Control Flow

CSE 307 – Principles of Programming Languages

Stony Brook University

http://www.cs.stonybrook.edu/~cse307

Specifying the Semantics of Programs

- ***Operational Semantics***:
  - Give how a program would cause a machine to behave (e.g., the execution of an annotated grammar in imperative parsing with actions)
    - The machine can be abstract, but it is still operational (for example, a machine has unlimited number of registers).
  - Control flow (the order of execution is very important)
- ***Denotational Semantics***:
  - Each phrase in the language is interpreted as a ***denotation***: a conceptual meaning as a mathematical object in a mathematical space
    - For example, denotational semantics of functional languages often translate the language into domain theory (or as functions from states to states)
- ***Axiomatic Semantics***: map the language statements to some logic = their meaning is exactly what can be proven about them in the logic.

Control Flow

- ***Control flow*** is the **ordering** in program execution
- Ordering mechanisms:
  - ***Sequencing***: statements are executed in the order in which they appear in the program (e.g., inside a method in imperative programming),
  - ***Selection / alternation***: a choice is made based on a condition (e.g., *if* and *case switch* statements),
  - ***Iteration***: a fragment of code is to be executed repeatedly either a certain number of times or until a certain run time condition is true (e.g., *for*, *do while* and *repeat* loops)

Control Flow

- ***Procedural Abstraction***: a subroutine is encapsulated in a way that allows it to be treated as a single unit (usually subject to parameterization)
- ***Recursion***: an expression is defined in terms of simpler versions of itself either directly or indirectly (the computational model requires a stack on which to save information about partially evaluated instances of the expression – implemented with subroutines)
- ***Concurrency***: two or more program fragments are to be executed at the same time either in parallel on separate processors or interleaved on a single processor in a way that achieves the same effect.

Control Flow

- ***Exception Handling and Speculation***: if the execution encounters a special exception, then it branches to a handler that executes in place of the remainder of the protected fragment or in place of the entire protected fragment in the case of speculation (for speculation, the language implementation must be able to roll back any visible effects of the protected code)
- ***Nondeterminacy***: the choice among statements is deliberately left unspecified implying that any alternative will lead to correct results (e.g., rule selection in logic programming)
  - ***Backtracking***: implementation/execution mechanism in logic programming

Control Flow

- **Sequencing** is central to imperative languages (von Neumann architecture)
- Logic programming and functional languages make heavy use of **recursion**

Expression Evaluation

- An ***expression*** is a statement which always produces a value
- An expression consists of:
  - simple things: literal or variable
  - functions or expressions applied to expressions
    - ***Operators*** are built-in functions that use a special, simple syntax
      - ***operands*** are the arguments of an operator

Expression Notation

- **Notation**: whether the function name appears before, among, or after its several arguments
  - ***Infix*** operators, e.g., `1 + 2`
  - ***Prefix*** operators, e.g., `(-1)`, `+ 1 2` Polish notation
  - ***Postfix*** operators (reverse Polish), e.g., `1 2 3 * +`
- Most imperative languages use infix notation for binary operators and prefix notation for unary operators.
- Lisp uses prefix notation for all functions, *Cambridge Polish* notation: `(* (+ 1 3) 2)`, `(append a b)`
- Prolog uses the infix notation in the UI: *X is 1+2* and the prefix notation internally (e.g., *is(X,+(1,2))*)

Precedence and Associativity

- ***Precedence and Associativity*: when written in infix notation, without parentheses, the operators lead to ambiguity as to what is an operand of what.**
  - E.g., a + b * c**d**e/f
    - Answer: a + ((b * (c**(d**e)))/f)
    - Neither ((((a + b) * c)**d)**e)/f nor a + (((b * c)**d)**(e/f))

** is right-associative

- Precedence rules specify that certain operators, in the absence of parentheses, group "more tightly" than other operators.
  - E.g., multiplication and division group more tightly than addition and subtraction: 2+3*4 = 2 + 12 = 14 and not 20.
- Bad precedence: the *and* operator in Pascal is higher than <
  - 1<2 and 3<4 is a static compiler error

Pascal	C
	++, -- (post-inc., dec.)
not	++, -- (pre-inc., dec.), +, - (unary), &, * (address, contents of), !, ~ (logical, bit-wise not)
*, /, div, mod, and	* (binary), /, % (modulo division)
+, - (unary and binary), or	+, - (binary)
	<<, >> (left and right bit shift)
<, <=, >, >=, =, <>, IN	<, <=, >, >= (inequality tests)
	==, != (equality tests)
	& (bit-wise and)
	^ (bit-wise exclusive or)
	| (bit-wise inclusive or)
	&& (logical and)
	|| (logical or)
	?: (if . . . then . . . else)
	=, +=, -=, *=, /=, %=, >>=, <<=, &=, ^=, |= (assignment)

Operator precedence levels in C and Pascal: the operators at the top of the figure group most tightly.

Precedence and Associativity

- Precedence:
  - In the United States, the acronym ***PEMDAS*** is common:
    - Parentheses, Exponents, Multiplication, Division, Addition, Subtraction
    - PEMDAS is often expanded to the mnemonic "*Please Excuse My Dear Aunt Sally*"
  - Canada and New Zealand use ***BEDMAS***, standing for Brackets, Exponents, Division, Multiplication, Addition, Subtraction.
  - In the UK, India and Australia is ***BODMAS***: Brackets, Of Order, Division, Multiplication, Addition and Subtraction.

Precedence and Associativity

- ***Associativity rules*** specify whether sequences of operators of equal precedence group to the right or to the left:
  - Summation associate left-to-right, so 9 - 3 - 2 is (9–3)–2=4 and not 8.
  - The exponentiation operator (**) follows standard mathematical convention, and associates right-to-left, so 4**3**2 is 4**(3**2)=262,144 and not (4**3)**2 = 4,096.
- Most expressions are ***left associative***
- The assignment operation is ***right associative***
  - x = y = z will assign the value of z to y and then also to x:

    x = (y = z)
- The power operator is also right associative
- **Rule 0:** inviolability of parentheses!!! That is, developers put expressions into parenthesis to make sure what is the semantics.

Execution Ordering

- Execution ordering is not necessarily defined:
  - In (1<2 and 3>4), which is evaluated first?
  - Some languages define order left to right, some allow re-order:
    - E.g., query optimization in databases is re-ordering!
  - Re-order can increase speed, exploit math identities
  - Re-order can reduce precision, have side-effects
- Optimization by Applying Mathematical Identities:
  - By using the *common subexpression* in the equations.

```
a = b + c                    a = b/c/d        (/ is left-associative)
d = c + e + b                e = f/d/c

Is optimized to:             Is optimized to:
a = b + c                    t = c * d
d = a + e                    a = b/t ,   e = f/t
```

Expression Evaluation

- ***Short-circuiting***:
  - Consider (a < b) && (b < c):
    - If a >= b there is no point evaluating whether b < c because (a < b) && (b < c) is automatically false.
  - Most operators are ***short-circuiting***
  - However, some languages have 2 operators, e.g., Java has the && and & (an operator which evaluates both expressions):
    - (false) & (i++<0) will have the side-effect of incrementing i
- Short-circuiting is also useful in *guards*:

```
if (b != 0 && a/b == c) ...
if (*p && p->foo) ...
```

Expression Evaluation

- ***Assignments*:** in imperative languages, computation typically consists of an ordered series of changes to the values of variables in memory

Expression Evaluation

- **References and Values:**

```
d = a;  // the right-hand side of the assignment refers to
        // the  value of a, which we wish to place into d
a = b + c; // the left-hand side refers to the location of a,
        // where we want to put the sum of b and c
```

- An assignment is a statement that takes pair of arguments: a value (called *r-value*) and a *reference* to a variable into which the value should be placed (called *l-value*)

Expression Evaluation

- Example:
  - Variable a contains 100
  - Variable b contains 200

```
a = b;
```

    - a and b are expressions:
      - b - evaluated for r-value
      - a - evaluated for l-value (location)
  - The value is placed into the location!

Expression Evaluation

- ***Value model*** for variables: variables are names/aliases of locations in memory that contain directly a value
- ***Reference model***: variables are aliases to locations in memory that contain an address where the value is on the heap
- The ***value semantics*** versus ***reference semantics***:
  - the variables refer to values
  - the variables refer to objects
- **Java has both**:
  - built-in types are values in variables,
  - user-defined types are objects and variables are references.
    - When a variable appears in a context that expects an r-value, it must be dereferenced to obtain the value to which it refers
      - In most languages, the dereference is implicit and automatic

Expression Evaluation

- Early versions of Java (2) required the programmer to "***wrap***" objects of built-in types:

```
import java.util.Hashtable;
...
Hashtable ht = new Hashtable();
...
Integer N = new Integer(13);        // Integer is a "wrapper" class
ht.put(N, new Integer(31));
Integer M = (Integer) ht.get(N);
int m = M.intValue();
```

  - The wrapper class was needed here because **`Hashtable`** expects a parameter of a class derived from **`Object`**, and an **`int`** is not an **`Object`**.
  - Recent versions of Java (5) perform automatic ***boxing*** and ***unboxing*** operations: the compiler creates hidden **`Integer`** objects to hold the values and it returns an **`int`** when needed:

```
ht.put(13, 31);
int m = (Integer) ht.get(13);
```

Expression Evaluation

- Expression-oriented vs. **statement-oriented languages**:
  - expression-oriented (all statements are evaluated to a value):
    - functional languages (Lisp, Scheme, ML)
    - logic programming (everything is evaluated to a boolean value: **`true`**, **`false`** or **`undefined`**/unknown in XSB Prolog).
  - statement-oriented: some statements do not return anything
    - most imperative languages (e.g., **`print`** method returns **`void`**)
  - C is halfway in-between (some statements return values)
    - allows expressions to appear instead of statements and vice-versa:

```
if (a == b) {
    /* do the following if a equals b */

if (a = b) {
    /* assign b into a and then do
       the following if the result is nonzero */
```

      - C lacks a separate Boolean type: accepts an integer
        - if 0 then false, any other value is true.

Expression Evaluation

- Combination Assignment Operators:
  - $a = a + 1$ has the effect to increment **the old value of *a* into *a***
  - However, *A[index_fn(i)] = A[index_fn(i)] + 1* is not safe because the function may have a side effect and different values can be returned by *index_fn(i)*
    - It is safer to write :

      j = index_fn(i); OR *A[index_fn(i)]++;*

      A[j] = A[j] + 1;
- More assignment operators: +=, -=
  - Handy, avoid redundant work (or need for optimization) and perform side effects exactly once.
  - --, ++ in Java or C:
    - Prefix or postfix form (different value returned by the sub-expression)
    - The assignment also returns values!

Expression Evaluation

- Side Effects:
  - often discussed in the context of functions
  - a ***side effect*** is some permanent state change caused by execution of function
    - some noticeable effect of call other than return value
    - in a more general sense, assignment statements provide the ultimate example of side effects
      - they change the value of a variable

Expression Evaluation

- SIDE EFFECTS ARE FUNDAMENTAL TO THE WHOLE VON NEUMANN MODEL OF COMPUTING!
- In (pure) functional, logic, and dataflow languages, there are no such changes
  - These languages are called SINGLE-ASSIGNMENT languages
- Several languages outlaw side effects for functions
  - **easier to prove things about programs**
  - closer to Mathematical intuition
  - easier to optimize
  - (often) easier to understand

Expression Evaluation

- ***Multiway Assignments***: in ML, Perl, Python, and Ruby:

```
a, b = c, d;
```

  - Tuples consisting of multiple l-values and r-values
  - The effect is: `a = c;` `b = d;`
  - The comma operator on the left-hand side produces a tuple of l-values, while to comma operator on the right hand side produces a tuple of r-values.
  - The multiway (tuple) assignment allows us to write things like: `a, b = b, a;` # that swap a and b

which would otherwise require auxiliary variables.

  - Multiway assignment also allows functions to return tuples:

```
a, b, c = foo(d);  # foo returns a tuple of 3 elements
```

Expression Evaluation

- ***Definite Assignment***: the fact that variables used as r-values are initialized can be statically checked by the compiler.
  - Every possible control path to an expression must assign a value to every variable in that expression!

```
int i;
int j = 3;
...
if (j > 0) {
    i = 2;
}
...                             // no assignments to j in here
if (j > 0) {
    System.out.println(i); // error: "i might not have been initialized"
}
```

  - more difficult to check with static semantic rules, but most languages do it statically

Unstructured and Structured Flow

- ***Unstructured Programming / Flow***: (conditional or unconditional) jumps / **`GOTO`** statements (usually in assembly languages)
- ***Structured programming*** is a programming paradigm aimed at improving the clarity, quality, and development time of a computer program by making extensive use of **subroutines, block structures, for and while loops**
  - in contrast to using simple tests and **jumps** such as the **`GOTO`** statement, which could lead to "spaghetti code" that is difficult to follow and maintain

Unstructured and Structured Flow

- Control flow in assembly languages is achieved by means of conditional and unconditional **jumps**
  - ***Unconditional jumps***: GOTO statements

```
10    PRINT "HELLO"
20    GOTO 10
```

    Edsger Dijkstra (ACM Turing Award in 1972):

    "**GOTO *considered harmful***".

  - Problem: GOTO are **not limited** to nested scopes, so it is very hard to limit behavior.
    - It is also very hard/impossible to analyze the behavior of programs with GOTOs.
  - Modern languages hardly allow it.

Structured and Unstructured Flow

- ***Conditional Unstructured Flow***: conditional jumps

JZ	op1	jump if zero
JNZ	op1	jump if not zero
JE	op1 = op2	jump if equal
JNE	op1 != op2	jump if not equal
JG	op1 > op2	jump if greater than
JNG	!(op1 > op2)	jump if not greater than
JGE	op1 >= op2	jump if greater than or equal
JNGE	!(op1 >= op2)	jump if not greater than or equal
JL	op1 < op2	jump if less than
JNL	!(op1 < op2)	jump if not less than
JLE	op1 <= op2	jump if less than or equal
JNLE	!(op1 <= op2)	jump if not less than or equal

Structured and Unstructured Flow

- ***Structured programming***: top-down design (progressive refinement), modularization of code, structured types, imperative algorithm elegantly expressed with only sequencing, selection, iteration or recursion.
  - It still includes some **alternatives** to GOTO, but well defined behaviour:
    - **`return/continue/break`** statements
    - Exceptions
    - Continuations (used in Ruby and Scheme) wrap current scope in an object (requires scopes to be on heap)
      - Calling objects restores scope and location.

Structured and Unstructured Flow

- Continuation in Scheme:

```
(define the-continuation #f)
 (define (test)
   (let ((i 0))
     ; call/cc calls its first function argument, passing a continuation variable the-continuation
     (call/cc (lambda (k) (set! the-continuation k)))
     ;The next time the-continuation is called, we start here.
     (set! i (+ i 1))
     i))
          > (test)
          1
          > (the-continuation)
          2
          > (the-continuation)
          3
          ; stores the current continuation (which will print 4 next) away
          > (define another-continuation the-continuation)
          > (test) ; resets the-continuation
          1
          > (the-continuation)
          2
          > (another-continuation) ; uses the previously stored continuation
          4
```

Sequencing

- In a block, statements execute in order (top to bottom)
  - Some languages might waive this for optimization:

```
a = foo()
b = bar()
return a + b
```

    - If **`foo`** and **`bar`** do not have side-effects, then the first two instructions can be executed sequentially, OR in reverse order OR even concurrently

Selection

- Selection statement types (in increasing convenience):
  - **`If`**
  - **`If/Else`** - no repeat negating condition.
  - **`If/Elif/Else`** - don't require nesting (keep terminators from piling up at the end of nested **`if`** statements)
  - **`Switch-Case`** statements
    - Can use **array/hash table** to look up where to go to
    - Can be more efficient than having to execute lots of conditions
- **Short-circuit evaluation of statements:**

  ```
  if foo() or bar(): ...
  ```

  - we can short-circuit evaluation: if **`foo()`** is true, **`bar()`** is not called

Target Machine Architecture

- A compiler is simply a translator:
  - It translates programs written in one language into programs written in another lower-level language
  - This second language can be almost anything—some other high-level language, phototypesetting commands, VLSI (chip) layouts—but most of the time it's the machine language for some available computer
    - Just as there are many different programming languages, **there are many different machine languages**, though the latter tend to display considerably less diversity than the former
    - Each machine language corresponds to a different ***processor architecture***

Selection Code Generation

```
if ((A > B) and (C > D)) or (E <> F):
```

... No short-circuit

```
        r1 := A
        r2 := B
        r1 := r1 > r2
        r2 := C
        r3 := D
        r2 := r2 > r3
        r1 := r1 & r2
        r2 := E
        r3 := F
        r2 := r2 != r3
        r1 := r1 | r2
        if r1 = 0  goto L2        (JZ r1, L2)
L1:  then clause  (label not actually used)
        goto L3
L2:  else clause
L3:
```

Short-circuit

```
        r1 := A
        r2 := B
        if r1 <= r2 goto L4  (JLE r1,r2,L4)
        r1 := C
        r2 := D
        if r1 > r2 goto L1    (JG r1,r2,L1)
L4: r1 := E
        r2 := F
        if r1 = r2 goto L2    (JE r1,r2,L2)
L1: then clause
        goto L3
L2: else clause
L3:
```

Java: The unconditional & and | Operators

- Java has short-circuit operators && and ||, but also unconditional operators & and |:
  - If **x** is **1**, what is **x** after this expression?

```
(1 > x) && (1 > x++)                    1
```

  - If **x** is **1**, what is **x** after this expression?

```
(1 > x) & (1 > x++)                     2
```

  - How about?

```
(1 == x) || (1 > x++)?                  1
(1 == x) | (1 > x++)?                   2
```

Java Boolean operators

```
if ((A <= B | C > D) & (E > F | G < H): I
        r1 := A
        r2 := B
        r1 := r1 <= r2
        r2 := C
        r3 := D
        r2 := r2>r3
        r1 := r1 | r2
        r2 := E
        r3 := F
        r2 := r2 > r3
        r3 := G
        r4 := H
        r3 := r3 < r4
        r2 := r2 | r3
        r1 := r1 & r2
        JZ r1, L1                    (if !r1 goto L1)
        (I)
    L1:
```

Java Boolean operators

if ((A <= B || C > D) && (E > F || G < H): I

```
        r1 := A
        r2 := B
        JLE r1, r2, L1          (if r1 <= r2 goto L1)
        r1 := C
        r2 := D
        JLE r1, r2, L3          (if r1<=r2 goto L3)
L1:     r1 := E
        r2 := F
        JG r1, r2, L2           (if r1>r2 goto L2)
        r1 := G
        r2 := H
        JGE r1, r2, L3          (if r1 >= r2 goto L3)
L2:     (I)
L3:
```

Java Boolean operators

if ((A <= B || C > D) && (E > F | G < H): I

```
        r1 := A
        r2 := B
        JLE r1, r2, L1          (if r1 <= r2 goto L1)
        r1 := C
        r2 := D
        JLE r1, r2, L2          (if r1<=r2 goto L2)
L1:     r1 := E
        r2 := F
        r1 := r1 > r2
        r2 := G
        r3 := H
        r2 := r2 < r3
        r1 := r1 | r2
        JZ r1, L2               (if ! r1 goto L2)
        (I)
L2:
```

Selection

```
CASE ... (* potentially complicated expression *) OF
      1: clause A
      | 2, 7: clause B
      | 3..5: clause C
      | 10: clause D
      ELSE clause E
END
```

- Less verbose,
- More efficient than:

```
IF (* potentially complicated expression *) == 1 THEN
      clause A
ELSIF (* potentially complicated expression *) IN 2,7 THEN
      clause B
ELSIF ...
```

Iteration

- Simplest: variants of **while**, controlled by a condition

```
i = 0;
while (i <= 100) {
    ...
    i += 10;
}
```

- **Do**…**while** have condition executed after the block
- **For**-variations: move number through a range:

```
FOR i := 0 to 100 by 10 DO...END // Pascal
```

OR

```
do i = 1, 10, 2  // Fortran
    ...
enddo
```

Iteration

- The modern **`for`**-loop is a variant of while:

```
for(i=first; i <=last; i+=step)…
```

C defines this to be precisely equivalent to

```
i = first;
while (i <= last) {
    ...
    i += step;
}
```

- Recommendation/Requirement for some languages:
  - no changes to bounds within loop

Iteration

- *Code Generation for* for*-Loops:*

```
    r1 := first
    r2 := step
    r3 := last
L1: if r1 > r3 goto L2
    . . . - - loop body; use r1 for i
    r1 := r1 + r2
    goto L1
L2:
```

Is this efficient?

Iteration

- ***Code Generation for*** for-***Loops:***

```
        r1 := first
        r2 := step
        r3 := last
        goto L2
L1:  . . . – – loop body; use r1 for i
        r1 := r1 + r2
L2:  if r1 ≤ r3 goto L1
```

  - Faster implementation because each of the iteration's contains a single conditional branch, rather than a conditional branch at the top and an unconditional jump at the bottom.

Iteration

- Iterator: pull values from the iterator object

```
for i in range(0, 101, 10):  # Python
    ...
```

  - User can usefully define his own iterator object which makes it possible to iterate over other things:

```
for (Iterator<Integer> it =
        myTree.iterator(); it.hasNext();) {
    Integer i = it.next();
    System.out.println(i);
}
```

- **changes to loop variable within loop are not recommended/allowed**

Iteration

- *Post-test Loops:*

```
repeat
    readln(line);
until line[1] = '$';
```

instead of

```
readln(line);
while line[1] <> '$' do
    readln(line);
```

- Post-test loop whose condition works "the other direction":

```
do {
    readln(line);
} while (line[0] != '$');
```

Iteration

- ***Midtest Loops:***
  - Iteration often allows us to escape the block:
    - **continue**
    - **break**

```
for (;;) {
  readln(line);
  if (all_blanks(line)) break;
  consume_line(line);
}
```

Recursion

- Recursion:
  - equally powerful to iteration
  - mechanical transformations back and forth
  - **often more intuitive** (sometimes less)
  - naive implementation is less efficient than iteration:
    - **Stack frame allocations at every step**: copying values is slower than updates in iterations
  - advantages of recursion:
    - fundamental to functional languages like Scheme
    - no special syntax required

Recursion

- Example:

```
int gcd(int a, int b) { // assume a,b > 0
    if (a == b)
       return a;
    if (a > b)
       return gcd(a-b, b);
    else
       return gcd(a, b - a);
}
```

- Instead of iteration:

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

Recursion

- *Tail recursion*:
  - No computation follows recursive call:

```
def gcd(a, b):
    if a == b:
       return a
    if a > b:
       return gcd(a-b, b)
    else:
       return gcd(a, b - a)
```

  - When the result is a call to same function, **reuse space**

```
def gcd(a, b):
  start:
    if a == b:
       return a
    if a > b:
       a = a - b
       goto start
    else:
       b = b - a
       goto start
```

Recursion

- Tail-recursion:
  - Dynamically allocated stack space is unnecessary: the compiler can *reuse* the space belonging to the current iteration when it makes the recursive call (many compilers do it)

Applicative- and Normal-Order Evaluation

- We have assumed implicitly that arguments are evaluated before passing them to a subroutine
  - This need not be the case
  - It is possible to pass a representation of the unevaluated arguments to the subroutine instead, and to evaluate them only when/(if) the value is actually needed.
- The former option (evaluating before the call) is known as *applicative-order evaluation*
- The latter (evaluating only when the value is actually needed) is known as ***normal-order evaluation***

Normal-Order Evaluation

- *Lazy evaluation*: in the absence of side effects, expression evaluation is delayed until the value is needed
  - A *delayed expression* is sometimes called a *promise*
- *Memoization*: the implementation keeps track of which expressions have already been evaluated, so it can reuse their values if they are needed more than once in a given referencing environment.