Dictionary Moment

- In medicine, a side effect is an effect, whether therapeutic or adverse, that is secondary to the one intended; although the term is predominantly employed to describe adverse effects, it can also apply to beneficial, but unintended, consequences of the use of a drug.
Expressions

- precedence and associativity

- **Side Effect**: influences subsequent computation in any way other than by returning a value
- **operator=(target, value)**
  Since target gets changed (and is a parameter), changing the target is considered a side effect of assignment. (I know that seems weird)

Why do we care about side effects (given that they are not a surprise)?

---

Expression Evaluation

- Several languages outlaw side effects for functions
  - easier to prove things about programs
  - closer to Mathematical intuition
  - easier to optimize
  - (often) easier to understand

- But side effects can be nice
  - consider rand() - changes the seed
Expression Evaluation

- Side effects are a particular problem if they affect state used in other parts of the expression in which a function call appears
  - It's nice not to specify an order, because it makes it easier to optimize
    ```
    int a = 10;  int b = 7;
    cout << doit(a+b,a++) << " " << a << " " << b; // 28 11 7
    ```
  - Fortran says it's OK to have side effects
    - they aren't allowed to change other parts of the expression containing the function call
    - Compilers can't check this completely, and most don't at all

Expressions

- In their purest form, expressions do not involve control issues: subexpressions can be evaluated in arbitrary order, and the order does not affect the result. Functional programming tries to achieve this goal for whole programs.
- If expressions (with side effects) could have arbitrary evaluation order, programs would become non-deterministic
- Common subexpression elimination becomes challenging. `x+y` could have changed even though I don’t see changes to `x` or `y`
Strictness

- An evaluation order for expressions is strict (eager) if all subexpressions of an expression are evaluated, whether or not they are needed to determine the value of the result, non-strict (lazy) otherwise.
- Arithmetic is almost always strict.
- Every language has at least a few non-strict expressions (?:, &&, ||).
- A form of strict evaluation called applicative-order is common: "bottom-up" or "inside-out". Expressions are evaluated in the order they are encountered in evaluating the expression tree.

Applicative Order – as needed

Consider: \((a+b)/(c+d)+((e-f)-g)/(h+j)\) – which operation is evaluated first?

- Still leaves open the question: whether left-to-right or not.
Short Circuit Evaluation

- C, C++, and Java: use short-circuit evaluation for the usual Boolean operators (\&\& and ||), but also provide bitwise Boolean operators that are not short circuit (& and |)
- Ada: programmer can specify either (short-circuit is specified with and then and or else)
- Short-circuit evaluation exposes the potential problem of side effects in expressions e.g. \((a > b) || (b++ / 3)\)

Control Flow

- Basic paradigms for specifying ordering
  - Sequencing
  - Selection
  - Iteration
  - Procedural Calls
  - Recursion
  - Concurrency
  - Exception Handling
  - Nondeterminacy
Expression Evaluation

- Infix, prefix operators
- Precedence, associativity
  - C has 15 levels - too many to remember
  - Pascal has 3 levels - too few for good semantics
  - Fortran has 8
  - Ada has 6
    - Ada puts `and` & `or` at same level
    - Lesson: when unsure, use parentheses!

Figure 6.1
Operator precedence levels in Fortran, Pascal, C, and Ada. The operator s at the top of the figure group most tightly.
• In C,
  • what is x = -3 -4

• Referential transparency: a property of pure functional languages - whereby an expression can be replaced by its value without affecting the program.
• Example. If rand() yields 5436, can you replace the call to rand with 5436?
• In other words: “A referentially transparent function is one that, given the same parameter(s), always returns the same result.“ It does not depend on its referencing environment.
Expression Evaluation

- Ordering of operand evaluation
- Inability to represent numbers exactly causes problems in application of identities
  - commutativity (assumed to be safe)
  - associativity (known to be dangerous)

```c
int a = INT_MAX;     2147483647
int b = INT_MIN;   -2147483648
int c = 10;
cout << a << " " << c << " " << a+c << endl; -2147483639
```

Associativity difference

```c
for (int i=0; i < 20000000; i++)
    w-= epsilon;
    cout << "w " << w << endl;
    2.14748e+009
    cout << "w " << w -20000000*epsilon<< endl;
    2.14736e+009
```
Expression Evaluation

• **Short-circuiting**
  – Consider \((a < b) \&\& (b < c)\):
    • If \(a \geq b\) there is no point evaluating whether \(b < c\) because \((a < b) \&\& (b < c)\) is automatically false
  
  – Sometimes the difference can be important…
    ```
    if (b != 0 && a/b == c) ...
    if (*p && p->foo) ...
    if (f || expensive()) ...
    ```

*Expression Evaluation*

• Variables as values vs. variables as references
  – value-oriented languages
    • C, Pascal, Ada
  
  – reference-oriented languages
    • most functional languages (Lisp, Scheme, ML)
    • Clu, Smalltalk
  
  – Java in-between
    • built-in types are values
    • user-defined types are objects - references
**Expression Evaluation**

- **Orthogonality**
  - Features that can be used in any combination
    - Meaning is consistent
      
      \[
      \text{if} \ (\text{if} \ b \neq 0 \ \text{then} \ a/b = c \ \text{else} \ \text{false}) \ \text{then} \ ...
      \]
      
      \[
      \text{if} \ (\text{if} \ f \ \text{then} \ \text{true} \ \text{else} \ \text{messy}()) \ \text{then} \ ...
      \]

---

**Selection**

- **Selection**
  - sequential if statements
    
    \[
    \text{if} \ ... \ \text{then} \ ... \ \text{else}
    \]
    
    \[
    \text{if} \ ... \ \text{then} \ ... \ \text{elsif} \ ... \ \text{else} \ (\text{Easier to read as limited form of nesting})
    \]
    
    \[
    \text{(cond}
    \]
    
    \[
    (C1) \ (E1)
    \]
    
    \[
    (C2) \ (E2)
    \]
    
    \[
    ...
    \]
    
    \[
    (Cn) \ (En)
    \]
    
    \[
    (T) \ (Et) \ (\text{True – default case})
    \]

Copyright © 2009 Elsevier
Selection

- Selection
  - Fortran computed gotos
    - `GO TO (label-list)[,] expr`
  - jump code
    - for selection and logically-controlled loops

Selection

- At the assembly language level, jump is especially useful in the presence of short-circuiting
- **Example** (section 6.4.1 of book):
  ```
  if then
  then_clause
  else
  else_clause
  ```
Selection

• Code generated without short-circuiting (Pascal)

```pascal
r1 := A           -- load
r2 := B           \{(A > B) and (C > D)\} or (E <> F)
r1 := r1 > r2
r2 := C
r3 := D
r2 := r2 > r3    // Reuse of registers - confusing to read
r1 := r1 & r2
r2 := E
r3 := F
r2 := r2 $<>$ r3
r1 := r1 $|$ r2
if r1 = 0 goto L2

L1:    code for then_clause -- label not actually used
    goto L3
L2:    code for else_clause
L3:
```

Copyright © 2009 Elsevier

Selection

• Code generated with short-circuiting (C)

```c
r1 := A    \{(A > B) and (C > D)\} or (E <> F)
r2 := B
if r1 <= r2 goto L4
r1 := C
r2 := D
if r1 > r2 goto L1

L4:      r1 := E
        r2 := F
        if r1 = r2 goto L2
L1:      code for then_clause
        goto L3
L2:      code for else_clause
L3:
```

Copyright © 2009 Elsevier
Order of Evaluation

```c
int doit(int b) {  return b;}
q = 2;
m = 3*(q++)+ 2*(q++) + (q++);
cout << " m2 = " << m<< endl;
q=2;
m = 3*(q++)+ 2*doit(q++) + doit(q++);
cout << " m3 = " << m<< endl;
```

What do you think is produced?

---

Order of Evaluation

```c
int doit(int b) {  return b;}
q = 2;
m = 3*(q++)+ 2*(q++) + (q++);
cout << " m2 = " << m<< endl;
q=2;
m = 3*(q++)+ 2*doit(q++) + doit(q++);
cout << " m3 = " << m<< endl;
```

“Equivalent” code is not equivalent
Resultant answers depend on compiler
12 19  (visual studio)
12 13  (gcc)
sequence point

- A sequence point is a point in time at which the dust has settled and all side effects which have been seen so far are guaranteed to be complete and no side effects from subsequent evaluations have yet been performed.
- The sequence points listed in the C standard are:
  - at the end of the evaluation of a full expression
  - at the ||, &&, ?, and comma operators; and
  - at a function call (after the evaluation of all the arguments, and just before the actual call).
- In the expression f()+g(), it is possible that either f() or g() will be executed first.
- In the code f(),g() the order of evaluation is defined

Example 1
int x = 4;
x = 2 + x++ * 3;
What should x be?

Example 2
xxx = 5;
cout << xxx++ << xxx++ << xxx++ << endl;
What should be output?
Example 1

```c
int x = 4;
x = 2 + x++ * 3;
x = 15 in Visual studio
x = 14 on HPUX (unknown compiler and version)
```

Example 2

```c
xxx = 5;
cout << xxx++ << xxx++ << xxx++ << endl;
```

765 in Visual Studio

Example 3

```c
int xxx = 5;
bool res = xxx++ >= 5 && (xxx++ >= 6);
if (res) cout << "TRUE";
```

What should be output?
Example 3

```cpp
int xxx = 5;
bool res = xxx++ >= 5 && (xxx++ >= 6);
if (res) cout << "TRUE";
```

TRUE with visual studio

---

**Take away**

- Within statements, there is an order of evaluation which (a) may not be clear and (b) may differ between compilers
Iteration

- Enumeration-controlled
  - C++ for loops
    - scope of control variable
    - allow changes to bounds within loop?
    - allow other changes to loop variable within loop?
    - value of control variable after the loop?

```c++
int final = 3;
for (i = 0; i <= final*2; i++)
{
    cout << i++ << " " ;
    final = 4;
}
```

Outputs 0 2 4 6 8 (10) in C++

Recursion

- Recursion
  - equally powerful to iteration
  - mechanical transformations back and forth
  - often more intuitive (sometimes less)
  - naïve implementation less efficient
    - updating call stack is time/memory consuming
    - no special syntax required
    - fundamental to functional languages like Haskell
Recursion

• Tail recursion  No computation follows recursive call

```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 calls are significant because they can be implemented without adding a new activation record to the call stack. Most of the frame of the current AR is not needed any more, and can be replaced by the frame of the tail call, modified as appropriate.

Switch Statements

• Rules
• cases can be constants, constant expressions, constant ranges
• no overlapping cases
• error if unspecified case occurs (or ignore)
• Usually implemented via jump table: vector of unconditional jumps is stored sequentially in memory
• Need one location in jump table for every value between range of possible values
typedef enum
{ADD, MULT, MINUS, DIV, MOD, BAD}
    op_type;

char unparse_symbol(op_type op)
{
    switch (op) {
        case ADD :
            return '+';
        case MULT:
            return '*';
        case MINUS:
            return '-';
        case DIV:
            return '/';
        case MOD:
            return '%';
        case BAD:
            return '?';
    }
}

Switch Statements
• Implementation Options
  – Series of conditionals
    • Good if few cases
    • Slow if many
  – Jump Table
    • Lookup branch target
    • Avoids conditionals
    • Possible when cases are small integer constants
  – Flexible
    • Picks one based on case structure

Jump Table Structure

<table>
<thead>
<tr>
<th>Switch Form</th>
<th>Jump Table</th>
<th>Jump Targets</th>
</tr>
</thead>
<tbody>
<tr>
<td>switch(op)</td>
<td>jtab:</td>
<td>Targ0:</td>
</tr>
<tr>
<td>case 0:</td>
<td>Targ0</td>
<td>Code Block 0</td>
</tr>
<tr>
<td>Block 0</td>
<td>Targ1</td>
<td>Go to N</td>
</tr>
<tr>
<td>case 1:</td>
<td>Targ2</td>
<td>Code Block 1</td>
</tr>
<tr>
<td>Block 1</td>
<td>...</td>
<td>Go to N</td>
</tr>
<tr>
<td>...</td>
<td>Targn-1</td>
<td>Code Block 2</td>
</tr>
<tr>
<td>case n-1:</td>
<td>Code Block n-1</td>
<td></td>
</tr>
<tr>
<td>Block n-1</td>
<td></td>
<td>Go to N</td>
</tr>
</tbody>
</table>

Code Generated
if op > upperbound|op < lowerbound
    go to N
    target = JTab[op];
    goto target;
N:
Jump Table Structure
Could be a space hog

Switch Form

```java
switch(op) {
    case 0:
        Block 0
    case 1000:
        Block 1
    case n-1:
        Block n-1
}
```

Jump Table

```
jtab:
    Targ0
    Targ1
    Targ2
    ...
    Targn-1
```

Jump Targets

```
Targ0: Code Block 0
    Go to N
Targ1: // no content
    Go to N
Targ2: // no content
    Go to N
```

Code Generated

```java
if op > upperbound | op < lowerbound
    go to N

    target = jtab[op];
    goto target;
```

Jump Table Space Costs

- **Jump tables**
  - best for large number of case labels ($\geq 8$)
  - may take a large amount of space if the labels are not well-clustered.
- A jump table with max. and min. case labels $c_{\text{max}}$ and $c_{\text{min}}$ needs $c_{\text{max}} - c_{\text{min}}$ entries.
  This can be wasteful if the entries aren’t “dense enough”, e.g.:
  ```java
  switch (x) {
      case 1: ...
      case 1000: ...
      case 1000000: ...
  }
  ```
- Define the **density** of a set of case labels as
  ```latex
  \text{density} = \text{number of case labels}/(c_{\text{max}} - c_{\text{min}})
  ```
- Compilers will not generate a jump table if density below some threshold (typically, 0.5).
Use of Switch Statements

- if number of case labels is small ($\leq 8$), use linear or binary search on table of case values/jump pairs.
  - use number of case labels to decide between the two.
- if density $\geq$ threshold ($\sim 0.5$) :
  - generate a jump table;
  else :
  - divide the set of case labels into sub-ranges such that each sub-range has density $\geq$ threshold;
  - generate code to use binary search to choose amongst the sub-ranges;

Guarded Commands

Selection: if <boolean> - > <statement> [ ] <boolean> - > <statement> [ ] ... [ ] <boolean> - > <statement> fi

- Semantics: when this construct is reached,
  - Evaluate all boolean expressions
  - If more than one are true, choose one nondeterministically. (In Haskell, first true expression is used.)
  - If none are true, it is a runtime error
Guarded Commands Idea: if the order of evaluation is not important, the program should not specify one

In Haskell, first one that matches is used.

Summary

• Every language has three major program components: expressions, statements, and declarations.

• Expressions are executed for their values (but may have side effects), and may or may not be sequenced.

• Statements are executed solely for their side effects, and they must be sequenced.

• Declarations define names; they can also give values to those names. They may or may not be viewed by a language as executable.