



Informatik II

Languages, Compilers, and Theory

Lecture 7:

Control Flow



Review

- Last Time
 - Scopes
 - Static versus Dynamic
 - Object lifetimes
 - Storage management
 - Static allocations, stacks, and heaps
 - Advanced language features and bindings
 - Implementing static scopes for nested subroutines
 - Implementing subroutine references



What's Left

- Today
 - Control flow
 - Specifying the order in which elements of a high-level language program execute
 - Control flow mechanisms
 - Statements, loops, subroutine calls, recursion, etc.
- Friday
 - Translation of control flow mechanisms to executable code
 - Review for final exam



Control Flow

- **The obvious:** programs do their work by performing computations
- **Control flow** specifies the order in which computations are performed
- Languages provide a wide variety of **control flow mechanisms** which allow the programmer to specify control flow

Control Flow and Control Flow Mechanisms

- Iteration
 - Performs a group of computations repeatedly
 - One of seven categories of control flow mechanisms
- How do languages implement iteration?

Loops: C, C++, Java

```
for(i=0;i<N;i++) {  
    do_something(i);  
}
```

Iterators: Icon, CLU

```
every do_something(0 to N-1)
```

Some languages use other mechanisms to accomplish the same goal

Recursion: ML


```
fun foo i N =  
    if i < N then  
        do_something i  
        foo i + 1 N  
foo 0 N
```



Important Questions

- What are the categories of control flow mechanisms?
- What features do languages provide to implement the control flow mechanisms in a given category?
- How do compilers translate control flow mechanisms to executable code?

Categories of control flow mechanisms




- Sequencing
 - Given a pair of computations, determines which executes first
- Selection
 - Chooses which of several computations is to be executed based on a run-time condition
- Iteration
 - Performs a group of computations repeatedly
- Recursion
 - Allows computations to be defined in terms of themselves
- Procedural abstraction
 - Allows a group of computations to be named, possibly parameterized, and executed wherever the name is referenced
- Concurrency
 - Allows computations to be executed "at the same time"
- Nondeterminacy
 - Ordering among computations is left unspecified



Sequencing

- Given a pair of computations, determines which executes first
- Two types of computations for which we will consider ordering
 - Expressions
 - Computations which produce a value
 - Examples:
 - Binary arithmetic expression: $foo+bar$
 - Do we evaluate the sub-expression foo or bar first before performing the addition?
 - Assignments
 - Computations which change the values of variables
 - Example:
 - $foo = bar + 1;$
 - $bar = foo + 1;$
 - Which of these two assignments do we evaluate first?

What Exactly Is an Expression?



- Computations which produce a value
 - Variable references
 - Fetches the variables value from memory
 - Constant references
 - 1, 2, 3, 'a', 'b', 'c', ...
 - Operator or function applied to a collection of sub-expressions
 - Function calls
 - `foo(a,b+10)`
 - Arithmetic operations
 - `foo + bar`



Expression Evaluation and Side Effects

- Expression evaluation can produce **side effects** in addition to **values**
- Example:
 - `int foo (void) { a = 10; return a; }`
 - `... foo() + 20 ...`
 - Evaluating "foo()+20" yields the value 30 AND as a side effect assigns the value 10 to the global a
- Expressions without side effects are said to be **referentially transparent**
 - Such expressions can be treated as mathematical objects and reasoned about accordingly



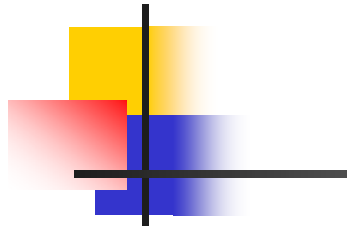
Precedence and associativity

- Two ways of specifying the order in which sub-expressions of complex expressions are evaluated
- Precedence rules
 - Specify how operators "group" in the absence of parentheses
- Associativity
 - Similar to the mathematical concept of the same name
 - Specify whether operators of the same precedence "group" to the right or left



Precedence

- Example:
 - $-a^*c$
 - What is the order of evaluation?
 - $(-a)^*c$ or $-(a^*c)$?
 - In C, $-$ (unary negation) has higher precedence than $*$ (multiplication), so the first parenthization is correct
- Precedence rules vary widely from language to language



Fortran	Pascal	C	Ada
		++, -- (post-inc., dec.)	
**	not	++, -- (pre-inc., dec.), +, - (unary), & (address of), * (contents of), ! (logical not), ~ (bit-wise not)	abs (absolute value), not, **
*, /	*, /, div, mod, and	*(binary), /, % (modulo division)	*, /, mod, rem
+, -	+, - (unary and binary), or	+, - (binary)	+, - (unary)
		<<, >> (left and right bit shift)	+, - (binary), & (concatenation)
.eq., .ne., .lt., .le., .gt., .ge. (comparisons)		<, >, <=, >= (inequality tests)	=, /=, <=, >, >= (comparisons)
.not.		==, != (equality tests)	
		& (bit-wise and)	
		^ (bit-wise exclusive or)	
		(bit-wise inclusive or)	
.and.		&& (logical and)	and, or, xor (logical operators)
.or.		(logical or)	
.eqv., .neqv. (logical comparisons)		?: (if...then...else)	
		=, +=, -=, *=, /=, %= <<=, &=, ^=, = (assignment)	
		, (sequencing)	



Associativity

- More uniform across languages
- Basic arithmetic operators associate left to right
 - Operators are grouped and evaluated left to right
 - Example (subtraction):
 - $9 - 3 - 2$ evaluates as $(9-3)-2$ rather than $9-(3-2)$
- Some arithmetic operators associate right to left
 - Example (exponentiation):
 - $4^{**}3^{**}2$ evaluates as $4^{**}(3^{**}2)$ rather than $(4^{**}3)^{**}2$
- In languages where assignments are permitted in expressions, assignment associates right to left
 - Example:
 - $a = b = a + c$ evaluates as $a = (b = a + c)$
 - $a + c$ is computed and assigned to b , then b is assigned to a



Working With Precedence and Associativity

- Precedence and associativity rules vary **WIDELY** from language to language
 - In Pascal:
 - "if $A < B$ and $C < D$ then..."
 - Could evaluate as "if $A < (B \text{ and } C) < D$ then..."
 - Whoops!
 - **Guideline:**
 - When in doubt, use parentheses, especially if you often switch between languages

More on Expression Evaluation Order

- Precedence and associativity may not always determine an evaluation order
 - Example:
 - $a - f(b) - c * d$
 - Precedence gives us $a - f(b) - (c*d)$ and associativity further gives us $(a-f(b))-(c*d)$
 - Which executes first though: $a-f(b)$ or $(c*d)$?
 - Why do we care?
 - Side effects
 - If $f(b)$ modifies c or d , then the value of the whole expression depends on whether or not $(a-f(b))$ is executed first
 - Code improvement
 - Want to compute $(c*d)$ first so that we don't have to save and restore the computed result before and after the call to $f(b)$
 - Again: when in doubt, use parentheses!

More on Expression Evaluation Order

- Boolean expressions
 - Expressions which perform a logical operation which evaluates to either true or false
 - Examples:
 - `a < b`
 - `a && b || c`
- Boolean expression evaluation can be optimized using **short circuiting**
- Example:
 - `(a < b) && (c < d)`
 - If `(a < b)` evaluates to false there is no need to evaluate `(c < d)`
- Becomes important with code such as:
 - `if (unlikely_condition && expensive_function())...`
 - We would like to short-circuit (prevent) the evaluation of `expensive_function()` if `unlikely_condition` is false



Assignments

- Computation which effects the value of variables
 - Example: $c = a + b$
 - Assigns the value computed by expression "a + b" to the variable c
- Assignments are the fundamental mechanism for producing **side effects**
 - Values assigned to variables may effect future computations
- Assignments are essential to the imperative programming model

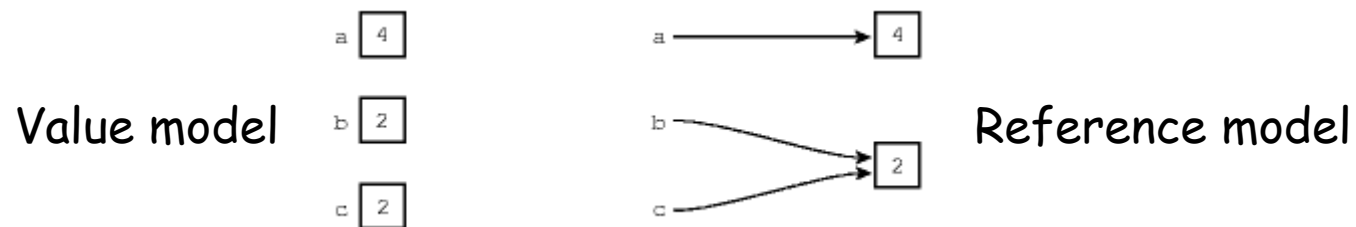


What Exactly Is a Variable?

- Interpretation of a variable name depends on the context in which it occurs
- Example:
 - $d = a$
 - The reference to variable "a" on the **right hand side** of an assignment requires a value
 - Called an **r-value context** and the "a" is called an **r-value**
 - $a = b + c$
 - The reference to variable "a" on the **left hand side** of an assignment refers to a's location
 - Called an **l-value context** and the "a" is called an **l-value**

What Exactly Is a Variable?

- Two ways to treat variables
 - Value model
 - Variables are named containers for values
 - Both l-value and r-value interpretations of variables possible
 - Model used by Pascal, Ada, C, ...
 - Reference model
 - Variables are named reference to values
 - Only l-value interpretation is possible
 - Variables in an r-value context must be "dereferenced" to produce a value
 - Model used by Clu





Assignment Sequencing

- Simple in most languages
 - Assignments are executed in the order they occur in the program text
 - Example:
 - $a = 10; b = a;$ ← First 10 is assigned to a, then a is assigned to b
- Exceptions:
 - Assignment expressions
 - Example: $a = b = a * c;$
 - Recall from expression evaluation order: associate right to left in most programming languages!
 - Initialization
 - Combination assignment operators



Assignment Sequencing

- How do we initialize, or assign initial values to variables?
- Use assignment operator to assign initial values at run-time
 - Problems:
 - Inefficiency: if we know the initial values of variables at compile time, the compiler can assign them to memory without requiring any initial run-time assignments
 - Programming errors: if variables aren't assigned an initial value when declared, the program may use a variable before it contains any value
 - Solutions:
 - Static initializers
 - Example for C: `static char s[] = "foo"`
 - Compiler can assign values to the elements of array `s` at compile time, saving 4 run-time assignments (one for each character plus the NULL terminator)
 - Default values
 - Language can specify a default value for any declaration of built-in types
 - Dynamic value checks
 - Make it a run-time error to use a variable before it has been assigned a value
 - Static value checks
 - Make it a compile-time error if a variable **could** be used before it has been assigned a value



Assignment Sequencing

- Is there a more efficient way of handling complex assignments?
 - Examples:
 - `a = a + 1;`
 - `b.c[3].d = b.c[3].d * e;`
- Problems:
 - Difficult and error-prone to write due to repeated text
 - May lead to inefficient compiled code
- Solution: combination assignment operators
 - Examples:
 - `a += 1;`
 - `b.c[3].d *= e;`
 - Assignments are combined with expression operators
 - Eliminates repeated code
 - Allows the compiler to more easily generate efficient code



Sequencing Wrap-up

- Expressions
 - Precedence and associativity control sequencing
 - Must consider side effects of sub-expressions
 - Logical expressions may benefit from short-circuit evaluation
- Assignments
 - Order of occurrence in program text determines sequencing
 - Compile-time initialization allows run-time initialization assignments to be eliminated
 - Compound assignment operators combine expression evaluation and assignment for the sake of efficiency



Selection

- Chooses which of several computations is to be executed based on a run-time condition
- Most frequently carried out by if...then...else language constructs
 - Example:
 - `if ((a < b) && (c < d)) { do_something(); }`
 - If both the conditional expressions "a<b" and "c<d" evaluate to true, then perform the computation referenced by the sub routine `do_something()`



Selection

- Short circuiting

- Recall: may be used to eliminate unnecessary expression evaluations

- Example:

- `if ((a < b) && (c < d)) { do_something(); }`

- If "a < b" is false then `do_something()` cannot execute regardless of the value of "c < d"

- Short circuiting will generate code so that the evaluation of "c < d" is skipped, along with the execution of `do_something()`, when "a < b" is false



Selection

- Suppose we have code that looks like:

```
j := ... (* something complicated *)  
IF j = 1 THEN  
    clause_A  
ELSIF j IN 2, 7 THEN  
    clause_B  
ELSIF j in 3..5 THEN  
    clause_C  
ELSE  
    clause_D  
END
```
- Problem:
 - Complicated to write



Selection

- Solution: case/switch statements
- Example:

```
CASE ...
```

```
1:          clause_A
```

```
| 2,7:     clause_B
```

```
| 3..5:    clause_C
```

```
ELSE clause_D
```

```
END
```

- Simpler to write and to understand



Iteration

- Performs a group of computations repeatedly
- Languages provide looping constructs to perform iteration
 - Enumeration-controlled loops:
 - Loop is executed once for each value in a finite set (enumeration)
 - Logically-controlled loops:
 - Loop is executed until some Boolean condition changes value
- Note:
 - The imperative style of programming favors loops over recursion
 - Both iteration and recursion can be used to repeatedly execute a group of computations

Enumeration-controlled Loops



- Example from FORTRAN I, II, & IV:
do 10 i = 1,10,2
...
10: continue
 - The variable *i* is called the **index variable** and will take on the values 1, 3, 5, 7, 9
 - Statements between the first and last lines are called the **loop body**, and in this example will execute once for each of the five values of *i*

Enumeration-controlled Loops



- Example from FORTRAN I, II, & IV:
do 10 i = 1,10,2
...
10: continue
- Problems:
 - The body will always be executed at least once
 - Statements inside the loop body can modify *i*, thus changing the behavior of the loop
 - Goto statements can jump into or out of the loop
 - Goto's jumping into the loop without properly initializing *i* will probably result in a run-time error
 - The loop is terminated when the value of *i* **exceeds** the loop upper limit. This could cause an (unnecessary) arithmetic overflow if the upper limit is near the largest value that can be stored in *i*.



Enumeration-controlled loops

- General form of loops:
FOR j := first TO last BY step DO
 ...
END
- Questions to ask about such loops:
 1. Can j, first, and/or last be modified in the loop body? If so, what is the effect on control?
 2. What happens if first is larger than last?
 3. What is the value of j when the loop is finished?
 4. Can control jump into the loop from outside?



Answers

- Can loop indices or bounds be modified?
 - Prohibited by most languages
 - Algol 68, Pascal, Ada, Fortran 77 and 90, and Modula-3
- What happens if “first” is larger than “last”?
 - Most languages evaluate “first” and “last” before the loop executes
 - If “first” is larger than “last” then the loop body never executes
- What is the value of the loop index at the end of the loop?
 - Undefined in some languages, e.g., Fortran IV and Pascal
 - The most recently defined value in other languages
 - Some languages make the index variable a local variable of the loop, thus the variable isn't visible outside the loop's scope
- Can control jump into a loop from the outside?
 - No, for most languages
 - Many languages do allow jumps out of the loop from the inside
 - Many languages provide statements (break, exit, last, etc.) which allow early termination of the loop without requiring a jump



Iterators

- All the loops we've looked at so far iterate over arithmetic sequences (1,3,5,7)
- How do we iterate over arbitrary sets of objects?
- Answer: iterators
 - "FOR j := first TO last BY step DO ... END" loop can be written as "every j := first to last by step do { ... }" in ICON
- Example:
 - "every write (1 + upto(' ', s))" in ICON writes every position in the string s that follows a blank space
 - upto(' ',s) generates every position in s following a blank space
- Iterators are expressions that generate multiple values and which cause their containing expressions and statements to be evaluated or executed multiple times
- Another example:
 - "write(10 + 1 to 20)" will write 10 + j for every j between 1 and 20
 - 1 to 20 generates the sequence 1 through 20



Logically Controlled Loops

- Example:
 - Pre-test loops
 - “while condition do statement”
 - The C for loop is a pre-test, logically controlled loop, not an enumeration-controlled loop
 - for(pre-statement;condition;post-statement) block
 - Post-test loops
 - “repeat statement until condition”
 - “do statement while condition”
 - Midtest loops
 - “loop statements **when condition exit** statements end”



Recursion

- Allows computations to be defined in terms of themselves
 - In other words, complex computations are defined in terms of simpler computations
 - Similar to the principle of mathematical induction
- Preferred method of executing computations repeatedly in functional languages
 - In fact, most of the time, it is the only way of executing computations repeatedly in functional languages
 - Why?
 - Because the alternative, iteration, applies the execution of statements with side effects (assignments) which is prohibited in purely functional languages
 - This really isn't a restriction since recursion is strictly as powerful as iteration and vice versa



Recursion and Iteration

- Informally speaking, loop “FOR j := first TO last BY step DO ... END” using recursion becomes:

```
fun loop j step last =  
  if j <= last then  
    ...  
    loop j + step step last
```

- Problem with recursive emulation of iteration: **slow!**

Improving Performance of Recursion

- Given loop

```
fun loop j step last =  
  if j <= last then
```

```
    ...  
    loop j + step step last
```

- The recursive call to "loop" is the last computation performed by the function "loop"
 - Called **tail recursion**
- Smart compilers will replace tail recursion with a loop
 - Arguments to the tail recursive call are evaluated and placed in the appropriate locals and a jump is made back to the beginning of the function instead of making the recursive call
 - Can result in significant performance increases



Applicative versus Normal-Order Evaluation

- How are arguments to functions evaluated
 - Before they are passed to the function?
 - Called applicative-order evaluation
 - Default in most languages
 - As they are needed by the called function?
 - Called normal-order evaluation
 - Can lead to improved performance if arguments passed to a function are sometimes not used
 - Example:

```
void foo (int a, int b, int c) {  
    if (a + b < N) { return c; } else { return a + b; }  
}
```

 - `foo(a,b,expensive_function())`
 - In some cases "c" is not used, and c's value might be determined by some expensive-to-compute function



Next Time

- Compiling Control Flow
- The compilation process revisited
- Review for the exam