Control Flow

- Imperative Languages (C/C++/Java)
  - Execution proceeds according to a sequence of statements
  - Statements are executed for their side effects – they do not return a value
  - Many statements are expressions that are evaluated
    - In a functional language, everything is an expression
    - C/C++ are expression oriented – more expressions than statements
    - Expressions may have side effects in an imperative language, so order of evaluation is important

Structured Control Flow

- Basic flow of control is a transfer to a statement
  - Branching allows out of sequence execution
  - Blocks to be executed (single entry, single exit)
- Modern languages use structured control
  - No arbitrary goto’s (unnecessary, considered bad form)
  - Structured selection (if-else, switch)
  - Structured loops (optimization issues)
    - break, continue permit early exit, early iteration
  - Procedure/function call transfers to structured blocks
    - Argument passing, return values, use of stack
Exceptions

- Late addition to C++, designed in Java from start
- Allow handling of error situations without need for many flags
- Solve problem of single return value used for computed value as well as success indicator
- Exceptions make control flow less obvious
  - But at least very predictable
- Add significant complexity to code generation
  - Especially handling of destructors, reference counts
  - Add performance overhead

Expressions

- The basic building block of C/C++ and Java
- Many operators – unary, binary, and ternary
- Expressions group according to
  - Operator precedence
  - Associativity rules
  - Possible evaluation order rules (but only for a few operators)
- These may determine a partial evaluation order
- But may not determine evaluation order completely
  - Order may be left explicitly undefined by language spec
Expression Evaluation

- Left to right or right to left operand evaluation?
  - In C/C++, undefined except for &&, ||, comma, and ?: 
  - Left to right for these, with short circuiting for && and || 
  - Only one of 2nd and 3rd in ?: is evaluated 
  - Comma is a sequencing operator (not in Java) 
  - Java evaluates all operands left to right 
- Why not specify order? 
  - Compiler implementer should be free to use machine features, optimize subexpression evaluation, etc.

Special Operators

- C/C++/Java increment/decrement operators: ++  --
- Operators with a side effect, equivalent to effect of 
  \[ x = x + 1 \text{ or } x = x - 1 \]
- Prefix and postfix notation
  - Affects value that expression evaluates to be (side effect happens in any case) 
  - Prefix – expression value is after increment/decrement (e.g., the value is pre-incremented) 
  - Postfix – expression value is original (e.g., the target is post-incremented - after determining the value for the expression)
Functional Notation

- Operator expressions are really a syntactic convenience for function notation
  - \( \text{add}(x, \text{mult}(y,z)) \) would be the functional way to write \( x+y*z \) if we had functions named \text{add}, \text{mult}
  - \textbf{Applicative evaluation order} means arguments evaluated before call
  - But order of operand evaluation is still not fixed – arguments to function could be evaluated left to right or right to left
- Can think of \( + \) as a built in function with special infix syntax (syntactic sugar)

Procedures

- Re-entrant blocks of code
- Originally used to divide program into manageable chunks
- Real functions have return values – are expressions, not statements
- Fundamental idea of procedure call is to create an environment for the execution of the procedure
  - Parameters to function
  - Local variables of function
  - Return value
  - Control information
Activation Frame

- local variables (automatic)
- function arguments
- return value
- control link
- access link

Stack

Function Characteristics

- Procedure name
- Formal parameters (names and type)
- Return type
- Body
- May have side effects (change static data)
- Pure functions (ML) do not have side effects
  - Only depend on values passed
  - Independent of where, when called
Calling Semantics

Call By Value

- Caller’s arguments are unaffected
- Argument values are **copied** to activation frame
- Safer – hides implementation, fewer surprises
- Default calling semantics of C and C++
- Calling semantics for primitive types in Java
  - Objects handled a bit differently

Call by Value Stack Trace

```c
int max(int n1, int n2) {
    if (n1 > n2)
        return n1;
    else
        return n2;
}

int max3(int n1, int n2, int n3) {
    return max(max(n1,n2), n3);
}

void main() {
    m = max3(x,y,z);
}
```
Calling Semantics

Call By Reference
- Caller’s arguments are made available to function
- Argument values are not copied
- Function may change the caller’s data
- Can simulate in C by using pointers (which are themselves passed by value)
- C++ allows real call by reference with & syntax
- Passing Java objects behaves like call by reference, almost (what happens if you assign to a passed object?)

Call By Value Result
- Copies of caller’s arguments are made
- Function executes with copies
- Final values of copies is copied back over caller’s values before return
- A combination of call-by-value and call-by-reference
- Can simulate in C++, C
### Calling Semantics

#### Call By Name
- No evaluation of arguments at call
- Actual name of caller’s argument value is passed (with possible renaming of function variables to avoid collisions)
- May have multiple side effects
- Delayed evaluation, may have no evaluation if formal parameter not used
- Much like simple textual substitution

#### Calling Semantics Example

```c
int i;
int a[10];
void inc(int x) {
    i++;
    x++;
}

void main() {
    i = 1;
    a[1] = 1; a[2] = 2;
    inc(a[i]);
}
```

- `a[1]` is incremented if call by reference
- `a[2]` is incremented if call by name
- `a` is unchanged if call by value
### Example

```c
void foo(int a, int b) {
    int tempa = a;
    int tempb = b;
    a = tempa-tempb;
    b = tempa+tempb;
}

void main() {
    int a = 7, b = 13;
    int c;
    foo(b, a);
    c = a*b;
}
```

### Call-by-value

<table>
<thead>
<tr>
<th></th>
<th>before foo</th>
<th>during foo</th>
<th>after foo</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>foo</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>local var tempa</td>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>local var tempb</td>
<td>13</td>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>parameter b</td>
<td>7</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>parameter a</td>
<td>13</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>return value</td>
<td>none</td>
<td></td>
<td></td>
</tr>
<tr>
<td>access link</td>
<td>main</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>main</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>local var c</td>
<td>91</td>
<td></td>
<td></td>
</tr>
<tr>
<td>local var b</td>
<td>13</td>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>local var a</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>parameters</td>
<td>none</td>
<td></td>
<td></td>
</tr>
<tr>
<td>return value</td>
<td>void</td>
<td></td>
<td></td>
</tr>
<tr>
<td>access link</td>
<td>–</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Call-by-reference

<table>
<thead>
<tr>
<th></th>
<th>before foo</th>
<th>during foo</th>
<th>after foo</th>
</tr>
</thead>
<tbody>
<tr>
<td>foo</td>
<td>local var tempb</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>local var tempa</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td></td>
<td>parameter b</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>parameter a</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>return value</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td></td>
<td>access link</td>
<td>main</td>
<td></td>
</tr>
<tr>
<td>main</td>
<td>local var c</td>
<td>120</td>
<td></td>
</tr>
<tr>
<td></td>
<td>local var b</td>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>local var a</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>parameters</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td></td>
<td>return value</td>
<td>void</td>
<td></td>
</tr>
<tr>
<td></td>
<td>access link</td>
<td>–</td>
<td></td>
</tr>
</tbody>
</table>