

7. Control Flow

Overview

- Expressions
 - Evaluation order
 - Assignments
- Structured and unstructured flow constructs
 - Goto's
 - Sequencing
 - Selection
 - Iteration and iterators
 - Recursion
 - Nondeterminacy

Note: Study Chapter 6 of the textbook except Section 6.6.2.

Ordering Program Execution: What is Done First?

Categories for specifying ordering in programming languages:

1. *Sequencing*: the execution of statements and evaluation of expressions is usually in the order in which they appear in a program text
2. *Selection (or alternation)*: a run-time condition determines the choice among two or more statements or expressions
3. *Iteration*: a statement is repeated a number of times or until a run-time condition is met
4. *Procedural abstraction*: subroutines encapsulate collections of statements and subroutine calls can be treated as single statements
5. *Recursion*: subroutines which call themselves directly or indirectly to solve a problem, where the problem is typically defined in terms of simpler versions of itself
6. *Concurrency*: two or more program fragments executed in parallel, either on separate processors or interleaved on a single processor
7. *Nondeterminacy*: the execution order among alternative constructs is deliberately left unspecified, indicating that any alternative will lead to a correct result

Expression Syntax

- An expression consists of
 - An atomic object, e.g. number or variable
 - An *operator* applied to a collection of *operands* (or arguments) which are expressions
- Common syntactic forms for operators:
 - *Function call notation*, e.g. `somefunc(A, B, C)`, where `A`, `B`, and `C` are expressions
 - *Infix notation for binary operators*, e.g. `A + B`
 - *Prefix notation for unary operators*, e.g. `-A`
 - *Postfix notation for unary operators*, e.g. `i++`
 - *Cambridge Polish notation*, e.g. `(* (+ 1 3) 2)` in Lisp $= (1+3)*2=8$
 - *"Multi-word" infix*, e.g. `a>b?a:b` in C and `myBox displayOn: myScreen at: 100@50` in Smalltalk, where `displayOn:` and `at:` are written infix with arguments `myBox`, `myScreen`, and `100@50`

Expression Evaluation Ordering: Precedence and Associativity

- The use of infix, prefix, and postfix notation leads to ambiguity as to what is an operand of what
 - Fortran example: `a+b*c**d**e/f`
- The choice among alternative evaluation orders depends on
 - *Operator precedence*: higher operator precedence means that a (collection of) operator(s) group more tightly in an expression than operators of lower precedence
 - *Operator associativity*: determines evaluation order of operators of the same precedence
 - *left associative*: operators are evaluated left-to-right (most common)
 - *right associative*: operators are evaluated right-to-left (Fortran power operator `**`, C assignment operator `=` and unary minus)
 - *non-associative*: requires parenthesis when composed (Ada power operator `**`)
- Pascal's flat precedence levels is a design mistake
 - `if A<B and C<D then` is read as `if A<(B and C)<D then`
- Note: levels of operator precedence and associativity are easily captured in a grammar as we saw earlier.

Operator Precedence Levels

Fortran	Pascal	C/C++	Ada
		++ -- (post-inc/dec)	

** (power)	not	++ -- (pre-inc/dec) + - (unary) & (address of) * (contents of) ! (logical not) ~ (bit-wise not)	abs not **
* /	* / div mod and	* (binary) / % (mod)	* / mod rem
+ -	+ - (unary and binary) or	+ - (binary)	+ - (unary)
		<< >>	+ - (binary) & (concatenation)
.EQ. .NE. .LT. .LE. .GT. .GE. (comparisons)	= <> < <= > >=	< > <= >=	= /= < <= > >=
.NOT.		== !=	
		& (bit-wise and)	
		^ (bit-wise xor)	
		(bit-wise or)	
.AND.		&& (logical and)	and or xor (logical)
.OR.		(logical or)	
.EQV. .NEQV. (logical comparisons)		?: (if-then-else)	
		= += -= *= /= %= >> <<= &= ^= =	

		, (sequencing)	
--	--	----------------	--

Evaluation Order in Expressions

- Precedence and associativity define rules for *structuring* expressions, but do *not* determine the operand evaluation order!
 - Expression $a-f(b)-c*d$ is structured as $(a-f(b))-(c*d)$, but either $(a-f(b))$ or $(c*d)$ can be evaluated first when the program runs
 - Also the evaluation order of arguments in function and subroutine calls can differ
- Knowing the operand evaluation order is important
 - *Side effects*: e.g. if $f(b)$ above modifies d (i.e. $f(b)$ has a side effect) the expression value will depend on the operand evaluation order
 - *Code improvement*: compilers rearrange expressions to maximize efficiency
 - *Improve memory loads*:
 $a:=B[i];$ *load a from memory*
 $c:=2*a+3*d;$ *compute 3*d first, because waiting for a to arrive in processor*
 - *Common subexpression elimination*:
 $a:=b+c;$
 $d:=c+e+b;$ *rearranged as $d:=b+c+e$, it can be rewritten into $d:=a+e$*
 - *Register allocation*: rearranging operand evaluation can decrease the number of processor registers used for temporary values (register spill)

Expression Reordering Issues

- Rearranging expressions may lead to arithmetic overflow or different floating point results
 - Assume b , d , and c are very large positive integers, then if $b-c+d$ is rearranged into $(b+d)-c$ arithmetic overflow occurs
 - Floating point value of $b-c+d$ may differ from $b+d-c$
 - Most programming languages will not rearrange expressions when parenthesis are used, e.g. write $(b-c)+d$ to avoid problems
- Design choices:
 - Java: expressions evaluation is always left to right and overflow is always detected
 - Pascal: expression evaluation is unspecified and overflows are always detected
 - C and C++: expression evaluation is unspecified and overflow detection is implementation dependent
 - Lisp: no limit on number representation

Short-Circuit Evaluation

- *Short-circuit evaluation* of Boolean expressions means that in certain cases the result of a binary Boolean operator can be determined from the evaluation of just one operand
- Pascal does not use short-circuit evaluation
 - The program fragment below has the problem that element $a[11]$ can be accessed resulting in a dynamic semantic error:

```

var a:array [1..10] of integer;
...
i:=1;
while i<=10 and a[i]>0 do
  i:=i+1

```
- C, C++, and Java use short-circuit *conditional and/or* operators
 - If a in $a \&\& b$ evaluates to false, b is not evaluated
 - If a in $a || b$ evaluates to true, b is not evaluated
 - Avoids the Pascal problem, e.g. `while (i <= 10 && a[i] != 0) ...`
 - Useful to increase program efficiency, e.g. `if (unlikely_condition && expensive_condition()) ...`
- Ada conditional and/or uses `then` keyword, e.g.: `cond1 and then cond2`
- Ada, C, and C++ also have regular bit-wise Boolean operators

Assignments

- Fundamental difference between imperative and functional languages
 - Imperative: "computing by means of side effects" i.e. computation is an ordered series of changes to values of variables in memory (state) and statement ordering is influenced by run-time testing values of variables
 - Expressions in functional language are *referentially transparent*:
 - All values used and produced depend on the referencing environment of the expression and *not* on the evaluation time of the expression
 - A function is *idempotent* in a functional language:
 - Always returns the same value given the same arguments because of the absence of side-effects

L-Values, R-Values, Value Model, and Reference Model

- Suppose we have an assignment of the form: $a := b$
 - The left-hand side a of the assignment is a location, called *l-value* which is an expression that should denote a location, e.g. an array element $a[2]$ or a variable $f00$ or a dereferenced pointer $*p$
 - The right-hand side b of the assignment is a value, called *r-value* which can be any syntactically valid expression with a type compatible to the right-hand side
- Languages that adopt *value model* of variables copy values: Ada, Pascal, C, C++ copy the value of b into the location of a
- Languages that adopt *reference model* of variables copy references, not values, resulting in shared data values via multiple references
 - Clu copies the reference of b into a so that a and b refer to the same object
 - Java uses value model for built-in types and reference model for classes

Special Cases of Assignments

- Assignment by variable initialization
 - *Implicit*: e.g. `0` or `NaN` (not a number) is assigned by default
 - *Explicit*: by programmer (more efficient than using an explicit assignment, e.g. `int i=1;` declares i and initializes it to `1` in C)
 - Use of uninitialized variable is source of many problems, sometimes compilers are able to detect this but cannot be detected in general
- Combination of assignment operators
 - In C/C++ `a+=b` is equivalent to `a=a+b` (but `a[i++]+=b` is different from `a[i++]=a[i++] + b`)
 - Compiler produces better code, because the address of a variable is only calculated once
- *Multiway assignments* in Clu, ML, and Perl
 - `a,b := c,d` assigns c to a and d to b *simultaneously*, e.g. `a,b := b,a` swaps a with b
 - `a,b := 1` assigns `1` to both a and b

Structured and Unstructured Flow

- *Unstructured flow*: the use of `goto` statements and *statement labels* to obtain control flow
 - Merit or evil?
 - Generally considered bad, but sometimes useful for jumping out of nested loops and for programming errors and exceptions
 - Java has no `goto` statement
- *Structured flow*:
 - *Sequencing*: the subsequent execution of a list of statements in that order
 - *Selection*: **if-then-else** statements and **switch** or **case**-statements
 - *Iteration*: **for** and **while** loop statements
 - Subroutine calls and *recursion*
 - All of which promotes *structured programming*

Sequencing

- A list of statements in a program text is executed in top-down order
- A *compound statement* is a delimited list of statements
 - A compound statement is a *block* when it includes variable declarations
 - C, C++, and Java use { and } to delimit a block
 - Pascal and Modula use **begin ... end**
 - Ada uses **declare ... begin ... end**
- C, C++, and Java: expressions can be used where statements can appear
- In pure functional languages, sequencing is impossible (and not desired!)

Selection

- Forms of **if-then-else** selection statements:
 - C and C++ EBNF syntax:
if (<expr>) <stmt> [**else** <stmt>]
Condition is integer-valued expression. When it evaluates to 0, the *else-clause* statement is executed otherwise the *then-clause* statement is executed. If more than one statement is used in a clause, grouping with { and } is required
 - Java syntax is like C/C++, but condition is Boolean type
 - Ada syntax allows use of multiple **elsif**'s to define nested conditions:
if <cond> **then**
 <statements>
elsif <cond> **then**
 <statements>
elsif <cond> **then**
 <statements>
 ...
else
 <statements>
end if

Selection (cont'd)

- **Case/switch** statements are different from **if-then-else** statements in that an expression can be tested against multiple constants to select statement(s) in one of the *arms* of the **case** statement:
 - C, C++, and Java syntax:
switch (<expr>)
{ **case** <const>: <statements> **break**;
 case <const>: <statements> **break**;
 ...
 default: <statements>
}
 - **break** is necessary to transfer control at the end of an arm to the end of the **switch** statement
- The use of a switch statement is much more efficient compared to nested **if-then-else** statements

Iteration

- *Enumeration-controlled loops* repeat a collection of statements a number of times, where in each iteration a loop index variable takes the next value of a set of values specified at the beginning of the loop
- *Logically-controlled loops* repeat a collection of statements until some Boolean condition changes value in the loop
 - *Pretest loops* test condition at the begin of each iteration
 - *Posttest loops* test condition at the end of each iteration
 - *Midtest loops* allow structured exits from within loop with exit conditions

Enumeration-Controlled Loops

- Many lessons have been learned from history for a proper design of enumeration-controlled loops

- Fortran-IV:

```
DO 20 i = 1, 10, 2
...
20 CONTINUE
which is defined to be equivalent to
i = 1
20 ...
i = i + 2
IF i.LE.10 GOTO 20
```

- Problems:

- Requires positive constant *loop bounds* (1 and 10) and *step size* (2)
- If loop *index variable* *i* is modified in the *loop body*, the number of iterations is changed compared to the iterations set by the loop bounds
- `GOTO` statements can jump out of the loop and also in the loop omitting the *loop header*
- The value of *i* after the loop is implementation dependent
- The body of the loop will be executed at least once, even when the low bound is higher than the high bound

Enumeration-Controlled Loops (cont'd)

- Fortran-77:

- Same syntax as in Fortran-IV, but many dialects also allow `ENDDO` to terminate loop
- Integer- and real-typed expressions for loop bounds and step size
- Change of *i* in loop body is not allowed
- Change of bounds in loop body does not affect number of iterations which is fixed to $\max(\frac{H-L+S}{S}, 0)$ for low bound *L*, high bound *H*, step size *S*
- Body is not executed when $(H-L+S)/S < 0$
- Terminal value of loop index variable is the most recent value assigned, which is $L + \max(\frac{H-L+S}{S}, 0) * S$
- Can jump out of the loop, but cannot jump inside

- Algol-60 combines logical conditions:

```
for <id> := <forlist> do <stmt>
where the EBNF syntax of <forlist> is
<forlist> -> <enumerator> [, enumerator]*
<enumerator> -> <expr>
| <expr> step <expr> until <expr>
| <expr> while <cond>
```

- Difficult to understand and too many forms that behave the same:

```
for i := 1, 3, 5, 7, 9 do ...
for i := 1 step 2 until 10 do ...
for i := 1, i+2 while i < 10 do ...
```

Enumeration-Controlled Loops (cont'd)

- Pascal has simple design:

- `for <id> := <expr> to <expr> do <stmt>`
- `for <id> := <expr> downto <expr> do <stmt>`
- Can iterate over any discrete type, e.g. integers, chars, elements of a set
- Index variable cannot be assigned and its terminal value is undefined

- Ada `for` loop is much like Pascal's:

```
for <id> in <expr>..<expr> loop
  <statements>
end loop
for <id> in reverse <expr>..<expr> loop
  <statements>
end loop
```

- Index variable has a local scope in loop body, cannot be assigned, and is not accessible outside of the loop

- C, C++, and Java do not have enumeration-controlled loops although the logically-controlled `for` statement can be used to create an enumeration-controlled loop:

```
for (i = 1; i <= n; i++) ...
Iterates i from 1 to n by testing i <= n before each iteration and updating i by 1 after each iteration
```

- Programmer's responsibility to modify, or not to modify, *i* and *n* in loop body
- C++ and Java also allow local scope for index variable, for example

```
for (int i = 1; i <= n; i++) ...
```

Problems With Enumeration-Controlled Loops

- C/C++:

- This C program never terminates:

```
#include <limits.h>
main()
{ int i;
  for (i = 0; i <= INT_MAX; i++)
    ...
}
```

because the value of *i* overflows (`INT_MAX` is the maximum positive value `int` can hold) after the iteration with `i==INT_MAX` and *i* becomes a large negative integer

- In C/C++ it is easy to make a mistake by placing a `;` at the end of a `while` or `for` statement, e.g. the following loop never terminates:

```
i = 0;
while (i < 10);
{ i++; }
```

- Fortran-77

- The C/C++ overflow problem is avoided by calculating the number of iterations in advance
- However, for `REAL` typed index variables an exception is raised when overflow occurs

- Pascal and Ada:

- Can only specify step size 1 and -1
- Pascal and Ada can avoid overflow problem

Logically-Controlled Pretest Loops

- Logically-controlled *pretest* loops test an exit condition *before* each loop iteration
- Not available Fortran-77 (!)
- Pascal:
 - **while** <cond> **do** <stmt>
where the condition is a Boolean expression and the loop will terminate when the condition is false. Multiple statements need to be enclosed in **begin** and **end**
- C, C++:
 - **while** (<expr>) <stmt>
where the loop will terminate when expression evaluates to 0 and multiple statements need to be enclosed in { and }
- Java is like C++, but condition is Boolean expression

Logically-Controlled Posttest Loops

- Logically-controlled *posttest* loops test an exit condition *after* each loop iteration
- Not available in Fortran-77 (!)
- Pascal:
 - **repeat** <stmt> [; <stmt>]* **until** <cond>
where the condition is a Boolean expression and the loop will terminate when the condition is true
- C, C++:
 - **do** <stmt> **while** (<expr>)
where the loop will terminate when the expression evaluates to 0 and multiple statements need to be enclosed in { and }
- Java is like C++, but condition is a Boolean expression

Logically-Controlled Midtest Loops

- Logically-controlled *midtest* loops test exit conditions within the loop
- Ada:
 - **loop**
 <statements>
 exit when <cond>;
 <statements>
 exit when <cond>;
 <statements>
 ...
 end loop
 - Also allows exit of outer loops using labels:
 outer: **loop**
 ...
 for i in 1..n **loop**
 ...
 exit outer **when** cond;
 ...
 end loop;
 end outer **loop**;
- C, C++:
 - Use **break** statement to exit loops
 - Use **continue** to jump to beginning of loop to start next iteration
- Java is like C++, but combines Ada's loop label idea to allow jumps to outer loops

Recursion

- Subroutines which call themselves directly or indirectly to solve a problem, where the problem is typically defined in terms of simpler versions of itself
- Example: to compute the length of a list, remove the first element, calculate the length of the remaining list n , and return $n+1$. If the list is empty, return 0
- Iteration and recursion are equally powerful in theoretical sense: iteration can be expressed by recursion and vice versa
- Recursion can be less efficient, but most compilers for functional languages will optimize recursion and are often able to replace it with iterations
- Recursion is more elegant to use to solve a problem that is naturally recursively defined, such as a tree traversal algorithm

Recursive Problem Formulations

- The GCD function is mathematically defined by

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

- The GCD function in C:

```
int gcd(int a, int b)
{ if (a==b) return a;
  else if (a>b) return gcd(a-b, b);
  else return gcd(a, b-a);
}
```

Tail Recursive Functions

- *Tail recursive* functions are functions in which no computations follow a recursive call in the function
- `gcd` example recursively calls `gcd` without any additional computations after the calls
- A recursive call could in principle reuse the subroutine's frame on the run-time stack and avoid deallocation of old frame and allocation of new frame
- This observation is the key idea to *tail-recursion* optimization in which a compiler replaces recursive calls by jumps to the beginning of the function

- For the `gcd` example, a good compiler will optimize the function into:

```
int gcd(int a, int b)
{ start:
  if (a==b) return a;
  else if (a>b) { a = a-b; goto start; }
  else { b = b-a; goto start; }
}
```

which is just as efficient as the iterative implementation of `gcd`:

```
int gcd(int a, int b)
{ while (a!=b)
  if (a>b) a = a-b;
  else b = b-a;
  return a;
}
```

Continuation-Passing-Style

- Even functions that are not tail-recursive can be optimized by compilers for functional languages by using *continuation-passing style*:
 - With each recursive call an argument is included in the call that is a reference (continuation function) to the remaining work
 - The remaining work will be done by the recursively called function, not after the call, so the function appears to be tail-recursive

Other Recursive Function Optimizations

- Another function optimization that can be applied by hand is to remove the work after the recursive call and include it in some other form as an argument to the recursive call

- For example:

```
typedef int (*int_func)(int);
int summation(int_func f, int low, int high)
{ if (low==high) return f(low)
  else return f(low)+summation(f, low+1, high);
}
```

can be rewritten into the tail-recursive form:

```
int summation(int_func f, int low, int high, int subtotal)
{ if (low==high) return subtotal+f(low)
  else return summation(f, low+1, high, subtotal+f(low));
}
```

- This example in Scheme:

```
(define summation (lambda (f low high)
  (if (= low high) ;condition
      (f low) ;then part
      (+ (f low) (summation f (+ low 1) high))))))
```

else-part

rewritten:

```
(define summation (lambda (f low high subtotal)
  (if (=low high)
      (+ subtotal (f low))
      (summation f (+ low 1) high (+ subtotal (f low))))))
```

Avoid Algorithmically Inferior Recursive Programs

- The Fibonacci numbers are defined by

$$F_n = \begin{cases} 1 & \text{if } n=0 \text{ or } n=1 \\ F_{n-1} + F_{n-2} & \text{otherwise} \end{cases}$$

- When *naively* implemented as a recursive function, it is very inefficient as it takes $O(n^2)$ time:

```
(define fib (lambda (n)
  (cond ((= n 0) 1)
        ((= n 1) 1)
        (#t (+ (fib (- n 1)) (fib (- n 2)))))))
```

- With a helper function, we can do it in $O(n)$ time:

```
(define fib (lambda (n)
  (letrec ((fib-helper (lambda (f1 f2 i)
                        (if (= i n)
                            f2
                            (fib-helper f2 (+ f1 f2) (+ i 1)))))
    (fib-helper 0 1 0))))
```

Exercise 1: Implement the following C code:

```
#include <stdio.h>
main()
{ int x = 0;
  printf("%d %d\n", x + (x = 1), (x += 1) + x);
}
```

Compile with gcc. What values does the program print? Explain why.

Exercise 2: Implement the following C code:

```
#include <stdio.h>
main()
{ float x;
  for (x = 0.0; x <= 10.0; x += 0.1)
    printf("%f\n", x);
}
```

Compile with gcc and run on linprog. The program does not print the value 10.0. Explain why.

Exercise 3: Write a tail-recursive function in Scheme to compute n factorial. Hint: take a close look at the changes to the summation function defined in these notes.