

Structure of Programming Languages – Lecture 8

CS 636 – 536

June 1, 2009

- 1 Expressions and Evaluation
 - Assignment
 - Functions and Operators
- 2 Functions and Parameters
 - Passing Parameters
- 3 Miranda
 - Functions and Syntax
 - Order of Evaluation
 - Miranda Lists

Assignment is...

One of these:

- A statement – Pascal.
- An expression with a result: C
- Parallel: Python `a, b = b, a`
- A way to introduce a variable name: APL, Python
- Replaced by binding: APL, Miranda, Python, functional languages

Python has a "global" declaration to permit a function to bind a value to a semi-global name.

Operators are Functions

- Infix operator syntax is an alternative way to call a function. Because it is ambiguous, the simple rules for parsing function calls must be extended – usually with rules for precedence and associativity.
- Prefix and postfix expressions do not need parentheses – they are not ambiguous in languages where the number of operands is known for every operator.
- Postfix is the simplest to parse and to evaluate, since it fits the properties of a stack.
- LISP uses prefix syntax WITH parentheses. It also allows variable numbers of arguments for functions.
- APL: right-to-left, infix and prefix operators, arrays, parens.
 $(B - C) \times 1\ 3\ 5 + \rho\ ARY \times 2$

Familiar Ways to Pass Parameters

- Call-by-value: evaluate the argument expression and store the resulting value in the stack frame, where it will be bound to the parameter name. Possible with l- or r-values.
 - C, except for arrays and function-arguments.
 - Java, primitive values only.
- Call-by-reference: Only possible with l-values. Store the address of the argument in the stack frame. The parameter name becomes an alias for the argument name.
 - C arrays and function-arguments.
 - Java, all objects.
 - FORTRAN-IV, always. (If the argument is the result of an expression, it is stored somewhere and that address is passed. If it is a literal constant, the address of the constant is passed. In either case, the function can change that address and create garbage.)

Some Less-Familiar Ways.

- Call-by-result: A write-only reference parameter.
 - Ada “out” parameters
- Call-by-value-and-result: Call-by-value is used to copy the argument into the a parameter variable within the function. Just before the function returns, the contents of the parameter are copied back into the caller’s variable. Between those events, the caller’s variable cannot be read or changed.
 - Ideal for remote-procedure call, to minimize network traffic.
 - Ideal for use with threads, to avoid inconsistencies.

Functional Languages

- Call-by-name: The parameter is bound to the unevaluated argument expression.
 - A mistake made in Algol, trying to emulate Lambda Calculus.
 - Does not guarantee the same parameter value will be used throughout the function.
- Call-by-need: The parameter is bound to the unevaluated argument expression. Evaluation happens the first time the argument is needed, and the result of evaluation then replaces the unevaluated expression.
 - Miranda
 - ML, Haskell, and other modern functional languages

The Problem with Functional Arguments

- Free Variables
 - If a function refers to a free variable,
 - and that function is passed as an argument to another function,
 - the free variable must be interpreted in the context of the caller, not the called.
 - A further question is whether it should be the contents of the variable *at the time of the call* or later, *when the argument is used* in an expression.
- This is known as the "fun-arg problem".
- It is related to the idea of a **closure** – a function together with a context that supplies bindings for one or more of the function's parameters.

The Functional Philosophy

- A result of using lazy evaluation is that the sequence in which lines of code are written is not the same as the order in which they are evaluated. (A statement sequence is an illusion—the results would be the same if the lines were scrambled.)
- Instead, evaluation order is determined by how function calls are nested in the code, and by whether local variables and parameters are used at all.
- Functional languages use dynamic binding (no declarations) to attach names to objects, and allow only one binding during the lifetime of the name.
- Parameter binding is the primary means naming a value.

Miranda Basics: Function definitions, computation.

Lambda Calculus	$sq = \lambda n. \quad * \ n \ n$ $z = / \ (sq \ x) \ (sq \ y)$
Scheme, LISP	<code>(defun sq(n) (* n n))</code>
Scheme	<code>(define sq (lambda (n) (* n n)))</code> <code>(define z (/ (sq x) (sq y)))</code>
Python	<code>def sq(n): return n*n</code> <code>sq = lambda n: n*n</code> <code>z = sq(x) / sq(y)</code>
Miranda	<code>sq n = n * n</code> <code>z = sq x / sq y</code>
FORTH	<code>: sq (n1 >> n2) dup * ;</code> <code>x sq y sq z !</code>

Miranda: a parallel IF

- The conditions on the right are called *guards*.
- The guards are evaluated in parallel.
- One of them is supposed to be true.
- The clause (on the left) corresponding to the true guard is executed.
- The result of that clause is the result of the expression.

$$\begin{aligned} \text{gcd } a \ b &= \text{gcd } (a-b) \ b, & a > b \\ &= \text{gcd } a \ (b-a), & a < b \\ &= a, & a = b \end{aligned}$$

A very elegant gcd, but not very efficient because it uses repeated subtraction rather than division.

In Praise of Laziness

A result from **Lambda Calculus**:

Outside-in evaluation is strictly more powerful than inside-out:

```
if ( x != 0 ) answer = z/x; else answer = z;
```

O-I: Evaluate `if (x != 0)` first, then choose one clause.

I-O: Evaluate `z/x` first and bomb with a divide-by-zero error.

Every programming language evaluates conditionals outside in.
Most languages evaluate everything else inside-out.

Modern functional languages use outside-in evaluation consistently;
it is called **lazy evaluation**. Nothing is evaluated until and unless
the result is needed.

Miranda Basics: = is Binding, not assignment.

Either a variable has no meaning (no binding) or it has the same meaning it had originally. The meaning cannot be changed by an assignment.

Here is a function to compute the roots of a quadratic equation:

```
root a b c = error "complex roots",      delta < 0
            = [term1],                    delta = 0
            = [term1+term2, term1-term2], delta > 0
  where
    delta = b*b - 4*a*c
    radix = sqrt delta
    term1 = -b/(2*a)
    term2 = radix/(2*a)
```

Examples: Lazy evaluation.

Example: calculate root 1 0 1

- ① Bindings: $a=1$, $b=0$, $c=1$
- ② Start evaluating guarded expression; need a value for delta
- ③ Evaluate $\text{delta} = b*b - 4*a*c = 0^2 - 4 \times a \times c = -4$
- ④ Choose 1st alternative for guarded expression.
- ⑤ Return answer “complex roots”

Example: calculate root 1 -2 1

- ① Bindings: $a=1$, $b=-2$, $c=1$
- ② Start evaluating guarded expression – need a value for delta
- ③ Evaluate $\text{delta} = b*b - 4*a*c = (-2)^2 - 4 \times 1 \times 1 = 4 - 4 = 0$
- ④ Choose 2nd alternative for guarded expression.
- ⑤ Evaluate $\text{term1} = -b/(2*a) = -(-2)/(2 \times 1) = 2/2 = 1$
- ⑥ Finish step 4; return answer = [1]

Lazy evaluation sometimes evaluates everything.

Calculate `root 1 0 -1` Lazy evaluation is used throughout.

- 1 Bindings: `a=1, b=0, c=-1`
- 2 Start evaluating guarded expression; need a value for `delta`
- 3 Evaluate `delta = b*b-4*a*c = 02 - 4 × a × c = -(-4) = 4`
- 4 Choose 3rd alternative for guarded expression.
- 5 Evaluate `term1 = -b/(2*a) = -0/(2 × 1) = 0`
- 6 Start evaluating `term2`: need a value for `radix`.
- 7 Evaluate `radix = sqrt delta = √4 = 2`
- 8 Evaluate `term2 = radix/(2*a) = 2/(2 × 1) = 1`
- 9 Finish step 4; return answer = `[1, -1]`

Miranda Lists.

A list can be written literally or denoted by an expression.

`["one", "two", "three"]` A list of three strings.

`[]` The empty list.

`L1 = [2..6]` Integers from 2 to 6, inclusive.

`L2 = [1, 3 .. 100]` An arithmetic progression, 1, 3, 5 to 99

`[1, 3 ..]` An arithmetic progression, 1, 3, 5, etc.

`L3 = L2 -- L1` `[1, 7, 9..100]`

`N = #L2` 50, The number of items in L2.

List Comprehensions.

A list can be written literally or denoted by an expression.

L4 = $[n + 2 | n \leftarrow 13..20]$ [15..22]

L5 = $[n * n | n \leftarrow 1, 2..]$ All square numbers.

L6 = $[x + y | x \leftarrow 2..5; y \leftarrow 1..5; x > y]$ [3,4,5,5,6,7,6,7,8,9]

An **infinite list** such as L5 is implemented as the closure of a finite head (an array or list) followed by a tail which is a function.

The tail is prodded into action, when necessary to lengthen the list, in response to the program accessing an unevaluated subscript position.