it = ["abc","cc","xyz"] : StrOrder.elem list
StrSorting.inOrder(["cc","abc","xyz"]);
val it = false : bool
StrSorting.inOrder([3,0,-22,8]);
stdIn:593.1-593.32 Error: operator and operand don't agree
[literal]
  operator domain: strOrder.elem list
  operand domain: int list
  in expression:
  StrSorting.inOrder (3 :: 0 ::
    ["exp"] :: <exp> :: <exp>)
The SML Basis Library

SML provides a wide variety of useful types and functions, grouped into structures, that are included in the Basis Library.

A web page fully documenting the Basis Library is linked from the ML page that is part of the Programming Languages Links page on the CS 538 home page.

Many useful types, operators and functions are “preloaded” when you start the SML compiler. These are listed in the “Top-level Environment” section of the Basis Library documentation.

Many other useful definitions must be explicitly fetched from the structures they are defined in.

For example, the Math structure contains a number of useful mathematical values and operations. You may simply enter

open Math;

while will load all the definitions in Math. Doing this may load more definitions than you want. What’s worse, a definition loaded may redefine a definition you currently want to stay active. (Recall that ML has virtually no overloading, so functions with the same name in different structures are common.)

A more selective way to access a definition is to qualify it with the structure’s name. Hence

Math.pi;
val it = 3.14159265359 : real

gets the value of pi defined in Math.

Should you tire of repeatedly qualifying a name, you can (of course) define a local value to hold its value. Thus

val pi = Math.pi;
val pi = 3.14159265359 : real

works fine.

An Overview of Structures in the Basis Library

The Basis Library contains a wide variety of useful structures. Here is an overview of some of the most important ones.

- Option
  Operations for the option type.
- Bool
  Operations for the bool type.
- Char
  Operations for the char type.
- String
  Operations for the string type.
- Byte
  Operations for the byte type.
- Int
  Operations for the int type.
ML Type Inference

One of the most novel aspects of ML is the fact that it infers types for all user declarations.

How does this type inference mechanism work?

Essentially, the ML compiler creates an unknown type for each declaration the user makes. It then solves for these unknowns using known types and a set of type inference rules. That is, for a user-defined identifier \( i \), ML wants to determine \( T(i) \), the type of \( i \).

The type inference rules are:

1. The types of all predefined literals, constants and functions are known in advance. They may be looked-up and used. For example,

   ```
   2 : int
   true : bool
   [] : 'a list
   :: : 'a * 'a list -> 'a list
   ```

2. All occurrences of the same symbol (using scoping rules) have the same type.

3. In the expression

   \[ i = j \]

   we know \( T(i) = T(j) \).
4. In a conditional
   (if E1 then E2 else E3)
   we know that
   T(E1) = bool,
   T(E2) = T(E3) = T(conditional)

5. In a function call
   (f x)
   we know that if T(f) = 'a -> 'b
   then T(x) = 'a and T(f x) = 'b

6. In a function definition
   fun f x = expr;
   if t(x) = 'a and T(expr) = 'b
   then T(f) = 'a -> 'b

7. In a tuple (e_1, e_2, ..., e_n)
   if we know that T(e_i) = 'a_i \leq i \leq n
   then T(e_1, e_2, ..., e_n) =
     'a_1 * 'a_2 * ... * 'a_n

8. In a record
   { a=e_1, b=e_2, ... }
   if T(e_i) = 'a_i 1 \leq i \leq n then
     the type of the record =
     {a:'a_1, b:'a_2, ...}

9. In a list [v_1, v_2, ..., v_n]
   if we know that T(v_i) = 'a_i 1 \leq i \leq n
   then we know that
     'a_1='a_2=...='a_n and
     T([v_1, v_2, ..., v_n]) = 'a_1 list

To Solve for Types:
1. Assign each untyped symbol its own
   distinct type variable.
2. Use rules (1) to (9) to solve for and
   simplify unknown types.
3. Verify that each solution “works”
   (causes no type errors) throughout the
   program.

Examples
Consider
fun fact(n)=
  if n=1 then 1 else n*fact(n-1);
To begin, we'll assign type variables:
T(fact) = 'a -> 'b
(fact is a function)
T(n) = 'c

Now we begin to solve for the types
'a, 'b and 'c must represent.
We know (rule 5) that 'c = 'a since
n is the argument of fact.
We know (rule 3) that 'c = T(1) =
int since n=1 is part of the definition.
We know (rule 4) that T(1) = T(if
expression) = 'b since the if
expression is the body of fact.
Thus, we have
'a = 'b = 'c = int, so
T(fact) = int -> int
T(n) = int
These types are correct for all
occurrences of fact and n in the
definition.
A Polymorphic Function:

fun leng(L) = 
  if L = []
  then 0
  else 1+len(tl L);
To begin, we know that
T([]) = 'a list and
T(tl) = 'b list -> 'b list
We assign types to leng and L:
T(leng) = 'c -> 'd
T(L) = 'e
Since L is the argument of leng,
'e = 'c
From the expression L=[] we know
'e = 'a list

From the fact that 0 is the result of
the then, we know the if returns an
int, so 'd = int.
Thus T(leng) = 'a list -> int and
T(L) = 'a list
These solutions are type correct
throughout the definition.

Type Inference for Patterns
Type inference works for patterns too.
Consider
fun leng [] = 0 
  |  leng (a::b) = 1 + leng b;
We first create type variables:
T(leng) = 'a -> 'b
T(a) = 'c
T(b) = 'd
From leng [] we conclude that
'a = 'e list
From leng [] = 0 we conclude that
'b = int
From leng (a::b) we conclude that
'c = 'e and 'd = 'e list
Thus we have
T(leng) = 'e list -> int

T(a) = 'e
T(b) = 'e list
This solution is type correct
throughout the definition.
Not Everything can be Automatically Typed in ML

Let’s try to type

```
fun f x = (x x);
```

We assume

- \(T(f) = 'a \rightarrow 'b\)
- \(t(x) = 'c\)

Now (as usual) \('a = 'c\ since \(x\) is the argument of \(x\).

From the call \((x x)\) we conclude that \('c\ must be of the form \('d \rightarrow 'e\) (since \(x\) is being used as a function).

Moreover, \('c = 'd\ since \(x\) is an argument in \((x x)\).

Thus \('c = 'd \rightarrow 'e = 'c \rightarrow 'e\).

But \('c = 'c \rightarrow 'e\ has no solution, so in ML this definition is invalid. We

can’t pass a function to itself as an argument—the type system doesn’t allow it.

In Scheme this is allowed:

```
(define (f x) (x x))
```

but a call like

```
(f f)
```

certainly doesn’t do anything good!

Type Unions

Let’s try to type

```
fun f g = ((g 3), (g true));
```

Now the type of \(g\) is \('a \rightarrow 'b\ since \(g\)
is used as a function.

The call \((g 3)\) says \('a = \text{int} and the call \((g \text{true})\) says \('a = \text{boolean}.

Does this mean \(g\) is polymorphic?

That is, is the type of \(f\)

\(f : ('a\rightarrow 'b) \rightarrow 'b* 'b\)?

NO!

All functions have the type \('a \rightarrow 'b\) but not all functions can be passed to \(f\).

Consider \(\text{not} : \text{bool} \rightarrow \text{bool}.

The call \((\text{not} 3)\) is certainly illegal.

What we’d like in this case is a union type. That is, we’d like to be able to type \(g\) as \((\text{int} \mid \text{bool}) \rightarrow 'b\ which ML doesn’t allow.

Fortunately, ML does allow type constructors, which are just what we need.

Given

```
datatype T =
  I of \text{int} \mid B of \text{bool};
```

we can redefine \(f\) as

```
fun f g =
  (g (I(3)), g (B(\text{true})));

val f = fn : (T \rightarrow 'a) \rightarrow 'a * 'a
```
Finally, note that in a definition like

```
let
  val f = fn x => x (* id function*)
in (f 3, f true)
end;
```

type inference works fine:
```
val it = (3,true) : int * bool
```

Here we define \( f \) in advance, so its type is known when calls to it are seen.

---

**Prolog**

Prolog presents a view of programming that is very different from most other programming languages.

A famous text book is entitled “Algorithms + Data Structures = Programs”

This formula represents well the conventional approach to programming that most programming languages support.

In Prolog there is an alternative rule of programming:

“Algorithms = Logic + Control”

This rule encompasses a non-procedural view of programming.

---

**Reading Assignment**

- Roosta: Chapters 10 and 11
- Webber: Chapters 19, 20 and 22
Logic Programming

Prolog implements logic programming.
In fact Prolog means Prog(ramming) in Logic.
In Prolog programs are statements of rules and facts.
Program execution is deduction—can an answer be inferred from known rules and facts.
Prolog was developed in 1972 by Kowalski and Colmerauer at the University of Marseilles.

Elementary Data Objects

- In Prolog integers and atoms are the elementary data objects.
- Integers are ordinary integer literals and values.
- Atoms are identifiers that begin with a lower-case letter (much like symbolic values in Scheme).
- In Prolog data objects are called terms.
- In Prolog we define relations among terms (integers, atoms or other terms).
- A predicate names a relation. Predicates begin with lower-case letters.
- To define a predicate, we write clauses that define the relation.

- There are two kinds of program clauses, facts and rules.
- A fact is a predicate that prefixes a sequence of terms, and which ends with a period (".").
As an example, consider the following facts which define "fatherOf" and "motherOf" relations.

\[
\text{fatherOf}(\text{tom}, \text{dick}).
\text{fatherOf}(\text{dick}, \text{harry}).
\text{fatherOf}(\text{jane}, \text{harry}).
\text{motherOf}(\text{tom}, \text{judy}).
\text{motherOf}(\text{dick}, \text{mary}).
\text{motherOf}(\text{jane}, \text{mary}).
\]

The symbols fatherOf and motherOf are predicates. The symbols tom, dick, harry, judy, mary and jane are atoms.

Once we have entered rules and facts that define relations, we can make queries (ask the Prolog system questions).
Prolog has two interactive modes that you can switch between.
To enter definition mode (to define rules and facts) you enter

\[
\text{[user].}
\]
You then enter facts and rules, terminating this phase with \(^D\) (end of file).
Alternatively, you can enter

\[
\text{['filename'].}
\]
to read in rules and facts stored in the file named filename.
When you start Prolog, or after you leave definitions mode, you are in query mode.

In query mode you see a prompt of the form

| ?- or $?- (depending on the system you are running).

In query mode, Prolog allows you to ask whether a relation among terms is true or false.

Thus given our definition of motherOf and fatherOf relations, we can ask:

| ?- fatherOf(tom,dick).

yes

A “yes” response means that Prolog is able to conclude from the facts and rules it has been given that the relation queried does hold.

Variables in Queries

One of the attractive features of Prolog is the fact that variables may be included in queries. A variable always begins with a capital letter. When a variable is seen, Prolog tries to find a value (binding) for the variable that will make the queried relation true.

For example,

fatherOf(X, harry).

asks Prolog to find an value for X such that X’s father is harry.

When we enter the query, Prolog gives us a solution (if one can be found):

?– fatherOf(X, harry).

X = dick

If no solution can be found, it tells us so:

| ?– fatherOf(Y, jane).

no

Since solutions to queries need not be unique, Prolog will give us alternate solutions if we ask for them. We do so by entering a “;” after a solution is printed. We get a “no” when no more solutions can be found:

| ?– fatherOf(X, harry).

X = dick;

X = jane;

no
Variables may be placed anywhere in a query. Thus we may ask
| ?- fatherOf(jane,X).
  X = harry ;
  no
We may use more than one variable if we wish:
| ?- fatherOf(X,Y).
  X = tom,
  Y = dick ;
  X = dick,
  Y = harry ;
  X = jane,
  Y = harry ;
  no
(This query displays all the fatherOf relations).

Conjunction of Goals
More than one relation can be included as the “goal” of a query. A comma (“,”) is used as an AND operator to indicate a conjunction of goals—all must be satisfied by a solution to the query.
| ?- fatherOf(jane,X),motherOf(jane,Y).
  X = harry,
  Y = mary ;
  no
A given variable may appear more than once in a query. The same value of the variable must be used in all places in which the variable appears (this is called unification).

For example,
| ?- fatherOf(tom,X),fatherOf(X,harry).
  X = dick ;
  no

Rules in Prolog
Rules allow us to state that a relation will hold depending on the truth (correctness) of other relations. In effect a rules says, “If I know that certain relations hold, then I also know that this relation holds.”

A rule in Prolog is of the form
rel₁ :- rel₂, rel₃, ... relₙ.

This says rel₁ can be assumed true if we can establish that rel₂ and rel₃ and all relations to relₙ are true. rel₁ is called the head of the rule. rel₂ to relₙ form the body of the rule.
Example

The following two rules define a grandMotherOf relation using the motherOf and fatherOf relations:

\[
\text{grandMotherOf}(X, \text{GM}) :- \\
\text{motherOf}(X, M), \\
\text{fatherOf}(M, \text{GM}).
\]

\[
\text{grandMotherOf}(X, \text{GM}) :- \\
\text{fatherOf}(X, F), \\
\text{motherOf}(F, \text{GM}).
\]

| ?- grandMotherOf(tom, \text{GM}). |
| GM = mary ; no |
| ?- grandMotherOf(dick, \text{GM}). |
| no |
| ?- grandMotherOf(X, mary). |
| X = tom ; no |

As is the case for all programming, in all languages, you must be careful when you define a rule that it correctly captures the idea you have in mind.

Consider the following rule that defines a sibling relation between two people:

\[
\text{sibling}(X, Y) :- \\
\text{motherOf}(X, M), \text{motherOf}(Y, M), \\
\text{fatherOf}(X, F), \text{fatherOf}(Y, F).
\]

This rule says that \(x\) and \(y\) are siblings if each has the same mother and the same father.

But the rule is wrong!

Why?

Let’s give it a try:

| ?- sibling(X, Y). X = Y = tom |

Darn! That’s right, you can’t be your own sibling. So we refine the rule to force \(x\) and \(y\) to be distinct:

\[
\text{sibling}(X, Y) :- \\
\text{motherOf}(X, M), \text{motherOf}(Y, M), \\
\text{fatherOf}(X, F), \text{fatherOf}(Y, F), \\
\text{not}(X = Y).
\]

(A few Prolog systems use “\(+\)” for not; but most include a \text{not} relation.)

| ?- sibling(X, Y). X = dick, Y = jane ; X = jane, Y = dick ; no |

Note that distinct but equivalent solutions (like \(x = \text{dick}, y = \text{jane}\) vs. \(x = \text{jane}, y = \text{dick}\)) often appear in Prolog solutions. You may sometimes need to “filter out” solutions that are effectively redundant (perhaps by formulating stricter or more precise rules).
How Prolog Solves Queries

The unique feature of Prolog is that it automatically chooses the facts and rules needed to solve a query. But how does it make its choice? It starts by trying to solve each goal in a query, left to right (recall goals are connected using “,” which is the and operator). For each goal it tries to match a corresponding fact or the head of a corresponding rule.

A fact or head of rule matches a goal if:

- Both use the same predicate.
- Both have the same number of terms following the predicate.
- Each term in the goal and fact or rule head match (are equal), possibly binding a free variable to force a match.

For example, assume we wish to match the following goal:

\[ x(a, B) \]

This can match the fact

\[ x(a, b) \]

or the head of the rule

\[ x(Y, Z) : - Y = Z. \]

But \( x(a, B) \) can’t match

\[ y(a, b) \]  (wrong predicate name) or
\[ x(b, d) \]  (first terms don’t match) or
\[ x(a, b, c) \]  (wrong number of terms).

If we succeed in matching a rule, we have solved the goal in question; we can go on to match any remaining goals.

If we match the head of a rule, we aren’t done—we add the body of the rule to the list of goals that must be solved.

Thus if we match the goal \( x(a, B) \) with the rule

\[ x(Y, Z) : - Y = Z. \]

then we must solve \( a = B \) which is done by making \( B \) equal to \( a \).

Backtracking

If we reach a point where a goal can’t be matched, or the body of a rule can’t be matched, we backtrack to the last (most recent) spot where a choice of matching a particular fact or rule was made. We then try to match a different fact or rule. If this can’t be done, we go back to the next previous place where a choice was made and try a different match there. We try alternatives until we are able to solve all the goals in our query or until all possible choices have been tried and found to fail. If this happens, we answer “no” the query can’t be solved.

As we try to match facts and rules we try them in their order of definition.
Example

Let's trace how
\[ ?- \text{grandMotherOf}(\text{tom}, \text{GM}). \]
is solved.

Recall that
\[
\text{grandMotherOf}(X, \text{GM}) :\]
\[\quad \text{motherOf}(X, M),
\quad \text{motherOf}(M, \text{GM}).
\]

\[
\text{grandMotherOf}(X, \text{GM}) :\]
\[\quad \text{fatherOf}(X, F),
\quad \text{motherOf}(F, \text{GM}).
\]

\[
\text{fatherOf}(\text{tom}, \text{dick}).
\]
\[
\text{fatherOf}(\text{dick}, \text{harry}).
\]
\[
\text{fatherOf}(\text{jane}, \text{harry}).
\]
\[
\text{motherOf}(\text{tom}, \text{judy}).
\]
\[
\text{motherOf}(\text{dick}, \text{mary}).
\]
\[
\text{motherOf}(\text{jane}, \text{mary}).
\]

We try the first \text{grandMotherOf} rule first.
This forces \( x = \text{tom} \). We have to solve
\[
\text{motherOf}(\text{tom}, M),
\text{motherOf}(M, \text{GM}).
\]

We now try to solve
\[
\text{motherOf}(\text{tom}, M)
\]
This forces \( M = \text{judy} \).
We then try to solve
\[
\text{motherOf}(\text{judy}, \text{GM})
\]
None of the \text{motherOf} rules match this goal, so we backtrack. No other \text{motherOf} rule can solve
\[
\text{motherOf}(\text{tom}, M)
\]
so we backtrack again and try the second \text{grandMotherOf} rule:

\[
\text{grandMotherOf}(X, \text{GM}) :\]
\[\quad \text{fatherOf}(X, F),
\quad \text{motherOf}(F, \text{GM}).
\]

This matches, forcing \( x = \text{tom} \).
We have to solve
\[
\text{fatherOf}(\text{tom}, F), \text{motherOf}(F, \text{GM}).
\]
We can match the first goal with
\[
\text{fatherOf}(\text{tom}, \text{dick}).
\]
This forces \( F = \text{dick} \).
We then must solve
\[
\text{motherOf}(\text{dick}, \text{GM})
\]
which can be matched by
\[
\text{motherOf}(\text{dick}, \text{mary}).
\]
We have matched all our goals, so we know the query is true, with
\( \text{GM} = \text{mary} \).

List Processing in Prolog

Prolog has a notation similar to “cons cells” of Lisp and Scheme.
The \( . \) functor (predicate name) acts like \textbf{cons}.
Hence \( (a, b) \) in Prolog is essentially the same as \( (a \ . \ b) \) in Scheme.
Lists in Prolog are formed much the same way as in Scheme and ML:
\[
[\ ] \text{is the empty list}
\]
\[
[1, 2, 3] \text{is an abbreviation for}
\quad (1, \ . \ (2, \ . \ (3, [\ ])))
\]
just as
\[
(1, 2, 3) \text{ in Scheme is an abbreviation for}
\quad (\text{cons } 1 \ (\text{cons } 2 \ (\text{cons } 3 \ () \ )))
\]
The notation \([H|T]\) represents a list with \(H\) matching the head of the list and \(T\) matching the rest of the list.

Thus \([1,2,3]\) \(\equiv\) \([1| [2,3]]\) \(\equiv\) \([1,2| [3]]\) \(\equiv\) \([1,2,3| []]\)

As in ML, "\(\_\)" (underscore) can be used as a wildcard or "don’t care" symbol in matches.

Given the fact

\[ p([1,2,3,4]) . \]

The query

\[ \text{\texttt{| ?- p([X|Y]). \texttt{answers}} \]
\[ X = 1 , \]
\[ Y = [2,3,4] \]

---

**List Operations in Prolog**

List operations are defined using rules and facts. The definitions are similar to those used in Scheme or ML, but they are non-procedural.

That is, you don’t give a procedure order. Instead, you give recursive rules and non-recursive “base cases” that characterize the operation you are defining.

Consider append:

\[ \text{append([],L,L).} \]
\[ \text{append([H|T1],L2,[H|T3]) :- append(T1,L2,T3).} \]

The first fact says that an empty list (argument 1) appended to any list \(L\) (argument 2) gives \(L\) (argument 3) as its answer.

The query

\[ \text{\texttt{| ?- append([1],[2,3],[1,2,3]). \texttt{answers}} \]
\[ \text{\texttt{Yes}} \]

because with \(H=1\), \(T1=[]\), \(L2=[2,3]\) and \(T3=[2,3]\) it must be the case that \(\text{append([], [2,3], [2,3])}\) is true and fact (1) says that this is so.
Inverting Inputs and Outputs

In Prolog the division between “inputs” and “outputs” is intentionally vague. We can exploit this. It is often possible to “invert” a query and ask what inputs would compute a given output. Few other languages allow this level of flexibility.

Consider the query

append([1], X, [1, 2, 3]).

This asks Prolog to find a list X such that if we append [1] to X we will get [1, 2, 3].

Prolog answers

X = [2, 3]

How does it choose this answer?

First Prolog tries to match the query against fact (1) or rule (2).

Fact (1) doesn’t match (the first arguments differ) so we match rule (2).

This gives us H=1, T1= [], L2=X and T3 = [2, 3].

We next have to solve the body of rule (2) which is

append([1], L2, [2, 3]).

Fact (1) matches this, and tells us that L2= [2, 3]=X, and that’s our answer!

The Member Relation

A common predicate when manipulating lists is a membership test—is a given value a member of a list?

An “obvious” definition is a recursive one similar to what we might program in Scheme or ML:

member(X, [X|_]).

member(X, [_|Y]) :- member(X, Y).

This definition states that the first argument, X, is a member of the second argument (a list) if X matches the head of the list or if X is (recursively) a member of the rest of the list.

Note that we don’t have to “tell” Prolog that X can’t be a member of an empty list—if we don’t tell Prolog that something is true, it automatically assumes that it must be false.

Thus saying nothing about membership in an empty list is the same as saying that membership in an empty list is impossible.

Since inputs and outputs in a relation are blurred, we can use member in an unexpected way—to iterate through a list of values.
If we want to know if any member of a list \( L \) satisfies a predicate \( p \), we can simply write:

\[
\text{member}(X, L), p(X).
\]

There is no explicit iteration or searching. We simply ask Prolog to find an \( x \) such that \( \text{member}(X, L) \) is true (\( x \) is in \( L \)) and \( p(x) \) is true. Backtracking will find the “right” value for \( x \) (if any such \( x \) exists). This is sometimes called the “guess and verify” technique.

Thus we can query

\[
\text{member}(X, [3, -3, 0, 10, -10]),
\quad (X > 0).
\]

This asks for an \( x \) in the list \([3, -3, 0, 10, -10]\) which is greater than 0.

Prolog answers

\[
x = 3 ;
x = 10 ;
\]

Note too that our “obvious” definition of member is not the only one possible. An alternative definition (which is far less obvious) is

\[
\text{member}(X, L) :-
\quad \text{append}(_{,} [X|_{}, L].
\]

This definition says \( x \) is a member of \( L \) if I can take some list (whose value I don’t care about) and append it to a list that begins with \( x \) (and which ends with values I don’t care about) and get a list equal to \( L \).

Said more clearly, \( x \) is a member of \( L \) if \( x \) is anywhere in the “middle” of \( L \).

Prolog solves a query involving \( \text{member} \) by partitioning the list \( L \) in all possible ways, and checking to see if \( x \) ever is the head of the second list. Thus for \( \text{member}(X, [1, 2, 3]) \), it tries the partition \([\] and \([1, 2, 3] \) (exposing 1 as a possible \( x \)), then \([1] \) and \([2, 3] \) (exposing 2) and finally \([1, 2] \) and \([3] \) (exposing 3).

Sorting Algorithms

Sorting algorithms are good examples of Prolog's definitional capabilities. In a Prolog definition the “logic” of a sorting algorithm is apparent, stripped of the cumbersome details of data structures and control structures that dominate algorithms in other programming languages.

Consider the simplest possible sort imaginable, which we'll call the “naive sort.”

At the simplest level a sorting of a list \( L \) requires just two things:

- The sorting is a permutation (a reordering) of the values in \( L \).
- The values are “in order” (ascending or descending).
We can implement this concept of a sort directly in Prolog. We (a) permute an input list (b) check if it is in sorted order (c) repeat (a) & (b) until a sorting is found.

Permutations

Let's first look at how permutations are defined in Prolog. In most languages generating permutations is non-trivial—you need data structures to store the permutations you are generating and control structures to visit all permutations in some order.

In Prolog, permutations are defined quite concisely, though with a bit of subtlety:

\[ \text{perm}(X, Y) \] will be true if list \( Y \) is a permutation of list \( X \).

Only two definitions are needed:

\[ \text{perm}([], []) \].
\[ \text{perm}(L, [H|T]) :- \]
\[ \text{append}(V, [H|U], L), \]
\[ \text{append}(V, U, W), \text{perm}(W, T). \]

Permutations

Let's first look at how permutations are defined in Prolog. In most languages generating permutations is non-trivial—you need data structures to store the permutations you are generating and control structures to visit all permutations in some order.

In Prolog, permutations are defined quite concisely, though with a bit of subtlety:

\[ \text{perm}(X, Y) \] will be true if list \( Y \) is a permutation of list \( X \).

Only two definitions are needed:

\[ \text{perm}([], []) \].
\[ \text{perm}(L, [H|T]) :- \]
\[ \text{append}(V, [H|U], L), \]
\[ \text{append}(V, U, W), \text{perm}(W, T). \]

The first definition,
\[ \text{perm}([], []). \]
is trivial. An empty list may only be permuted into another empty list.

The second definition is rather more complex:
\[ \text{perm}(L, [H|T]) :- \]
\[ \text{append}(V, [H|U], L), \]
\[ \text{append}(V, U, W), \text{perm}(W, T). \]

This rule says a list \( L \) may be permuted in to a list that begins with \( H \) and ends with list \( T \) if:

1. \( L \) may be partitioned into two lists, \( V \) and \([H|U]\). (That is, \( H \) is somewhere in the “middle” of \( L \)).
2. Lists \( V \) and \( U \) (all of \( L \) except \( H \)) may be appended into list \( W \).
3. List \( W \) may be permuted into \( T \).

Let's see \text{perm} in action:

\[ \text{perm}([1, 2, 3], X). \]
\[ X = [1, 2, 3]; \]
\[ X = [1, 3, 2]; \]
\[ X = [2, 1, 3]; \]
\[ X = [2, 3, 1]; \]
\[ X = [3, 1, 2]; \]
\[ X = [3, 2, 1]; \]
\[ \text{no} \]

We'll trace how the first few answers are computed. Note though that all permutations are generated, and with no apparent data structures or control structures.

We start with \( L=[1, 2, 3] \) and \( X=[H|T] \).

We first solve \[ \text{append}(V, [H|U], L) \], which simplifies to \[ \text{append}(V, [H|U], [1, 2, 3]). \]
One solution to this goal is
\[ V = [], H = 1, U = [2,3] \]
We next solve \( \text{append}(V, U, W) \) which simplifies to \( \text{append}([1], [2,3], W) \).
The only solution for this is \( W = [2,3] \).
Finally, we solve \( \text{perm}(W, T) \), which simplifies to \( \text{perm}([2,3], T) \).
One solution to this is \( T = [2,3] \).
This gives us our first solution:
\[ [H|T] = [1,2,3] \]
To get our next solution we backtrack. Where is the most recent place we made a choice of how to solve a goal?
It was at \( \text{perm}([2,3], T) \). We chose \( T = [2,3] \), but \( T = [3,2] \) is another solution. Using this solution, we get out next answer \( [H|T] = [1,3,2] \).

Let's try one more. We backtrack again. No more solutions are possible for \( \text{perm}([2,3], T) \), so we backtrack to an earlier choice point.
At \( \text{append}(V, [H|U], [1,2,3]) \) another solution is
\[ V = [1], H = 2, U = [3] \]
Using this binding, we solve \( \text{append}(V, U, W) \) which simplifies to \( \text{append}([1], [3], W) \). The solution to this must be \( W = [1,3] \).
We then solve \( \text{perm}(W, T) \) which simplifies to \( \text{perm}([1,3], T) \). One solution to this is \( T = [1,3] \). This makes our third solution for \( [H|T] = [2,1,3] \).
You can check out the other bindings that lead to the last three solutions.

---

### A Permutation Sort

Now that we know how to generate permutations, the definition of a permutation sort is almost trivial.
We define an \textit{inOrder} relation that characterizes our notion of when a list is properly sorted:

\[ \text{inOrder}([]) \]  
\[ \text{inOrder}([ ]) \]  
\[ \text{inOrder}([A,B|T]) :-  
    A =< B, \text{inOrder}([B|T]) \]  

These definitions state that a null list, and a list with only one element are always in sorted order. Longer lists are in order if the first two elements are in proper order. \((A=<B)\) checks this and then the rest of the list, excluding the first element, is checked.

Now our naive permutation sort is only one line long:

\[
\text{naiveSort}(L1,L2) :-  
\text{perm}(L1,L2), \text{inOrder}(L2). 
\]

And the definition works too!

\[
|\ ?- \text{naiveSort}([1,2,3],[3,2,1]).  
no  
| ?- \text{naiveSort}([3,2,1],L).  
L = [1,2,3] ;  
no  
| ?- \text{naiveSort}([7,3,88,2,1,6,77, -23,5],L).  
L = [-23,1,2,3,5,6,7,77,88] 
\]
Though this sort works, it is hopelessly inefficient—it repeatedly "shuffles" the input until it happens to find an ordering that is sorted. The process is largely undirected. We don’t “aim” toward a correct ordering, but just search until we get lucky.

A Bubble Sort

Perhaps the best known sorting technique is the interchange or “bubble” sort. The idea is simple. We examine a list of values, looking for a pair of adjacent values that are “out of order.” If we find such a pair, we swap the two values (placing them in correct order). Otherwise, the whole list must be in sorted order and we are done.

In conventional languages we need a lot of code to search for out-of-order pairs, and to systematically reorder them. In Prolog, the whole sort may be defined in a few lines:

```
bubbleSort(L,L) :- inOrder(L).
bubbleSort(L1,L2) :-
    append(X, [A, B | Y], L1), A > B, 
    append(X, [B, A | Y], T),
    bubbleSort(T, L2).
```

The first line says that if \( L \) is already in sorted order, we are done.

The second line is a bit more complex. It defines what it means for a list \( L2 \) to be a sorting for list \( L1 \), using our insight that we should swap out-of-order neighbors. We first partition list \( L1 \) into two lists, \( X \) and \( [A, B | Y] \). This “exposes” two adjacent values in \( L, A \) and \( B \). Next we verify that \( A \) and \( B \) are out-of-order \((A>B)\). Next, in

```
append(X, [B, A | Y], T),
```

we determine that list \( T \) is just our input \( L \), with \( A \) and \( B \) swapped into \( B \) followed by \( A \).

Finally, we verify that \( \text{bubbleSort}(T, L2) \) holds. That is, \( T \) may be bubble-sorted into \( L2 \).

This approach is rather more directed than our permutation sort—we look for an out-of-order pair of values, swap them, and then sort the “improved” list. Eventually there will be no more out-of-order pairs, the list will be in sorted order, and we will be done.
**Merge Sort**

Another popular sort in the “merge sort” that we have already seen in Scheme and ML. The idea here is to first split a list of length $L$ into two sublists of length $L/2$. Each of these two lists is recursively sorted. Finally, the two sorted sublists are merged together to form a complete sorted list.

The bubble sort can take time proportional to $n^2$ to sort $n$ elements (as many as $n^2/2$ swaps may be needed). The merge sort does better—it takes time proportional to $n \log_2 n$ to sort $n$ elements (a list of size $n$ can only be split in half $\log_2 n$ times).

We first need Prolog rules on how to split a list into two equal halves:

- `split([], [], []).`
- `split([A], [A], []).`
- `split([A,B|T], [A|P1], [B|P2]) :- split(T, P1, P2).`

The first two lines characterize trivial splits. The third rule distributes one of the first two elements to each of the two sublists, and then recursively splits the rest of the list.

We also need rules that characterize how to merge two sorted sublists into a complete sorted list:

- `merge([], L, L).`
- `merge(L, [], L).`
- `merge([A|T1], [B|T2], [A|L2]) :- A =< B, merge(T1, [B|T2], L2).`
- `merge([A|T1], [B|T2], [B|L2]) :- A > B, merge([A|T1], T2, L2).`

The first 2 lines handle merging null lists. The third line handles the case where the head of the first sublist is $\leq$ the head of the second sublist; the final rule handles the case where the head of the second sublist is smaller.

With the above definitions, a merge sort requires only three lines:

- `mergeSort([], []).`
- `mergeSort([A], [A]).`
- `mergeSort(L1, L2) :- split(L1, P1, P2), mergeSort(P1, S1), mergeSort(P2, S2), merge(S1, S2, L2).`

The first two lines handle the trivial cases of lists of length 0 or 1. The last line contains the full “logic” of a merge sort: split the input list $L$ into two half-sized lists $P1$ and $P2$. Then merge sort $P1$ into $S1$ and $P2$ into $S2$. Finally, merge $S1$ and $S2$ into a sorted list $L2$. That’s it!
Quick Sort

The merge sort partitions its input list rather blindly, alternating values between the two lists. What if we partitioned the input list based on values rather than positions?

The quicksort does this. It selects a “pivot” value (the head of the input list) and divides the input into two sublists based on whether the values in the list are less than the pivot or greater than or equal to the pivot. Next the two sublists are recursively sorted. But now, after sorting, no merge phase is needed. Rather, the two sorted sublists can simply be appended, since we know all values in the first list are less than all values in the second list.

We need a Prolog relation that characterizes how we will do our partitioning. We define partition(E, L1, L2, L3) to be true if L1 can be partitioned into L2 and L3 using E as the pivot element. The necessary rules are:

\[
\text{partition}(E, [], [], []).\\
\text{partition}(E, [A|T1], [A|T2], L3) :- A<E, \text{partition}(E, T1, T2, L3).\\
\text{partition}(E, [A|T1], L2, [A|T3]) :- A>=E, \text{partition}(E, T1, L2, T3)
\]

The first line defines a trivial partition of a null list. The second line handles the case in which the first element of the list to be partitioned is less than the pivot, while the final line handles the case in which the list head is greater than or equal to the pivot.

With our notion of partitioning defined, the quicksort itself requires only 2 lines:

\[
\text{qsort}([], []).\\
\text{qsort}([A|T], L) :- \text{partition}(A, T, L1, L2), \text{qsort}(L1, S1), \text{qsort}(L2, S2), \text{append}(S1, [A|S2], L).
\]

The first line defines a trivial sort of an empty list. The second line says to sort a list that begins with \( A \) and ends with list \( T \), we partition \( T \) into sublists \( L1 \) and \( L2 \), based on \( A \). Then we recursively quick sort \( L1 \) into \( S1 \) and \( L2 \) into \( S2 \). Finally we append \( S1 \) to \( [A|S2] \) (\( A \) must be > all values in \( S1 \) and \( A \) must be ≤ all values in \( S2 \)). The result is \( L \), a sorting of \( [A|T] \).

Arithmetic in Prolog

The = predicate can be used to test bound variables for equality (actually, identity).

If one or both of =’s arguments are free variables, = forces a binding or an equality constraint.

Thus

\[
| ?- 1=2. \\
\text{no}\\
| ?- X=2. \\
X = 2\\
| ?- Y=X. \\
Y = X = _10751\\
| ?- X=Y, X=joe. \\
X = Y = joe
\]
Arithmetic Terms are Symbolic

Evaluation of an arithmetic term into a numeric value must be forced.
That is, \(1+2\) is an infix representation of the relation \(+ (1, 2)\). This term is not an integer!
Therefore
\[
?- 1+2=3. \\
no
\]
To force arithmetic evaluation, we use the infix predicate \(\text{is}\).
The right-hand side of \(\text{is}\) must be all ground terms (literals or variables that are already bound). No free (unbound) variables are allowed.

\[
\text{Hence}
\]
\[
?- 2 \text{ is } 1+1. \\
yes
\]
\[
?- x \text{ is } 3*4. \\
x = 12
\]
\[
?- y \text{ is } z+1. \\
! Instantiation error in argument 2 of \text{is}/2 \\
! goal: \_10712 \text{ is } \_10715+1
\]
The requirement that the right-hand side of an \(\text{is}\) relation be ground is essentially procedural. It exists to avoid having to invert complex equations. Consider,
\[
(0 \text{ is } (I**N)+(J**N)-K**N)), N>2.
\]

Counting in Prolog

Rules that involve counting often use the \(\text{is}\) predicate to evaluate a numeric value.
Consider the relation \(\text{len}(L,N)\) that is true if the length of list \(L\) is \(N\).
\[
\text{len([],0).} \\
\text{len([_|T],N) :- len(T,M), N is M+1.} \\
?- \text{len([1,2,3],X).} \\
X = 3
\]
The symbols \_10903 and \_10905 are “internal variables” created as needed when a particular value is not forced in a solution.

Debugging Prolog

Care is required in developing and testing Prolog programs because the language is untyped; undeclared predicates or relations are simply treated as false.
Thus in a definition like
\[
\text{adj([A,B|_]) :- A=B.} \\
\text{adj([_,B|T]) :- adk([B|T]).} \\
?- \text{adj([1,2,2]).} \\
no
\]
(Some Prolog systems warn when an undefined relation is referenced, but many others don’t).
Similarly, given

\[
\text{member}(A, [A|_]).
\]
\[
\text{member}(A, [_|T]) :- \text{member}(A, [T]).
\]

| ?- \text{member}(2, [1, 2]).

Infinite recursion! (Why?)

If you're not sure what is going on, Prolog's trace feature is very handy.

The command

\[
\text{trace}.
\]

turns on tracing. (notrace turns tracing off).

Hence

| ?- \text{trace}.

\[
\text{yes}\n\]

\[
\text{[trace]}\n\]

| ?- \text{member}(2, [1, 2]).

---

Termination Issues in Prolog

Searching infinite domains (like integers) can lead to non-termination, with Prolog trying every value.

Consider

\[
\text{odd}(1).
\]
\[
\text{odd}(N) :- \text{odd}(M), N \text{ is } M+2.
\]

| ?- \text{odd}(X).

\[
X = 1 ;
\]
\[
X = 3 ;
\]
\[
X = 5 ;
\]
\[
X = 7
\]

---

A query

| ?- \text{odd}(X), X=2.

going into an infinite search, generating each and every odd integer and finding none is equal to 2!

The obvious alternative,

\[
\text{odd}(2) \text{ (which is equivalent to } X=2, \text{ odd}(X)) \text{ also does an infinite, but fruitless search.}
\]

We'll soon learn that Prolog does have a mechanism to “cut off” fruitless searches.
Definition Order can Matter

Ideally, the order of definition of facts and rules should not matter. But, in practice definition order can matter. A good general guideline is to define facts before rules. To see why, consider a very complete database of motherOf relations that goes back as far as motherOf(cain,eve).

Now we define

\[
\text{isMortal}(X) :- \\
\text{isMortal}(Y), \text{motherOf}(X,Y).
\]

\[
\text{isMortal}(eve).
\]

These definitions state that the first woman was mortal, and all individuals descended from her are also mortal. But when we try as trivial a query as

\[
| \text{?- isMortal(eve).} \]

we go into an infinite search! Why?

Let’s trace what Prolog does when it sees

\[
| \text{?- isMortal(eve).} \]

It matches with the first definition involving isMortal, which is

\[
\text{isMortal}(X) :- \\
\text{isMortal}(Y), \text{motherOf}(X,Y).
\]

It sets \(X=\text{eve}\) and tries to solve

\[
\text{isMortal}(Y), \text{motherOf}(\text{eve},Y).
\]

It will then expand \(\text{isMortal}(Y)\) into

\[
\text{isMortal}(Z), \text{motherOf}(Y,Z).
\]

An infinite expansion ensues. The solution is simple—place the “base case” fact that terminates recursion first.

If we use

\[
\text{isMortal}(eve).
\]

\[
\text{isMortal}(X) :- \\
\text{isMortal}(Y), \text{motherOf}(X,Y).
\]

\[
\text{yes} \\
| \text{?- isMortal(eve).} \]

\[
\text{yes}
\]

But now another problem appears!

If we ask

\[
| \text{?- isMortal(clarkKent).} \]

we go into another infinite search! Why?

The problem is that Clark Kent is from the planet Krypton, and hence won’t appear in our motherOf database. Let’s trace the query.

It doesn’t match \(\text{isMortal(eve)}\).

We next try

\[
\text{isMortal(clarkKent) :- \\
\text{isMortal}(Y), \text{motherOf}(\text{clarkKent},Y).}
\]

We try \(Y=\text{eve}\), but \(\text{eve}\) isn’t Clark’s mother. So we recurse, getting:

\[
\text{isMortal}(Z), \text{motherOf}(Y,Z), \text{motherOf}(\text{clarkKent},Y).
\]

But \(\text{eve}\) isn’t Clark’s grandmother either! So we keep going further back, trying to find a chain of descendents that leads from \(\text{eve}\) to \(\text{clarkKent}\). No such chain exists, and there is no
limit to how long a chain Prolog will try.
There is a solution though!
We simply rewrite our recursive definition to be

\[
isMortal(X) :-
motherOf(X, Y), isMortal(Y).
\]
This is logically the same, but now we work from the individual \( x \) back toward \( eve \), rather than from \( eve \) toward \( x \). Since we have no \( motherOf \) relation involving \( clarkKent \), we immediately stop our search and answer \( no! \)

Extra-logical Aspects of Prolog
To make a Prolog program more efficient, or to represent negative information, Prolog needs features that have a procedural flavor. These constructs are called “extra-logical” because they go beyond Prolog’s core of logic-based inference.

The Cut
The most commonly used extra-logical feature of Prolog is the “cut symbol,” “!”. A “!” in a goal, fact or rule “cuts off” backtracking.

In particular, once a “!” is reached (and automatically matched), we may not backtrack across it. The rule we’ve selected and the bindings we’ve already selected are “locked in” or “frozen.”

For example, given

\[
x(A) :- y(A, B), z(B), !, v(B, C).
\]

once the “!” is hit we can’t backtrack to resatisfy \( y(A, B) \) or \( z(B) \) in some other way. We are locked into this

rule, with the bindings of \( A \) and \( B \) already in place.
We can backtrack to try various solutions to \( v(B, C) \).

It is sometimes useful to have several “!”’s in a rule. This allows us to find a partial solution, lock it in, find a further solution, then lock it in, etc.

For example, in a rule

\[
a(X) :- b(X), !, c(X, Y), !, d(Y).
\]

we first try to satisfy \( b(X) \), perhaps trying several facts or rules that define the \( b \) relation. Once we have a solution to \( b(X) \), we lock it in, along with the binding for \( x \).

Then we try to satisfy \( c(X, Y) \), using the fixed binding for \( x \), but perhaps trying several bindings for \( y \) until \( c(X, Y) \) is satisfied.
We then lock in this match using another !. Finally we check if \( d(Y) \) can be satisfied with the binding of \( Y \) already selected and locked in.

**When are Cuts Needed?**

A cut can be useful in improving efficiency, by forcing Prolog to avoid useless or redundant searches.

Consider a query like:

```prolog
member(X, list1),
member(X, list2), isPrime(X).
```

This asks Prolog to find an \( x \) that is in \( \text{list1} \) and also in \( \text{list2} \) and also is prime.

\( x \) will be bound, in sequence, to each value in \( \text{list1} \). We then check if \( x \) is also in \( \text{list2} \), and then check if \( x \) is prime.

Assume we find \( x=8 \) is in \( \text{list1} \) and \( \text{list2} \).isPrime(8) fails (of course). We backtrack to \( \text{member}(X, \text{list2}) \) and try to resatisfy it with the same value of \( x \).

**Expressing Negative Information**

Sometimes it is useful to state rules about what can't be true. This allows us to avoid long and fruitless searches.

`fail` is a goal that always fails. It can be used to represent goals or results that can never be true.

Assume we want to optimize our `grandMotherOf` rules by stating that a male can never be anyone's grandmother (and hence a complete search of all `motherOf` and `fatherOf` relations is useless).

A rule to do this is:

```prolog
grandMotherOf(X, GM) :-
    male(GM), fail.
```

But clearly there is never any point in trying to resatisfy \( \text{member}(X, \text{list2}) \). Once we know a value of \( x \) is in \( \text{list2} \), we test it using \( \text{isPrime}(X) \). If it fails, we want to go right back to \( \text{member}(X, \text{list1}) \) and get a different \( x \).

To create a version of `member` that never backtracks once it has been satisfied we can use !.

We define:

```prolog
member1(X, [X|_]) :- !.
member1(X, [_|Y]) :-
    member1(X,Y).
```

Our query is now:

```prolog
member1(X, list1),
member1(X, list2), isPrime(X).
```

(Why isn't \( \text{member1} \) used in both terms?)
This rule doesn't do quite what we hope it will!
Why?
The standard approach in Prolog is to try other rules if the current rule fails. Hence we need some way to “cut off” any further backtracking once this negative rule is found to be applicable.
This can be done using
\[
\text{grandMotherOf}(X, \text{GM}) :- \\
\text{male} (\text{GM}), !, \text{fail}.
\]

**Other Extra-Logical Operators**

- assert and retract
  These operators allow a Prolog program to add new rules during execution and (perhaps) later remove them. This allows programs to learn as they execute.

- findall
  Called as \text{findall}(X, \text{goal}, \text{List}) where \(X\) is a variable in \text{goal}. All possible solutions for \(X\) that satisfy \text{goal} are found and placed in \text{List}.
  For example,
  \[
  \text{findall}(X, (\text{append}_1, [X|_], [-1, 2, -3, 4]), (X<0)), L).
  \]
  \[
  L = [-1, -3]
  \]

- \text{var} and \text{nonvar}
  \text{var}(X) tests whether \(X\) is unbound (free).
  \text{nonvar}(Y) tests whether \(Y\) is bound (no longer free).
  These two operators are useful in tailoring rules to particular combinations of bound and unbound variables.
  For example, the rule
  \[
  \text{grandMotherOf}(X, \text{GM}) :- \\
  \text{male} (\text{GM}), !, \text{fail}.
  \]
  might backfire if \(\text{GM}\) is not yet bound. We could set \(\text{GM}\) to a person for whom \text{male} (\text{GM}) is true, then fail because we don't want grandmothers who are male!

To remedy this problem, we use the rule only when \(\text{GM}\) is bound. Our rule becomes
\[
\text{grandMotherOf}(X, \text{GM}) :- \\
\text{nonvar} (\text{GM}), \text{male} (\text{GM}), !, \text{fail}.
\]
An Example of Extra-Logical Programming

Factorial is a very common example program. It's well known, and easy to code in most languages.

In Prolog the “obvious” solution is:

\[
\text{fact}(N, 1) :- N =< 1.
\]

\[
\text{fact}(N, F) :- N > 1, M \text{ is } N-1, \text{ fact}(M, G), F \text{ is } N*G.
\]

This definition is certainly correct. It mimics the usual recursive solution.

But, in Prolog “inputs” and “outputs” are less distinct than in most languages. In fact, we can envision 4 different combinations of inputs and outputs, based on what is fixed (and thus an input) and what is free (and hence is to be computed):

1. \(N\) and \(F\) are both ground (fixed). We simply must decide if \(F=N!\)
2. \(N\) is ground and \(F\) is free. This is how \text{fact} is usually used. We must compute an \(F\) such that \(F=N!\)
3. \(F\) is fixed and \(N\) is free. This is an uncommon usage. We must find an \(N\) such that \(F=N!\), or determine that no such \(N\) is possible.
4. Both \(N\) and \(F\) are free. We generate, in sequence, pairs of \(N\) and \(F\) values such that \(F=N!\).

Our solution works for combinations 1 and 2 (where \(N\) is fixed), but not combinations 3 and 4. (The problem is that \(N =< 1\) and \(N > 1\) can’t be satisfied when \(N\) is free).

We'll need to use \text{nonvar} and \! to form a solution that works for all 4 combinations of inputs.

We first handle the case where \(N\) is ground:

\[
\text{fact}(1, 1).
\]

\[
\text{fact}(N, 1) :- \text{nonvar}(N), N =< 1, !.
\]

\[
\text{fact}(N, F) :- \text{nonvar}(N), N > 1, !, M \text{ is } N-1, \text{ fact}(M, G), F \text{ is } N*G, !.
\]

The first rule handles the base case of \(N=1\).

The second rule handles the case of \(N<1\).

The third rule handles the case of \(N >1\). The value of \(F\) is computed recursively. The first ! in each of these rules forces that rule to be the only one used for the values of \(N\) that match. Moreover, the second ! in the third rule states that after \(F\) is computed, further backtracking is useless; there is only one \(F\) value for any given \(N\) value.

To handle the case where \(F\) is bound and \(N\) is free, we use

\[
\text{fact}(N, F) :- \text{nonvar}(F), !, \text{ fact}(M, G), N \text{ is } M+1, F2 \text{ is } N*G, F =< F2, !, F=F2.
\]

In this rule we generate \(N, F2\) pairs until \(F2 \geq F\). Then we check if \(F=F2\). If this is so, we have the \(N\) we want. Otherwise, no such \(N\) can exist and we fail (and answer no).
For the case where both \( N \) and \( F \) are free we use:

\[
\text{fact}(N,F) :- \text{fact}(M,G), \ N \text{ is } M+1, \ \ F \text{ is } N*G.
\]

This systematically generates \( N, F \) pairs, starting with \( N=2, F=2 \) and then recursively building successor values (\( N=3, F=6 \), then \( N=4, F=24 \), etc.)

### Parallelism in Prolog

One reason that Prolog is of interest to computer scientists is that its search mechanism lends itself to parallel evaluation.

In fact, it supports two different kinds of parallelism:

- **AND Parallelism**
- **OR Parallelism**

### AND Parallelism

When we have a goal that contains subgoals connected by the "\(,\)" (And) operator, we may be able to utilize "and parallelism."

Rather than solve subgoals in sequence, we may be able to solve them in parallel if bindings can be properly propagated.

Thus in

\[
a(X), \ b(X,Y), \ c(X,Z), \ d(Y,Z).
\]

we may be able to first solve \( a(X) \), binding \( x \), then solve \( b(X,Y) \) and \( c(X,Z) \) in parallel, binding \( x \) and \( z \), then finally solve \( d(Y,Z) \).

### OR Parallelism

An example of this sort of and parallelism is

\[
\text{member}(X,\text{list1}), \ \ \text{member1}(X,\text{list2}), \ \text{isPrime}(X).
\]

Here we can let \( \text{member}(X,\text{list1}) \) select an \( x \) value, then test \( \text{member1}(X,\text{list2}) \) and \( \text{isPrime}(X) \) in parallel. If one or the other fails, we just select another \( x \) from \( \text{list1} \) and retest \( \text{member1}(X,\text{list2}) \) and \( \text{isPrime}(X) \) in parallel.
OR Parallelism

When we match a goal we almost always have a choice of several rules or facts that may be applicable. Rather than try them in sequence, we can try several matches of different facts or rules in parallel. This is "or parallelism."

Thus given

\[
\begin{align*}
a(X) :&= b(X). \\
a(Y) :&= c(Y).
\end{align*}
\]

when we try to solve

\[
a(10).
\]

we can simultaneously check both

\[
b(10) \quad \text{and} \quad c(10).
\]

Recall our definition of

\[
\text{member}(X, L) :- \text{append}(P, [X|S], L).
\]

where \text{append} is defined as

\[
\text{append}([], L, L).
\]

\[
\text{append}([X|L1], L2, [X|L3]) :- \text{append}(L1, L2, L3).
\]

Assume we have the query

\[
? \text{member}(2, [1,2,3]).
\]

This immediately simplifies to

\[
\text{append}(P, [2|S], [1,2,3]).
\]

Now there are two \text{append} definitions we can try in parallel:

(1) match \text{append}(P, [2|S], [1,2,3]) with \text{append}([], L, L). This requires that \[2|S] = [1,2,3], which must fail.

(2) match \text{append}(P, [2|S], [1,2,3]) with \text{append}([X|L1], L2, [X,L3]).

This requires that \(P=[[X|L1], [2|S]=L2, [1,2,3]=[X,L3].\)

Simplifying, we require that \(X=1, P=[1|L1], L3=[2,3].\)

Moreover we must solve \text{append}(L1, L2, L3) which simplifies to \text{append}([1|L1], [2|S], [2,3]).

We can match this call to \text{append} in two different ways, so or parallelism can be used again.

When we try matching \text{append}(L1, [2|S], [2,3]) against \text{append}([], L, L) we get \(2|S]=[2,3],\) which is satisfiable if \(S\) is bound to [3]. We therefore signal back that the query is true.

Speculative Parallelism

Prolog also lends itself nicely to speculative parallelism. In this form of parallelism, we “guess” or speculate that some computation may be needed in the future and start it early. This speculative computation can often be done in parallel with the main (non-speculative) computation.

Recall our example of

\[
\text{member}(X, list1), \quad \text{member1}(X, list2), \quad \text{isPrime}(X).
\]

After \text{member}(X, list1) has generated a preliminary solution for \(x,\) it is tested (perhaps in parallel) by \text{member1}(X, list2) and \text{isPrime}(X).

But this value of \(x\) may be rejected by one or both of these tests. If it is,
we'll ask `member(X, list1)` to find a new binding for `x`. If we wish, this next binding can be generated speculatively, while the current value of `x` is being tested. In this way if the current value of `x` is rejected, we'll have a new value ready to try (or know that no other binding of `x` is possible).

If the current value of `x` is accepted, the extra speculative work we did is ignored. It wasn’t needed, but was useful insurance in case further `x` bindings were needed.

**Reading Assignment**
- Java for C++ Programmers
  (linked from class web page)
- Roosta: Section 9.3
- Webber: Chapters 13, 15 and 17

---

**Java & Object-Oriented Programming**

Java is a fairly new and very popular programming language designed to support secure, platform-independent programming.

It is a good alternative to C or C++, trading a bit of efficiency for easier programming, debugging and maintenance.

Java is routinely interpreted (at the byte-code level), making it significantly slower than compiled C or C++. However true Java compilers exist, and are becoming more widespread. (IBM’s Jalapeno project is a good example). When compiled, Java’s execution speed is close to that of C or C++.

---

**Subclassing in Java**

When a new class is defined in terms of an existing class, the new class extends the existing class. The new class inherits all public and protected members of its parent (or base) class.

The new class may add new methods or fields. It may also redefine inherited methods or fields.

```java
class Point {
    int x, y;
    Point(int xin, int yin) {
        x = xin; y = yin;
    }
    static float dist(Point P1, Point P2) {
        return (float) Math.sqrt((P1.x-P2.x)*(P1.x-P2.x) + (P1.y-P2.y)*(P1.y-P2.y));
    }
}
```
class Point3 extends Point {
    int z;
    Point3(int xin, int yin, int zin) {
        super(xin, yin); z = zin;
    }
    static float dist(Point3 P1, Point3 P2) {
        float d = Point.dist(P1, P2);
        return (float) Math.sqrt((P1.z - P2.z) * (P1.z - P2.z) + d * d);
    }
}

Note that although Point3 redefines dist, the old definition of dist is still available by using the parent class as a qualifier (Point.dist).
The same is true for fields that are hidden when a field in a parent is redeclared.

Non-static methods are automatically virtual: a redefined method is automatically used in all inherited methods including those defined in parent classes that think they are using an earlier definition of the class.

Example:

```
class C {
    void DoIt() { PrintIt(); }
    void PrintIt() {
        println("C rules!");
    }
}
class D extends C {
    void PrintIt() {
        println("D rules!");
    }
    void TestIt() { DoIt();
    }
}  
d D dvar = new D();
dvar.TestIt();
D rules! is printed.
```

Static methods in Java are not virtual (this can make them easier to implement efficiently).

Abstract Classes and Methods

Sometimes a Java class is not meant to be used by itself because it is intentionally incomplete.
Rather, the class is meant to be starting point for the creation (via subclassing) of more complete classes.
Such classes are abstract.

Example:

```
abstract class Shape {
    Point location;
}
class Circle extends Shape {
    float radius;
}```
Methods can also be made abstract to indicate that their actual definition will appear in subclasses:

```java
abstract class Shape {
    Point location;
    abstract float area();
}
class Circle extends Shape {
    float radius;
    float area() {
        return Math.pi*radius*radius;
    }
}
```

Subtyping and Inheritance

We can use a subtyping mechanism, as found in C++ or Java, for two different purposes:

- We may wish to inherit the actual implementations of classes and members to use as the basis of a more complete or extended class. To inherit an implementation, we say a given class “extends” an existing class:
  ```java
class Derived extends Base {
    ...
};
```
  Class Derived contains all of the members of Base plus any others it cares to add.

- We may wish to inherit an interface—a set of method names and values that will be available for use. To inherit (or claim) an interface, we use a Java interface definition. An interface doesn't implement anything; rather, it gives a name to a set of operations or values that may be available within one or more classes.

Why are Interfaces Important?

Many classes, although very different, share a common subset of values or operations. We may be willing to use any such class as long as only interface values or operations are used.

For example, many objects can be ordered (or at least partially-ordered) using a “less than” operation.

If we always implement less than the same way, for example,

```java
boolean lessThan(Object o1, Object o2);
```

then we can create an interface that admits all classes that know about the lessThan function:
interface Compare {
    boolean lessThan(Object o1, Object o2);
}

Now different classes can each implement the Compare interface, proclaiming to the world that they know how to compare objects of the class they define:

class IntCompare implements Compare {
    public boolean lessThan(Object i1, Object i2) {
        return ((Integer)i1).intValue() < ((Integer)i2).intValue();
    }
}
class StringCompare implements Compare {
    public boolean lessThan(Object i1, Object i2) {
        return ((String)i1).compareTo((String)i2) < 0;
    }
}

The advantage of using interfaces is that we can now define a method or class that only depends on the given interface, and which will accept any type that implements that interface.

class PrintCompare {
    public static void printAns(Object v1, Object v2, Compare c) {
        System.out.println(v1.toString() + " < " + v2.toString() + " is " + new Boolean(c.lessThan(v1, v2)).toString());
    }
}
class Test {
    public static void main(String args[]) {
        Integer i1 = new Integer(2);
        Integer i2 = new Integer(1);
        PrintCompare.printAns(i1, i2, new IntCompare());
        String s2 = "abcdef";
        String s1 = "xyzaa";
        PrintCompare.printAns(s1, s2, new StringCompare());
    }
}

Since classes may have many methods and modes of use or operation, a given class may implement many different interfaces. For example, many classes support the Clonable interface, which states that objects of the class may be duplicated (cloned).

Multiple Inheritance

We have seen that a class may be derived from a given parent class. It is sometimes useful to allow a class to be derived from more than one parent, inheriting members of all parents. This is multiple inheritance; it is allowed by C++ and Python, but not by Java or C#.

The basic idea is that sometimes we want a “composite” object formed from more than one source. Hence a Computer object can be viewed as both a PhysicalObject (with height, weight, color, cost, etc.) and also a CPUImplementation (with memory size, processor design, processor speed, I/O ports, etc.)
Using multiple inheritance we merge aspects of a `PhysicalObject` and a `CPUImplementation`, and perhaps add additional data:

```java
class PhysicalObject {
    float height, width, weight;
    Color outsideColor;
    ...
}
class CPUImplementation {
    CPUClass CPUKind;
    int memorySize, CPUSpeed;
    ...
}
class Computer: PhysicalObject, CPUImplementation {
    String myURL; ...
}
```

The advantages of multiple inheritance are obvious—you can build a class from many sources rather than just one.

There are problems though:

- If the same name appears in more than one parent, which is used? For example, if both parents contain a “copyright” field, which do you get? C++ forbids access to fields common to several parents, though accidental clashes of member names are certainly possible. Python relies on order of specification of the parents (which can be somewhat arbitrary).

- Access to fields and methods can be less efficient. A method in a parent class can’t know how fields in derived class will be allocated when multiple parents may exist. Hence some form of indirection may be needed. For example, in class `PhysicalObject`, we may believe that `height` is the first field allocated, while in `CPUImplementation` we may believe that `CPUKind` is allocated first. But in class `Computer`, which contains both `height` and `CPUKind`, both fields can’t come first.

**Threads and Parallelism in Java**

Java is one of the few “main stream” programming languages to explicitly provide for user-programmed parallelism in the form of threads.

A Java programmer may organize a program as several threads that may execute concurrently.

Even if the program is run on a uni-processor, use of threads may improve performance. This is because the threads can be multi-programmed, with threads switched automatically on I/O delays, page faults or even cache misses.

A program that is designed to support multiple threads is also “prepared” for future upgrades to multiprocessors or multi-threaded processors.
Java Threads

In Java any class that implements the Runnable interface can be started as a concurrent thread:

```java
interface Runnable {
    public void run();
}
```

When the thread is started, the method run begins to execute (perhaps concurrently with other threads of the main program).

You create a thread using the Thread constructor:

```
new Thread(RunnableObject)
```

Creating a thread does not start it; you must execute the start method within the Thread object.

When this is done, the run method immediately starts, and continues until that method terminates normally, or it throws an uncaught exception, or it is explicitly stopped, or the main program stops.

On uniprocessors, thread execution is interleaved; on multiprocessors or multithreaded architectures execution can be concurrent.

```java
class DoSort implements Runnable {
    int [] data;
    DoSort(int[] in) {data=in;}
    public void run(){
        // sort the data array;
    }
}
```

We can start multiple threads, and the threads can use sleep to delay or synchronize their execution:

```java
class PingPong implements Runnable {
    int delay; String word;
    PingPong(String s,int i){
        delay=i;word=s;}
    public void run(){
        try {
            while(true){
                System.out.print(word+" ");
                Thread.sleep(delay);
            }
        } catch (InterruptedException e) {
            }
    }
```
public static void
    main(String args[]){
        Thread t1 = new Thread(
            new PingPong("ping",33));
        Thread t2 = new Thread(
            new PingPong("PONG",100));
        t1.start();
        t2.start();
    }
}

ping PONG ping ping PONG ping
ping ping PONG ping ping PONG ping ...

Synchronization in Java

We often want threads to co-operate, typically in how they access shared data structures.

Since thread execution is asynchronous, the details of how threads interact can be unpredictable.

Consider a method

```java
    void update() {
        n = n+1;
        val = f(n);
    }
```

that updates fields of an object.

If two or more threads execute `update` concurrently, we might get unexpected or even illegal behavior. (Why?)

A Java method may be synchronized, which guarantees that at most one thread can execute the method at a time. Other threads wishing access, are forced to wait until the currently executing thread completes.

Thus

```java
    void synchronized update() { ... }
```

can safely be used to update an object, even if multiple threads are active.

There is also a synchronized statement in Java that forces threads to execute a block of code sequentially.

```java
    synchronized(obj) {
        obj.n = obj.n+1;
        obj.val = f(obj.n);
    }
```

Synchronization Primitives

The following operations are provided to allow threads to safely interact:

- `wait()` Sleep until awakened
- `wait(n)` Sleep until awakened or until `n` milliseconds pass
- `notify()` Wake up one sleeping thread
- `notifyAll()` Wake up all sleeping threads

Using these primitives, correct concurrent access to a shared data structure can be programmed.
Consider a `Buffer` class in which independent threads may try to store or fetch data objects:

```java
class Buffer {
    private Queue q;
    Buffer() { q = new Queue(); }
    public synchronized void put(Object obj) {
        q.enqueue(obj);
        notify(); //Why is this needed?
    }
    public synchronized Object get() {
        while (q.isEmpty()) {
            //Why a while loop?
            wait();
        }
        return q.dequeue();
    }
}
```

**Locks, Semaphores and Monitors**

Java's synchronization mechanisms are based upon the notion of a lock. A lock is a special value that can be held by at most one thread.

If a thread holds a lock, it has permission to do some "critical" operation like writing a shared variable or restructuring a shared data object.

If a thread wants to do an operation but doesn't hold the necessary lock, it must wait until it gets the lock.

In Java there is a lock associated with each run-time object.

**Lock Granularity and Access**

Though each Java object has a lock, you often don't want to lock and unlock each object you access.

If you are manipulating a large data structure (like a tree or hash table), acquiring the lock for each object in the tree or table can be costly and error-prone.

Instead, it is common to create a lock corresponding to a group of objects. Hence holding the lock to the root of a tree may give you permission to access the whole tree.

There is a danger though—if all or most of a large structure is held by one thread, then other threads won't be able to access that structure concurrently.

For example, a large shared data base (like one used to record current bookings for an airline) shouldn't be held exclusively by one thread—hundreds of concurrent threads may want to access it at any time. An intermediate lock, like all reservations for a single fight, is more reasonable.

There is also an issue of how long you hold a lock. The ideal is to have exclusive access for as short a period as is necessary. Work that is not subject to interference by other threads (like computations using local variables) should be done before a lock is obtained. Hence Java's synchronized statement allows a method to get exclusive access to an object for a limited region, enhancing shared access.
Deadlock

A variety of programming problems appear in concurrent programs that don't exist in ordinary sequential programs.

The most serious of these is deadlock:

Two or more threads hold locks that other threads require. Each waits for the other thread to release a needed lock, and no thread is able to execute.

As an example of how deadlock may occur, consider two threads, \( t_1 \) and \( t_2 \). Each requires two files, a master file and a log file. Since these files are shared, each has a lock.

Assume \( t_1 \) gets the lock for the master file while \( t_2 \) (at the same instant) gets the lock for the log file.

Now each is stuck. Each has one file, and will wait forever for the other file to be released.

In Java deadlock avoidance is wholly up to the programmer. There are no language-level guarantees that deadlock can't happen.

Some languages have experimented with ways to help programmers avoid deadlock:

- If all locks must be claimed at once, deadlock can be avoided. You either get all of them or none, but you can't block other threads while making no progress yourself.
- Locks (and the resources they control) can be ordered, with the rule that you must acquire locks in the proper order. Now two threads can't each hold locks the other needs.
- The language can require that the largest set of locks ever needed be declared in advance. When locks are requested, the operating system can track what's claimed and what may be needed, and refuse to honor unsafe requests.

Fairness & Starvation

When one thread has a lock, other threads who want the lock will be suspended until the lock is released.

It can happen that a waiting thread may be forced to wait indefinitely to acquire a lock, due to an unfair waiting policy. A waiting thread that never gets a lock it needs due to unfair lock allocation faces starvation.

As an example, if we place waiting threads on a stack, newly arrived threads will get access to a lock before earlier arrivals. This can lead to starvation. Most thread managers try to be fair and guarantee that all waiting threads have a fair chance to acquire a lock.
How are Locks Implemented?

Internally, Java needs operations to acquire a lock and release a lock. These operations can be implemented using the notion of a semaphore.

A semaphore is an integer value (often just a single bit) with two atomic operations: up and down.

- `up(s)` increments `s` atomically.
- `down(s)` decrements `s` atomically. But if `s` is already zero, the process doing the `down` operation is put in a wait state until `s` becomes positive (eventually some other process should do an `up` operation).

Now locks are easy to implement.

You do a `down(lock)` to claim a lock. If someone else has it, you are forced to wait until the lock is released. If the lock value is > 0 you get it and all others are “locked out.”

When you want to release a lock, you do `up(lock)`, which makes `lock` non-zero and eligible for another thread to claim.

In fact, since only one thread will ever have a lock, the lock value needs to be only one bit, with 1 meaning currently free and unlocked and 0 meaning currently claimed and locked.

Monitors

Direct manipulation of semaphores is tedious and error-prone. If you acquire a lock but forget to release it, threads may be blocked forever.

Depending on where `down` and `up` operations are placed, it may be difficult to understand how synchronization is being performed.

Few modern languages allow direct use of semaphores. Instead, semaphores are used in the implementation of higher-level constructs like monitors.

A monitor is a language construct that guarantees that a block of code will be executed synchronously (one thread at a time).

The Java `synchronized` statement is a form of monitor.

When

```java
synchronized(obj) { ... }
```

is executed, “invisible” `getLock` and `freeLock` operations are added:

```java
synchronized(obj) {
    getLock(obj)
    ...
    freeLock(obj);
}
```

This allows the body of the `synchronized` statement to execute only when it has the lock for `obj`. Thus two different threads can never simultaneously execute the body of a `synchronized` statement because two threads can’t simultaneously hold `obj`’s lock.
In fact, synchronized methods are really just methods whose bodies are enclosed in an invisible `synchronized` statement.

If we execute

```java
obj.method()
```

where `method` is synchronized, `method`'s body is executed as if it were of the form

```java
synchronized(obj) {
    body of method
}
```

Operations like `sleep`, `wait`, `notify`, and `notifyAll` also implicitly cause threads to release locks, allowing other threads to proceed.

**Java 1.5 (Tiger Java)**

Recently, Java has been extended to include a variety of improvements, many drawn from functional languages.

Added features include:

- **Parametric polymorphism.**
  Classes and interfaces may be parameterized using a type parameter.
  ```java
class List<T> {
    T head;
    List<T> tail;
}
```
  Interfaces may also be parameterized.

- **Enhanced loop iterators.**
  ```java
  for (v : myArray) {
      // each element of myArray
      // appears as a value of v
  }
  ```

- **Automatic boxing and unboxing of wrapper classes.**
  Conversion from `int` to `Integer` or `Integer` to `int` is now automatic.

- **Typesafe enumerations.**
  ```java
  public enum Color {RED, BLUE, GREEN};
  ```

- **Static imports.**
  You may import all static members of a class and use them without qualification. Thus you may now write `out.println` rather than `System.out.println`.

- **Variable argument methods.**

- **Formatted output using `printf`:**
  ```java
  out.printf("Ans = %3d",a+b);
  ```

**Reading Assignment**

- **Python Tutorial** (linked from class web page)
- **Pizza Tutorial** (linked from class web page)
C# is Microsoft’s answer to Java. In most ways it is very similar to Java, with some C++ concepts reintroduced and some useful new features.

Similarities to Java include:
- C# is object-based, with all objects descended from class `Object`.
- Objects are created from classes using `new`. All objects are heap-allocated and garbage collection is provided.
- All code is placed within methods which must be defined within classes.
- Almost all Java reserved words have C# equivalents (many are identical).
- Classes have single inheritance.

C# Improves Upon Some Java Features
- Operators as well as methods can be overloaded:
  ```csharp
  class Point {
      int x, y;
      static Point operator + (Point p1, Point p2) {
          return new Point(p1.x+p2.x,
                          p1.y+p2.y);
      }
  }
  ``
- Switch statements may be indexed by string literals.
- In a switch, fall throughs to the next case are disallowed (if non-empty).
- `Goto`'s are allowed.
- Virtual methods must be marked.

- C# generates code for a virtual machine to support cross-platform execution.
- Interfaces are provided to capture functionality common to many classes.
- Exceptions are very similar in form to Java's.
- Instance and static data within an object must be initialized at point of creation.

- Persistent objects (that may be stored across executions) are available.
C# Adds Useful Features

- Events and delegates are included to handle asynchronous actions (like keyboard or mouse actions).
- Properties allow user-defined read and write actions for fields. You can add get and set methods to the definition of a field. For example,

```csharp
class Customer {
    private string name;
    public string Name {
        get { return name; }
        set { name = value; }
    }
}
```

Customer c; ...
string s = c.Name;

- Indexers allow objects other than arrays to be indexed. The [] operator is overloading. This allows you to define the meaning of `obj[123]` or `obj["abc"]` within any class definition.
- Collection classes may be directly enumerated:

```csharp
foreach (int i in array) ...
```

- Fields, methods and constructors may be defined within a struct as well as a class. Structs are allocated within the stack instead of the heap, and are passed by value. For example:

```csharp
struct Point {
    int x, y;
    void reset () {
        x = 0; y = 0;
    }
}
```

- When an object is needed, a primitive (int, char, etc.) or a struct will be automatically boxed or unboxed without explicit use of a wrapper class (like Integer or Character). Thus if method List.add expects an object, you may write

```csharp
List.add(123);
```

- Enumerations are provided:

```csharp
enum Color {Red, Blue, Green};
```

- Rectangular arrays are provided:

```csharp
int[,] multi = new int[5, 5];
```

- Reference, out and variable-length parameter lists are allowed.

Pizza

Pizza is an extension to Java developed in the late 90s by Odersky and Wadler.

Pizza shows that many of the best ideas of functional languages can be incorporated into a “mainstream” language, giving it added power and expressability.

Pizza adds to Java:

1. Parametric Polymorphism

   Classes can be parameterized with types, allowing the creation of “custom” data types with full compile-time type checking.

2. First-class Functions

   Functions can be passed, returned and stored just like other types.
3. Patterns and Value Constructors

Classes can be subdivided into a number of value constructors, and patterns can be used to structure the definition of methods.

---

Parametric Polymorphism

Java allows a form of polymorphism by defining container classes (lists, stacks, queues, etc.) in terms of values of type Object.

For example, to implement a linked list we might use:

```java
class LinkedList {
    Object value;
    LinkedList next;
    Object head() {return value;}
    LinkedList tail(){return next;}
    LinkedList(Object O) {
        value = O; next = null;
    }
    LinkedList(Object O, LinkedList L) {
        value = O; next = L;
    }
}
```

We use class Object because any object can be assigned to Object (all classes must be a subclass of Object).

Using this class, we can create a linked list of any subtype of Object.

But,

- We can't guarantee that linked lists are type homogeneous (contain only a single type).
- We must unbox Object types back into their "real" types when we extract list values.
- We must use wrapper classes like Integer rather than int (because primitive types like int aren't objects, and aren't subclass of Object).

For example, to use LinkedList to build a linked list of ints we do the following:

```java
LinkedList L =
    new LinkedList(new Integer(123));
int i =
    ((Integer) L.head()).intValue();
```

This is pretty clumsy code. We'd prefer a mechanism that allows us to create a "custom version" of LinkedList, based on the type we want the list to contain.

We can't just call something like LinkedList(int) or LinkedList(Integer) because types can't be passed as parameters.

Parametric polymorphism is the solution. Using this mechanism, we can use type parameters to build a
“custom version” of a class from a general purpose class.
C++ allows this using its template mechanism. Pizza also allows type parameters.
In both languages, type parameters are enclosed in “angle brackets” (e.g., LinkedList<T> passes T, a type, to the LinkedList class).
In Pizza we have
class LinkedList<T> {
    T value; LinkedList<T> next;
    T head() {return value;}
    LinkedList<T> tail() {
        return next;
    }
    LinkedList(T O) {
        value = O; next = null;
    }
    LinkedList(T O,LinkedList<T> L) {
        value = O; next = L;
    }
}

When linked list objects are created (using new) no type qualifiers are needed—the type of the constructor’s parameters are used. We can create
LinkedList<int> L1 =
    new LinkedList(123);
int i = L1.head();
LinkedList<String> L2 =
    new LinkedList("abc");
String s = L2.head();
LinkedList<LinkedList<int>> L3 =
    new LinkedList(L1);
int j = L3.head().head();

Bounded Polymorphism
In Pizza we can use interfaces to bound the type parameters a class will accept.
Recall our Compare interface:
interface Compare {
    boolean lessThan(Object o1, Object o2);
}
We can specify that a parameterized class will only takes types that implement Compare:
class LinkedList<T implements Compare> {
    ... }

In fact, we can improve upon how interfaces are defined and used. Recall that in method lessThan we had to use parameters declared as type Object to be general enough to match (and accept) any object type. This leads to clumsy casting (with run-time correctness checks) when lessThan is implemented for a particular type:
class IntCompare implements Compare {
    public boolean lessThan(Object i1, Object i2) {
        return ((Integer)i1).intValue() <
            ((Integer)i2).intValue();
    }
}
Pizza allows us to parameterize class definitions with type parameters, so why not do the same for interfaces?
In fact, this is just what Pizza does. We can now define Compare as

```java
interface Compare<T> {
    boolean lessThan(T o1, T o2);
}
```

Now we define class LinkedList as

```java
class LinkedList<T implements Compare<T>> { ... }
```

Given this form of interface definition, no casting (from type Object) is needed in classes that implement Compare:

```java
class IntCompare implements Compare<Integer> {
    public boolean lessThan(Integer i1, Integer i2) {
        return i1.intValue() < i2.intValue();
    }
}
```

First-class Functions in Pizza

In Java, functions are treated as constants that may appear only in classes.
To pass a function as a parameter, you must pass a class that contains that function as a member. For example,

```java
class Fct {
    int f(int i) { return i+1; }
}
class Test {
    static int call(Fct g, int arg) {
        return g.f(arg);
    }
}
```

Changing the value of a function is even nastier. Since you can't assign to a member function, you have to use subclassing to override an existing definition:

```java
class Fct2 extends Fct {
    int f(int i) { return i+111; }
}
```

Computing new functions during executions is nastier still, as Java doesn't have any notion of a lambda-term (that builds a new function).

Pizza makes functions first-class, as in ML. You can have function parameters, variables and return values. You can also define new functions within a method.

The notation used to define the type of a function value is

```latex
(T_1, T_2, \ldots) \rightarrow T_0
```

This says the function will take the list \((T_1, T_2, \ldots)\) as its arguments and will return \(T_0\) as its result.

Thus

```latex
(int) \rightarrow int
```

represents the type of a method like

```java
int plus1(int i) {return i+1;}
```
The notation used by Java for fixed functions still works. Thus
static int f(int i){return 2*i;};
denotes a function constant, $f$.
The definition
static (int)->int g = f;
defines a field of type (int)->int named $g$ that is initialized to the value of $f$.
The definition
static int call((int)->int f, int i)
    {return f(i);};
defines a constant function that takes as parameters a function value of type (int)->int and an int value. It calls the function parameter with the int parameter and returns the value the function computes.

Pizza also has a notation for anonymous functions (function literals), similar to $\text{fn}$ in ML and lambda in Scheme. The notation
fun (T_1 a_1, T_2 a_2, ...) -> T_0 (Body)
defines a nameless function with arguments declared as (T_1 a_1, T_2 a_2, ...) and a result type of T_0. The function's body is computed by executing the block (Body).

For example,
static (int)->int compose(
    (int)->int f, (int)->int g)
    {return f(g(i));};
defines a method named compose. It takes as parameters two functions, $f$ and $g$, each of type (int)->int.
The function returns a function as its result. The type of the result is (int)->int and its value is the composition of functions $f$ and $g$:
    return f(g(i));
Thus we can now have a call like
compose(f1,f2)(100)
which computes $f_1(f_2(100))$.

With function parameters, some familiar functions can be readily programmed:

```java
class Map {
    static int[] map((int)->int f, int[] a){
        int [] ans = new int[a.length];
        for (int i=0;i<a.length;i++)
            ans[i]=f(a[i]);
        return ans;
    }
}
```

```java
static int f(int i){return 2*i;};
```
And we can make such operations polymorphic by using parametric polymorphism:

class Map<T> {
    private static T dummy;
    Map(T val) {dummy=val;};
    static T[] map((T)->T f, T[] a) {
        T[] ans = (T[])a.clone();
        for (int i=0; i<a.length; i++)
            ans[i] = f(a[i]);
        return ans;
    }
}

Algebraic Data Types

Pizza also provides “algebraic data types” which allow a type to be defined as a number of cases. This is essentially the pattern-oriented approach we saw in ML.

A list is a good example of the utility of algebraic data types. Lists come in two forms, null and non-null, and we must constantly ask which form of list we currently have. With patterns, the need to consider both forms is enforced, leading to a more reliable programming style.

In Pizza, patterns are modeled as “cases” and grafted onto the existing switch statement (this formulation is a bit clumsy):

class List {
    case Nil;
    case Cons(char head, List tail);
    int length() {
        switch(this) {
            case Nil: return 0;
            case Cons(char x, List t): return 1 + t.length();
        }
    }
}

And guess what! We can use parametric polymorphism along with algebraic data types:

class List<T> {
    case Nil;
    case Cons(T head, List<T> tail);
    int length() {
        switch(this) {
            case Nil: return 0;
            case Cons(T x, List<T> t): return 1 + t.length();
        }
    }
}
Enumerations as Algebraic Data Types

We can use algebraic data types to obtain a construct missing from Java and Pizza—enumerations. We simply define an algebraic data type whose constructors are not parameterized:

```java
class Color {
    case Red;
    case Blue;
    case Green;
    String toString() {
        switch(this) {
            case Red: return "red";
            case Blue: return "blue";
            case Green: return "green";
        }
    }
}
```

This approach is better than simply defining enumeration values as constant (final) integers:

```java
final int Red = 1;
final int Blue = 2;
final int Green = 3;
```

With the algebraic data type approach, Red, Blue and Green, are not integers. They are constructors for the type Color. This leads to more thorough type checking.

Python

One of the newest and most innovative scripting languages is Python, developed by Guido van Rossum in the mid-90s. Python is named after the BBC “Monty Python” television series. Python blends the expressive power and flexibility of earlier scripting languages with the power of object-oriented programming languages.

It offers a lot to programmers:

- An interactive development mode as well as an executable “batch” mode for completed programs.
- Very reasonable execution speed. Like Java, Python programs are compiled. Also like Java, the compiled code is in an intermediate language for which an interpreter is written. Like Java this insulates Python from many of the vagaries of the actual machines on which it runs, giving it portability of an equivalent level to that of Java. Unlike Java, Python retains the interactivity for which interpreters are highly prized.
- Python programs require no compilation or linking. Nevertheless, the semi-compiled Python program still runs much faster than its traditionally interpreted rivals such as the shells, awk and perl.
- Python is freely available on almost all platforms and operating systems (Unix, Linux, Windows, MacOs, etc.)
• Python is completely object oriented. It comes with a full set of objected oriented features.
• Python presents a first class object model with first class functions and multiple inheritance. Also included are classes, modules, exceptions and late (run-time) binding.
• Python allows a clean and open program layout. Python code is less cluttered with the syntactic “noise” of declarations and scope definitions. Scope in a Python program is defined by the indentation of the code in question. Python thus breaks with current language designs in that white space has now once again acquired significance.

• Like Java, Python offers automated memory management through run-time reference counting and garbage collection of unreferenced objects.
• Python can be embedded in other products and programs as a control language.
• Python’s interface is well exposed and is reasonably small and simple.
• Python’s license is truly public. Python programs can be used or sold without copyright restrictions.
• Python is extendable. You can dynamically load compiled Python, Python source, or even dynamically load new machine (object) code to provide new features and new facilities.

• Python allows low-level access to its interpreter. It exposes its internal plumbing to a significant degree to allow programs to make use of the way the plumbing works.
• Python has a rich set of external library services available. This includes, network services, a GUI API (based on tcl/Tk), Web support for the generation of HTML and the CGI interfaces, direct access to databases, etc.

Using Python
Python may be used in either interactive or batch mode.
In interactive mode you start up the Python interpreter and enter executable statements. Just naming a variable (a trivial expression) evaluates it and echoes its value.
For example (>>> is the Python interactive prompt):

```python
>>> 1
1
>>> a=1
>>> a
1
>>> b=2.5
>>> b
2.5
```
You can also incorporate Python statements into a file and execute them in batch mode. One way to do this is to enter the command
`python file.py`
where `file.py` contains the Python code you want executed. Be careful though; in batch mode you must use a `print` (or some other output statement) to force output to be printed. Thus
```
1
a=1
```
```
b=2.5
b
a+b
print a+b
```
when run in batch mode prints only 3.5 (the output of the `print` statement).
You can also run Python programs as Unix shell scripts by adding the line
```
#!/usr/bin/env python
```
to the head of your Python file. (Since `#` begins Python comments, you can also feed the same augmented file directly to the Python interpreter)

**Python Command Format**

In Python, individual primitive commands and expressions must appear on a single line.
This means that
```
a = 1 + b
```
does not assign `1+b` to `a`! Rather, it assigns `1` to `a`, then evaluates `+b`.
If you wish to span more than one line, you must use `\` to escape the line:
```
a = 1 \ + b
```
is equivalent to
```
a = 1 + b
```

Compound statements, like `if` statements and `while` loops, can span multiple lines, but individual statements within an `if` or `while` (if they are primitive) must appear one a single line.

Why this restriction?
With it, `;`'s are mostly unnecessary!
A `;` at the end of a statement is legal but usually unnecessary, as the end-of-line forces the statement to end.
You can use `;` to squeeze more than one statement onto a line, if you wish:
```
a=1; b=2 ; c=3
```
Identifiers and Reserved Words

Identifiers look much the same as in most programming languages. They are composed of letters, digits and underscores. Identifiers must begin with a letter or underscore. Case is significant. As in C and C++, identifiers that begin with an underscore often have special meaning.

Python contains a fairly typical set of reserved words:

```
and  del  for  is  raise
assert  elif  from  lambda  return
break  else  global  not  try
class  except  if  or  while
continue  exec  import  pass
def  finally  in  print
```

Numeric Types

There are four numeric types:

1. Integers, represented as a 32 bit (or longer) quantity. Digits sequences (possibly) signed are integer literals:
   
   1  -123  +456

2. Long integers, of unlimited precision. An integer literal followed by an `l` or `L` is a long integer literal:

   12345678900000000000000L

3. Floating point values, represented as a 64 bit floating point number. Literals are of fixed decimal or exponential form:

   123.456  1e10  6.0231023

4. Complex numbers, represented as a pair of floating point numbers. In complex literals `j` or `J` is used to denote the imaginary part of the complex value:

   1.0+2.0j  -22.1j  10e10J+20.0

There is no character type. A literal like `"a"` or `"c"` denotes a string of length one.

There is no boolean type. A zero numeric value (any form), or `None` (the equivalent of void) or an empty string, list, tuple or dictionary is treated as false; other values are treated as true.

Hence

"abc" and "def"

is treated as true in an `if`, since both strings are non-empty.

Arithmetic Operators

<table>
<thead>
<tr>
<th>Op</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>**</td>
<td>Exponentiation</td>
</tr>
<tr>
<td>+</td>
<td>Unary plus</td>
</tr>
<tr>
<td>-</td>
<td>Unary minus</td>
</tr>
<tr>
<td>~</td>
<td>Bit-wise complement (int or long only)</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
</tr>
<tr>
<td>%</td>
<td>Remainder</td>
</tr>
<tr>
<td>+</td>
<td>Binary plus</td>
</tr>
<tr>
<td>-</td>
<td>Binary minus</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>Bit-wise left shift (int or long only)</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>Bit-wise right shift (int or long only)</td>
</tr>
<tr>
<td>&amp;</td>
<td>Bit-wise and (int or long only)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>^</td>
<td>Bit-wise Xor (int or long only)</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
</tr>
</tbody>
</table>
Operator Precedence Levels

Listed from lowest to highest:
- `or` Boolean OR
- `and` Boolean AND
- `not` Boolean NOT
- `<`, `<=`, `>`, `>=`, `!=`, `==` Comparisons
- `|` Bitwise OR
- `^` Bitwise XOR
- `&` Bitwise AND
- `<<`, `>>` Shifts
- `+`, `-` Addition and subtraction
- `*`, `/`, `%` Multiplication, division, remainder
- `**` Exponentiation
- `+`, `-` Positive, negative (unary)
- `~` Bitwise not

Arithmetic Operator Use

Arithmetic operators may be used with any arithmetic type, with conversions automatically applied. Bit-wise operations are restricted to integers and long integers. The result type is determined by the “generality” of the operands. (Long is more general than int, float is more general than both int and long, complex is the most general numeric type). Thus

```python
>>> 1+2
3
>>> 1+111L
112L
>>> 1+1.1
2.1
>>> 1+2.0j
(1+2j)
```

Unlike almost all other programming languages, relational operators may be “chained” (as in standard mathematics).

Therefore

```
a > b > c
```

means

```
(a > b) and (b > c)
```
Assignment Statements

In Python assignment is a statement not an expression.

Thus

\[ a + (b=2) \]

is illegal.

Chained assignments are allowed:

\[ a = b = 3 \]

Since Python is dynamically typed, the type (and value) associated with an identifier can change because of an assignment:

```python
>>> a = 0
>>> print a
0
>>> a = a + 0L
>>> print a
0L
>>> a = a + 0.0
>>> print a
0.0
>>> a = a + 0.0J
>>> print a
0j
```

If and While Statements

Python contains if and while statements that are fairly similar to those found in C and Java.

There are some significant differences though.

A line that contains an if, else or while ends in a ":". Thus we might write:

```python
if a > 0:
    b = 1
```

Moreover the indentation of the then part is significant! You don’t need {} in Python because all statements indented at the same level are assumed to be part of the same block.

In the following Python statements

```python
if a>0:
    b=1
c=2
d=3
```

the assignments to \(b\) and \(c\) constitute then part; the assignment to \(d\) follows the if statement, and is independent of it. In interactive mode a blank line is needed to allow the interpreter to determine where the if statement ends; this blank line is not needed in batch mode.
The if Statement

The full form of the if statement is:

```python
if expression:
    statement(s)
elif expression:
    statement(s)
...
else:
    statement(s)
```

Note those pesky ':'s at the end of the if, elif and else lines. The expressions following the if and optional elif lines are evaluated until one evaluates to true. Then the following statement(s), delimited by indentation, are executed. If no expression evaluates to true, the statements following the else are executed.

Use of else and elif are optional; a "bare" if may be used.

If any of the lists of statements is to be null, use pass to indicate that nothing is to be done. For example:

```python
if a>0:
    b=1
elif a < 0:
    pass
else:
    b=0
```

This if sets b to 1 if a is > 0; it sets b to 0 if a == 0, and does nothing if a < 0.

While Loops

Python contains a fairly conventional while loop:

```python
while expression:
    body
```

Note the ":" that ends the header line. Also, indentation delimits the body of the loop; no braces are needed. For example,

```python
>>> a=0; b=0
>>> while a < 5:
    ... b = b+a**2
    ... a= a+1
    ...
>>> print a,b
5 30
```

Break, Continue and Else in Loops

Like C, C++ and Java, Python allows use of break within a loop to force loop termination. For example,

```python
>>> a=1
>>> while a < 10:
    ... if a+a == a**2:
    ...     break
    ... else:
    ...     a=a+1
    ...
>>> print a
2
```
A `continue` may be used to force the next loop iteration:

```python
>>> a=1
>>> while a < 100:
...   a=a+1
...   if a%2==0:
...     continue
...   a=3*a
...
>>> print a
105
```

Python also allows you to add an `else` clause to a `while` (or `for`) loop. The syntax is:

```python
while expression:
  body
else:
  statement(s)
```

The `else` statements are executed when the termination condition becomes false, but not when the loop is terminated with a `break`. As a result, you can readily program “search loops” that need to handle the special case of search failure:

```python
>>> a=1
>>> while a < 1000:
...   if a**2 == 3*a-1:
...     print "winner: ",a
...     break
...   a=3*a
... else:
...   print "No match"
...
No match
```

## Sequence Types

Python includes three sequence types: strings, tuples and lists. All sequence types may be indexed, using a very general indexing system.

Strings are sequences of characters; tuples and lists may contain any type or combination of types (like Scheme lists).

Strings and tuples are immutable (their components may not be changed). Lists are mutable, and be updated, much like arrays.

Strings may be delimited by either a single quote (`'`) or a double quote (`"`) or even a triple quote (`'''` or `"""`). A given string must start and stop with the same delimiter. Triply quoted strings may span multiple lines. There
is no character type or value; characters are simply strings of length 1. Legal strings include 'abc' "xyz" 'It's OK!' '

Lists are delimited by "[" and "]". Empty (or null lists) are allowed. Valid list literals include

```
[1,2,3] ["one",1]
["a"],["b"],["c"] []
```

Tuples are a sequence of values separated by commas. A tuple may be enclosed within parentheses, but this isn't required. A empty tuple is (). A singleton tuple ends with a comma (to distinguish it from a simple scalar value).

Thus (1,) or just 1, is a valid tuple of length one.

**Indexing Sequence Types**

Python provides a very general and powerful indexing mechanism. An index is enclosed in brackets, just like a subscript in C or Java. Indexing starts at 0.

Thus we may have

```
>>> 'abcde'[2]
'c'
>>> [1,2,3,4,5][1]
2
>>> (1.1,2.2,3.3)[0]
1.1
```

Using an index that's too big raises an IndexError exception:

```
>>> 'abc'[3]
IndexError: string index out of range
```

Unlike most languages, you can use negative index values; these simply index from the right:

```
>>> 'abc'[-1]
'c'
>>> [5,4,3,2,1][-2]
2
>>> (1,2,3,4)[-4]
1
```

You may also access a slice of a sequence value by supplying a range of index values. The notation is

```
data[i:j]
```

which selects the values in `data` that are \( \geq i \) and \( < j \). Thus

```
>>> 'abcde'[1:2]
'b'
>>> 'abcde'[0:3]
'abc'
```

You may omit a lower or upper bound on a range. A missing lower bound defaults to 0 and a missing upper bound defaults to the maximum legal index. For example,

```
>>> [1,2,3,4,5][2:]
[3, 4, 5]
>>> [1,2,3,4,5][:3]
[1, 2, 3]
```

An upper bound that's too large in a range is interpreted as the maximum legal index:

```
>>> 'abcdef'[3:100]
'def'
```

You may use negative values in ranges too—they're interpreted as being relative to the right end of the sequence:
Since arrays may be assigned to, you may assign a slice to change several values at once:

```python
>>> a = [1, 2, 3, 4, 5]
>>> a[2:3] = [3.1, 3.2]
>>> a
[1, 2, 3.1, 3.2, 4, 5]
>>> a[4:] = []
>>> a
[1, 2, 3.1, 3.2]
>>> a[:0] = [-3, -2, -1]
>>> a
[-3, -2, -1, 1, 2, 3.1, 3.2]
```

**Other Operations on Sequences**

Besides indexing and slicing, a number of other useful operations are provided for sequence types (strings, lists, and tuples).

These include:

- **+ (catenation):**
  ```python
  >>> [1, 2, 3] + [4, 5, 6]
  [1, 2, 3, 4, 5, 6]
  >>> (1, 2, 3) + (4, 5)
  (1, 2, 3, 4, 5)
  >>> (1, 2, 3) + [4, 5]
  TypeError: illegal argument type for built-in operation
  >>> "abc" + "def"
  'abcdef'
  ```

- **- (subtraction):**

- *** (Repetition):**
  ```python
  >>> 'abc' * 2
  'abcabc'
  >>> [3, 4, 5] * 3
  [3, 4, 5, 3, 4, 5, 3, 4, 5]
  ```

- **Membership (in, not in):**
  ```python
  >>> 3 in [1, 2, 3, 4]
  1
  >>> 'c' in 'abcde'
  1
  ```

- **max and min:**
  ```python
  >>> max([3, 8, -9, 22, 4])
  22
  >>> min('aa', 'bb', 'abc')
  'aa'
  ```

- **Meaning of the length of the value assigned to a slice need not be the same size as the slice itself, so you can shrink or expand a list by assigning slices:**

  ```python
  >>> a = [1, 2, 3, 4, 5]
  >>> a[2:3] = [3.1, 3.2]
  >>> a
  [1, 2, 3.1, 3.2, 4, 5]
  >>> a[4:] = []
  >>> a
  [1, 2, 3.1, 3.2]
  >>> a[:0] = [-3, -2, -1]
  >>> a
  [-3, -2, -1, 1, 2, 3.1, 3.2]
  ```
Operations on Lists

As well as the operations available for all sequence types (including lists), there are many other useful operations available for lists. These include:

• **count** (Count occurrences of an item in a list):
  ```python
  >>> [1, 2, 3, 3, 21].count(3)
  2
  ```

• **index** (Find first occurrence of an item in a list):
  ```python
  >>> [1, 2, 3, 3, 21].index(3)
  2
  >>> [1, 2, 3, 3, 21].index(17)
  ValueError: list.index(x): x not in list
  ```

• **remove** (Find and remove an item from a list):
  ```python
  >>> a=[1, 2, 3, 4, 5]
  >>> a.remove(4)
  >>> a
  [1, 2, 3, 5]
  >>> a.remove(17)
  ValueError: list.remove(x): x not in list
  ```

• **pop** (Fetch and remove i-th element of a list):
  ```python
  >>> a=[1, 2, 3, 4, 5]
  >>> a.pop(3)
  4
  >>> a
  [1, 2, 3, 5]
  >>> a.pop()
  5
  >>> a
  [1, 2, 3]
  ```

• **reverse** a list:
  ```python
  >>> a=[1, 2, 3, 4, 5]
  >>> a.reverse()
  >>> a
  [5, 4, 3, 2, 1]
  ```

• **sort** a list:
  ```python
  >>> a=[5, 1, 4, 2, 3]
  >>> a.sort()
  >>> a
  [1, 2, 3, 4, 5]
  ```

• **Create a range of values**:
  ```python
  >>> range(1, 5)
  [1, 2, 3, 4]
  >>> range(1, 10, 2)
  [1, 3, 5, 7, 9]
  ```

Dictionaries

Python also provides a **dictionary type** (sometimes called an associative array). In a dictionary you can use a number (including a float or complex), string or tuple as an index. In fact any immutable type can be an index (this excludes lists and dictionaries).

An empty dictionary is denoted `{ }`. A non-empty dictionary may be written as

{ `key_1`: `value_1`, `key_2`: `value_2`, ... }

For example,

```python
>>> c={ 'bmw': 650, 'lexus': 'sc 430', 'mercedes': 'SL 500' }
```
You can use a dictionary much like an array, indexing it using keys, and updating it by assigning a new value to a key:

```python
>>> c['bmw']
650
>>> c['bmw']='z4 coupe'
>>> c['honda']='accord'
```

You can delete a value using `del`:

```python
>>> del c['honda']
>>> c['honda']
KeyError: honda
```

You can also check to see if a given key is valid, and also list all keys, values, or key-value pairs in use:

```python
>>> c.has_key('edsel')
0
>>> c.keys()
['bmw', 'mercedes', 'lexus']
>>> c.values()
['z4 coupe', 'SL 500', 'sc 430']
>>> c.items()
[('bmw', 'z4 coupe'), ('mercedes', 'SL 500'), ('lexus', 'sc 430')]
```

**For Loops**

In Python’s `for` loops, you don’t explicitly control the steps of an iteration. Instead, you provide a sequence type (a string, list or sequence), and Python automatically steps through the values.

Like a `while` loop, you must end the for loop header with a “:” and the body is delimited using indentation.

For example,

```python
>>> for c in 'abc':
...     print c
...
a
b
c
```

The `range` function, which creates a list of values in a fixed range is useful in `for` loops:

```python
>>> a=[5,2,1,4]
>>> for i in range(0,len(a)):
...     a[i]=2*a[i]
...
>>> print a
[10, 4, 2, 8]
```
You can use an `else` with `for` loops too. Once the values in the specified sequence are exhausted, the `else` is executed unless the `for` is exited using a `break`. For example,

```python
for i in a:
    if i < 0:
        print 'Neg val:', i
        break
else:
    print 'No neg vals'
```

### Function Definitions

Function definitions are of the form

```python
def name(args):
    body
```

The symbol `def` tells Python that a function is to be defined. The function is called `name` and `args` is a tuple defining the names of the function’s arguments. The body of the function is delimited using indentation. For example,

```python
def fact(n):
    if n<=1:
        return 1
    else:
        return n*fact(n-1)
>>> fact(5)
120
>>> fact(20L)
2432902008176640000L
>>> fact(2.5)
3.75
>>> fact(2+1j)
(1+3j)
```

Scalar parameters are passed by value; mutable objects are allocated in the heap and hence are passed (in effect) by reference:

```python
>>> def asg(ar):
...     a[1]=0
...     print ar
...     
...     a=[1,2,3,4.5]
>>> asg(a)
[1, 0, 3, 4.5]
```

Arguments may be given a default value, making them optional in a call. Optional parameters must follow required parameters in definitions. For example,

```python
>>> def expo(val,exp=2):
...     return val**exp
...
>>> expo(3,3)
27
>>> expo(3)
9
>>> expo()
TypeError: not enough arguments; expected 1, got 0
```
A variable number of arguments is allowed; you prefix the last formal parameter with a *; this parameter is bound to a tuple containing all the actual parameters provided by the caller:

```python
>>> def sum(*args):
...     sum=0
...     for i in args:
...         sum=sum+i
...     return sum
...
>>> sum(1,2,3)
6
>>> sum(2)
2
>>> sum()
0
```

You may also use the name of formal parameters in a call, making the order of parameters less important:

```python
>>> def cat(left=",",body="", right="]"):
...     return left+body+right
...
>>> cat(body='xyz');
'[xyz]'
>>> cat(body='hi there!',left='--[')
'--[hi there!]
```

Scoping Rules in Functions

Each function body has its own local namespace during execution. An identifier is resolved (if possible) in the local namespace, then (if necessary) in the global namespace. Thus

```python
>>> def f():
...     a=11
...     return a+b
...
>>> b=2;f()
13
>>> a=22;f()
13
>>> b=33;f()
44
```

Assignments are to local variables, even if a global exists. To force an assignment to refer to a global identifier, you use the declaration

```python
global id
```
which tells Python that in this function id should be considered global rather than local. For example,

```python
>>> a=1;b=2
>>> def f():
...     global a
...     a=111;b=222
...
>>> f();print a,b
111 2
```
Other Operations on Functions

Since Python is interpreted, you can dynamically create and execute Python code.

The function `eval(string)` interprets `string` as a Python expression (in the current execution environment) and returns what is computed. For example,

```python
>>> a=1; b=2
>>> eval('a+b')
3
```

`exec(string)` executes `string` as arbitrary Python code (in the current environment):

```python
>>> a=1; b=2
>>> exec('for op in "+-*/":
print(eval("a"+op+"b"))')
3
-1
2
0
```

`execfile(string)` executes the contents of the file whose pathname is specified by `string`. This can be useful in loading an existing set of Python definitions.

The expression

```
lambda args: expression
```

creates an anonymous function with `args` as its argument list and `expression` as its body. For example,

```python
>>> (lambda a:a+1)(2)
3
```

And there are definitions of `map`, `reduce` and `filter` to map a function to a list of values, to reduce a list (using a binary function) and to select values from a list (using a predicate):

```python
>>> def double(a):
... return 2*a;
...
>>> map(double, [1,2,3,4])
[2, 4, 6, 8]
```

```python
>>> def sum(a,b):
... return a+b
...
>>> reduce(sum, [1,2,3,4,5])
15
```

```python
>>> def even(a):
... return not(a%2)
...
>>> filter(even, [1,2,3,4,5])
[2, 4]
```
I/O in Python

The easiest way to print information in Python is the `print` statement. You supply a list of values separated by commas. Values are converted to strings (using the `str()` function) and printed to standard out, with a terminating new line automatically included. For example,

```python
>>> print "1+1=",1+1
1+1= 2
```

If you don’t want the automatic end of line, add a comma to the end of the print list:

```python
>>> for i in range(1,11):
...    print i,
...
1 2 3 4 5 6 7 8 9 10
```

For those who love C’s `printf`, Python provides a nice formatting capability using a printf-like notation. The expression

```
format % tuple
```

formats a tuple of values using a format string. The detailed formatting rules are those of C’s `printf`. Thus

```python
>>> print "%d+%d=%d" % (10,20,10+20)
10+20=30
```

File-oriented I/O

You open a file using

```python
open(name,mode)
```

which returns a “file object.”

`name` is a string representing the file’s path name; `mode` is a string representing the desired access mode (‘r’ for read, ‘w’ for write, etc.).

Thus

```python
>>> f=open("/tmp/f1","w")
>>> f
<open file '/tmp/f1', mode 'w' at decd8>
```

opens a temp file for writing.

The command

```python
f.read(n)
```

reads `n` bytes (as a string).
Classes in Python

Python contains a class creation mechanism that's fairly similar to what's found in C++ or Java. There are significant differences though:

- All class members are public.
- Instance fields aren't declared. Rather, you just create fields as needed by assignment (often in constructors).
- There are class fields (shared by all class instances), but there are no class methods. That is, all methods are instance methods.

All instance methods (including constructors) must explicitly provide an initial parameter that represents the object instance. This parameter is typically called self. It's roughly the equivalent of this in C++ or Java.

Defining Classes

You define a class by executing a class definition of the form

class name:
    statement(s)

A class definition creates a class object from which class instances may be created (just like in Java). The statements within a class definition may be data members (to be shared among all class instances) as well as function definitions (prefixed by a def command). Each function must take (at least) an initial parameter that represents the class instance within which the function (instance method) will operate. For example,
class Example:
    cnt=1
    def msg(self):
        print "Bo"+"o"*Example.cnt+"!"*self.n
    >>> Example.cnt
    1
    >>> Example.msg
    <unbound method Example.msg>
Example.msg is unbound because we haven't created any instances of the Example class yet.
We create class instances by using the class name as a function:
    >>> e=Example()
    >>> e.msg()
    AttributeError: n
We get the AttributeError message regarding n because we haven't defined n yet! One way to do this is to just assign to it, using the usual field notation:
    >>> e.n=1
    >>> e.msg()
    Boo!
    >>> e.n=2;Example.cnt=2
    >>> e.msg()
    Booo!!
We can also call an instance method by making the class object an explicit parameter:
    >>> Example.msg(e)
    Booo!!
It's nice to have data members initialized when an object is created. This is usually done with a constructor, and Python allows this too.

A special method named __init__ is called whenever an object is created. This method takes self as its first parameter; other parameters (possibly made optional) are allowed.
We can therefore extend our Example class with a constructor:

```python
class Example:
    cnt=1
    def __init__(self,nval=1):
        self.n=nval
    def msg(self):
        print "Bo"+"o"*Example.cnt+"!"*self.n
    >>> e=Example()
    >>> e.n
    1
    >>> f=Example(2)
    >>> f.n
    2
```
You can also define the equivalent of Java's toString method by defining a member function named __str__(self).
For example, if we add
def __str__(self):
    return "<%d>"%self.n
to Example, then we can include Example objects in print statements:
    >>> e=Example(2)
    >>> print e
    <2>
Inheritance

Like any language that supports classes, Python allows inheritance from a parent (or base) class. In fact, Python allows multiple inheritance in which a class inherits definitions from more than one parent.

When defining a class you specify parents classes as follows:

```python
class name(parent classes):
    statement(s)
```

The subclass has access to its own definitions as well as those available to its parents. All methods are virtual, so the most recent definition of a method is always used.

```python
class C:
    def DoIt(self):
        self.PrintIt()
    def PrintIt(self):
        print "C rules!"

class D(C):
    def PrintIt(self):
        print "D rules!"
    def TestIt(self):
        self.DoIt()

dvar = D()
dvar.TestIt()
D rules!
```

If you specify more than one parent for a class, lookup is depth-first, left to right, in the list of parents provided. For example, given

```python
class A(B,C):
```

we first look for a non-local definition in B (and its parents), then in C (and its parents).

Operator Overloading

You can overload definitions of all of Python's operators to apply to newly defined classes. Each operator has a corresponding method name assigned to it. For example, + uses `__add__`, - uses `__sub__`, etc.
Given

class Triple:
    def __init__(self, A=0, B=0, C=0):
        self.a = A
        self.b = B
        self.c = C
    def __str__(self):
        return "(%d, %d, %d)" % (self.a, self.b, self.c)
    def __add__(self, other):
        return Triple(self.a + other.a,
                      self.b + other.b,
                      self.c + other.c)

the following code

t1 = Triple(1, 2, 3)
t2 = Triple(4, 5, 6)
print t1 + t2
produces
(5, 7, 9)

Exceptions

Python provides an exception mechanism that’s quite similar to the one used by Java.
You “throw” an exception by using a raise statement:
    raise exceptionValue

There are numerous predefined exceptions, including
OverflowError (arithmetic overflow), EOFError (when end-of-file is hit), NameError (when an undeclared identifier is referenced), etc.

You may define your own exceptions as subclasses of the predefined class Exception:

class badValue(Exception):
    def __init__(self, val):
        self.value = val

You catch exceptions in Python’s version of a try statement:

try:
    statement(s)
except exceptionName1, id1:
    statement(s)
...
except exceptionNameN, idN:
    statement(s)

As was the case in Java, an exception raised within the try body is handled by an except clause if the raised exception matches the class named in the except clause. If the raised exception is not matched by any except clause, the next enclosing try is considered, or the exception is reraised at the point of call.

For example, using our badValue exception class,

def sqrt(val):
    if val < 0.0:
        raise badValue(val)
    else:
        return cmath.sqrt(val)

try:
    print "Ans =", sqrt(-123.0)
except badValue, b:
    print "Can’t take sqrt of", b.value

When executed, we get
Ans = Can’t take sqrt of -123.0
Modules

Python contains a module feature that allows you to access Python code stored in files or libraries. If you have a source file `mydefs.py` the command

```python
import mydefs
```

will read in all the definitions stored in the file. What’s read in can be seen by executing

```python
dir(mydefs)
```

To access an imported definition, you qualify it with the name of the module. For example,

```python
mydefs.fct
```

accesses `fct` which is defined in module `mydefs`.

To avoid explicit qualification you can use the command

```python
from modulename import id_1, id_2, ...
```

This makes `id_1`, `id_2`, ... available without qualification. For example,

```python
>>> from test import sqrt
>>> sqrt(123)
(11.0905365064+0j)
```

You can use the command

```python
from modulename import *
```

to import (without qualification) all the definitions in `modulename`.

The Python Library

One of the great strengths of Python is that it contains a vast number of modules (at least several hundred) known collectively as the Python Library. What makes Python really useful is the range of prewritten modules you can access. Included are network access modules, multimedia utilities, data base access, and much more.

See

```python
www.python.org/doc/lib
```

for an up-to-date listing of what’s available.