CS 314 Principles of Programming Languages

Lecture 14

Zheng Zhang

Department of Computer Science
Rutgers University

Friday 21\textsuperscript{st} October, 2016
Class Information

- Homework 5 due this Saturday, October 22, 11:55pm. Need an extension?
- Reminder: midterm exam in less than two weeks, November 2 Wednesday, in class, closed book, closed notes.
Review: Procedures

- Modularize program structure
  - Argument: information passed from caller to callee (actual parameter)
  - Parameter: local variable whose value (usually) is received from caller (formal parameter)
- Procedure declaration
  - procedure name, formal parameters, procedure body with local declarations and statement lists, optional result type
  - example: `void translate(point *p, int dx)`
Parameter Passing Modes

- **pass-by-value**: C/C++, Java/C#, Pascal, Scheme, Algol 68
- **pass-by-result**: Algol W, Ada (out parameter)
- **pass-by-value-result**: Ada (in out parameter)
- **pass-by-reference**: C/C++, Fortran, Pascal (var parameter)
begin
  c: array[1..10] of integer;
  m, n: integer;
  procedure r(k, j: integer)
  begin
    k := k+1;
    j := j+2;
    end r;
  ...
  m := 5;
  n := 3
  r(m,n);
  write m,n;
end

Output:
5 3

Advantage: Argument protected from changes in callee
Disadvantage: Copying of values takes execution time and space, especially for aggregate values (e.g.: arrays, structs).
begin
  c: array[1..10] of integer;
  m, n: integer;
  procedure r(k, j: integer)
  begin
    k := k+1;
    j := j+2;
    end r;
...
  m := 5;
  n := 3
  r(m,n);
  write m,n;
end

Output:
6 5

Advantage: more efficient than copying
Disadvantage: leads to aliasing: there are two or more names for the same storage location; hard to track side effects
begin
  c: array[1..10] of integer;
  m, n: integer;
  procedure r(k, j: integer)
  begin
    k := k+1;  ==> ERROR: CANNOT USE PARAMETERS
    j := j+2;  WHICH ARE UNINITIALIZED
    end r;
...
  m := 5;
  n := 3
  r(m,n);
  write m,n;
end

Output: program doesn’t compile or has runtime error
Pass-by-result

begin
    c: array[1..10] of integer;
    m, n: integer;
    procedure r(k, j: integer)
    begin
        k := 1; ==> HERE IS ANOTHER PROGRAM
        j := 2; THAT WORKS
        end r;
    ...
    m := 5;
    n := 3
    r(m,m); ==> NOTE: CHANGED THE CALL
    write m,n;
end

Output: 1 or 2?
Problem: order of copy-back makes a difference; implementation dependent.

Zheng Zhang
begin
  c: array[1..10] of integer;
  m, n: integer;
  procedure r(k, j: integer)
  begin
    k := k+1;
    j := j+2;
  end r;
...
  m := 5;
  n := 3
  r(m,n); 
  write m,n;
end

Output:
6 5

Problem: order of copy-back can make a difference; implementation dependent.
begin
  c: array[1..10] of integer;
  m, n: integer;
  procedure r(k, j: integer)
  begin
    k := k+1;
    j := j+2;
    end r;
...
  /* set c[m] = m */
  m := 2;
  r(m,c[m]); ==> WHAT ELEMENT OF ‘‘c’’ IS ASSIGNED TO?
  write c[1], c[2], ... c[10];
end

Output:
1 4 3 4 5 ... 10 on entry
1 2 4 4 5 ... 10 on exit

Problem: When is the address computed for the copy-back operation? At procedure call (procedure entry), just before procedure exit, somewhere inbetween? (Example: ADA on entry)
Aliasing:

More than two ways to name the same object within a scope

Even without pointers, you can have aliasing through (global ↔ formal) and (formal ↔ formal) parameter passing.

```plaintext
begin
    j, k, m: integer;
    procedure q(a,b: integer);
    begin
        b := 3;
        m := m*a;
    end
    ...
    q(m,k);  ==> global/formal <m,a> ALIAS PAIR
    q(j,j);  ==> formal/formal <a,b> ALIAS PAIR
    write y;
end
```

Zheng Zhang
Comparison: by-value-result vs. by-reference

Actual parameters need to evaluate to L-values (addresses).

```pascal
begin
  y: integer;
  procedure p(x: integer);
  begin
    x := x+1;  ==> ref: x and y are ALIASED
    x := x+y;  ==> val-res: x and y are NOT ALIASED
  end
  ...
  y := 2;
  p(y);
  write y;
end
```

Output:
- pass-by-reference: 6
- pass-by-value-result: 5

Note: by-value-result: Requires copying of parameter values (expensive for aggregate values); does not have aliasing, but copy-back order dependence;
Functional Programming

Pure Functional Languages

Scott: Chapter 11

Fundamental concept: **application** of (mathematical) **functions** to **values**

1. **Referential transparency**: The value of a function application is independent of the context in which it occurs
   - value of \( f(a, b, c) \) depends only on the values of \( f, a, b \) and \( c \)
   - It does not depend on the global state of computation
   \( \Rightarrow \) all vars in function must be local (or parameters)
2. The concept of assignment is **not** part of functional programming
   - no explicit assignment statements
   - variables bound to values only through the association of actual parameters to formal parameters in function calls
   - function calls have no side effects
   - thus no need to consider global state

3. Control flow is governed by function calls and conditional expressions
   - no iteration
   - recursion is widely used
4. All storage management is implicit
   ▶ needs garbage collection

5. Functions are *First Class Values*
   ▶ Can be returned as the value of an expression
   ▶ Can be passed as an argument
   ▶ Can be put in a data structure as a value
   ▶ (Unnamed) functions exist as values
Pure Functional Languages

A program includes:

1. A set of function definitions
2. An expression to be evaluated

E.g. in Scheme:

> (define length
   (lambda (x)
     (if (null? x)
         0
         (+ 1 (length (rest x))))))

> (length '(A LIST OF 5 THINGS))
5
LISP

- Functional language developed by John McCarthy in the mid 50’s
- Semantics based on *Lambda Calculus*
- All functions operate on lists or symbols: (called “S-expressions”)
- Only five basic functions: list functions `cons`, `car`, `cdr`, `equal`, `atom` and one conditional construct: `cond`
- Useful for list-processing applications
- Programs and data have the same syntactic form: S-expressions
- Used in Artificial Intelligence
Lambda calculus

- formalism for studying ways in which functions can be formed, combined, and used for computation
- computation is defined as rewriting rules (operational semantics) ⇒ \( \beta \) reduction
- the syntactic notion of computation was developed first; a mathematical semantics followed much later

Examples:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Mathematical Notation</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f(x) = x+2 )</td>
<td>( \lambda x.x+2 )</td>
<td>different notation</td>
</tr>
<tr>
<td>( (\lambda x.x+2) \ 1 )</td>
<td>1+2 = 3</td>
<td>function application</td>
</tr>
<tr>
<td>( (\lambda x.x) \ (\lambda y.y) )</td>
<td>arguments and returned</td>
<td>“values” can be functions</td>
</tr>
<tr>
<td>( \lambda x.xx )</td>
<td>( f(x) = x(x) )</td>
<td>untyped lambda calculus</td>
</tr>
</tbody>
</table>

Zheng Zhang
SCHEME

- Developed in 1975 by G. Sussman and G. Steele
- A version of LISP
- Simple syntax, small language
- Closer to initial semantics of LISP as compared to COMMON LISP
- Provides basic list processing tools
- Allows functions to be first class objects
Expressions are written in prefix, parenthesized form

- \((\text{function } \text{arg}_1 \ \text{arg}_2 \ \ldots \ \text{arg}_n)\)
- \((+ \ 4 \ 5)\)
- \((+ \ (\times \ 3 \ 4) \ (- \ 5 \ 3))\)

Operational semantics: In order to evaluate an expression:

1. evaluate function to a function value
2. evaluate each \(\text{arg}_i\) in order to obtain its value
3. apply the function value to these values
S-expressions

S-expression ::= Atom | ‘(’ { S-expression } ‘)’
Atom ::= Name | Number | #t | #f

#t
()
(a b c)
(a (b c) d)
(((a b c) (d e (f)))
(1 (b) 2)

Lists have nested structure.
Lists in Scheme

The building blocks for lists are pairs or cons-cells. Lists use the empty list ( ) as an “end-of-list” marker.

Note: (a.b) is not a list!
Special (Primitive) Functions

- `eq?`: identity on names (atoms)
- `null?`: is list empty?
- `car`: selects first element of list (*contents of address part of register*)
- `cdr`: selects rest of list (*contents of decrement part of register*)
- `(cons element list)`: constructs lists by adding element to front of list
- `quote` or `'`: produces constants
Special (Primitive) Functions

- '() is the empty list

- (car '(a b c)) =

- (car '(((a) b (c d)))) =

- (cdr '(a b c)) =

- (cdr '(((a) b (c d)))) =
Special (Primitive) Functions

- car and cdr can break up any list:
  - \((\text{car} (\text{cdr} (\text{cdr} '((a) b (c d))))) = \)

  \((\text{cdr} '((a) b (c d))) \)

- cons can construct any list:
  - \((\text{cons} 'a () = \)

  \((\text{cons} 'd '(e)) = \)

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

  \((\text{cons} '(a b c) '((a) b)) = \)
More Scheme and higher-order functions