CSE 3302
Programming Languages

Control I
Expressions and Statements

Chengkai Li
Fall 2007

Control

• Control: what gets executed, when, and in what order.

• Abstraction of control:
  – Expression
  – Statement
  – Exception Handling
  – procedures and functions

Expression vs. Statement

• In pure (mathematical) form:
  – Expression:
    • no side effect
    • return a value
  – Statement:
    • side effect
    • no return value
• functional languages aim at achieving this pure form
• No clear-cut in most languages

Expression

• Constructed recursively:
  – Basic expression (literal, identifiers)
  – Operators, functions, special symbols
• Number of operands:
  – unary, binary, ternary operators
• Operator, function: equivalent concepts
  – \((3+4)\times5\) (infix notation)
  – \(\text{mul} (\text{add}(3,4), 5)\) (Ada, prefix notation)
  – \((\ast (+ 3 4) 5)\) (LISP, prefix notation)

Expression and Side Effects

• Side Effects:
  – changes to memory, input/output
  – Side effects can be undesirable
  – But a program without side effects does nothing!
• Expression:
  – No side effect: Order of evaluating subexpressions doesn’t matter (mathematical forms)
  – Side effect: Order matters

Postfix notation

• PostScript:
  • /Courier findfont
  20 scalefont
  setfont
  72 500 moveto
  (Hello world!) show
  showpage

  http://en.wikipedia.org/wiki/PostScript
Applicative Order Evaluation (Strict Evaluation)

- Evaluate the operands first, then apply operators (bottom-up evaluation)
  (subexpressions evaluated, no matter whether they are needed)

\[ \begin{align*}
&+ \\
&3 \quad 4 \quad 5 \\
&6 \quad 7 \\
&\end{align*} \]

- But is 3+4 or 5-6 evaluated first?

Order Matters

C:
```c
int x=1;
int f(void) {
    int x=1;
    return x;
}
main(){
    printf("%d\n", x + f());
    return 0;
}
```

Java:
```java
class example {
    static int x = 1;
    public static int f() {
        return x;
    }
    public static void main(String[] args) {
        System.out.println(x+f());
    }
}
```

Many languages don't specify the order, including C, Java.
- C: usually right-to-left
- Java: always left-to-right, but not suggested to rely on that.

Expected Side Effect

- Assignment (expression, not statement)

\[ x = (y = z) \]  
(right-associative operator)

\[ \text{Why?} \]

- \text{x++}, ++x

\[ \begin{align*}
&\text{int } x=1; \\
&\text{int } f(void) { \\
&\quad \text{return } x++; \\
&\}
&\text{main() { \\
&\quad \text{printf("%d\n", } x + f()); \\
&\text{return } 0; \\
&\}}
\end{align*} \]

Sequence Operator

- (expr1, expr2, ..., exprn)

- \text{- Left to right (this is indeed specified in C)}
- \text{- The return value is exprn}

\[ \begin{align*}
&x=1; \\
&y=2; \\
&x = (x=x+1, y++, x+y); \\
&\text{printf("%d\n", } x); \\
\end{align*} \]

Non-strict evaluation

- Evaluating an expression without necessarily evaluating all the subexpressions.

- short-circuit Boolean expression
- if-expression, case-expression

Short-Circuit Evaluation

- if (false and x) ... if (true or x)...

- \text{- No need to evaluate x, no matter x is true or false}
- \text{- What is it good for?}
  - \text{- if (i <= lastindex and a[i]=x) ...}
  - \text{- if (p != NULL and p->next==q) ...}
  - \text{- if (x /= 0) and then (y/x > 2) then ...}
  - \text{- if (ptr = NULL) or else (ptr.x = 0) then ...}
- \text{- if (x /= 0) and (y/x > 2) then ...}
- \text{- if (ptr = NULL) or (ptr.x = 0) then ...}

\[ \begin{align*}
&\text{if (x /= 0) and then (y/x > 2) then ...} \\
&\text{if (x /= 0) and (y/x > 2) then ...} \\
&\text{if (ptr = NULL) or else (ptr.x = 0) then ...} \\
&\text{if (ptr = NULL) or (ptr.x = 0) then ...}
\end{align*} \]
if-expression

• if (test-exp, then-exp, else-exp)
  ternary operator
  – test-exp is always evaluated first
  – Either then-exp or else-exp are evaluated, not both
    – if e1 then e2 else e3  (ML)
    – e1 ? e2 : e3  (C)

• Different from if-statement?

Normal order evaluation
(lazy evaluation)

• When there is no side-effect:
  Normal order evaluation (Expressions evaluated in mathematical form)
  – Operation evaluated before the operands are evaluated;
  – Operands evaluated only when necessary.

• int double (int x) { return x+x; }
  int square (int x) { return x*x; }

  Normal order evaluation : square(double(2)) = ...
  Normal order evaluation : square(double(2)) = ...

case-expression

• ML:
  case color of
    red => “R” | blue => “B” | green => “G” | _ => “AnyColor”;

What is it good for?

(int !NULL) ? p->next : NULL

int if_exp(bool x, int y, int z){ } if (x) return y; else return z;

if_exp(p!=NULL, p->next, NULL);

Examples

• Call by Name (Algol60)
• Macro

#define swap(a, b) {int t; t = a; a = b; b = t;}

– What are the problems here?

Unhygienic Macros

• Call by Name (Algol60)
• Macro

#define swap(a, b) {int t; t = a; a = b; b = t;}
#define DOUBLE(x) {x+x;}

main() {
  int t=2;
  int s=5;
  swap(s,t);
  }

#define swap(a, b) {int t; t = a; a = b; b = t;}
#define DOUBLE(x) {x+x;}

main() { int a;
  a = get_int()+get_int();
  printf("a=%d
", a); }

#define swap(a, b) {int t; t = a; a = b; b = t;}
#define DOUBLE(x) {x+x;}

main() {
  int a;
  a = get_int()+get_int();
  printf("a=%d
", a); }
Statements

- If-statements, case-(switch-)statements, loops

Exception Handling

CSE 3302 Programming Languages

Control II
Procedures and Environments

Chengkai Li
Fall 2007

Procedures vs. Functions

- Function:
  - no side effect
  - return a value
  - Function call: expression
- Procedure:
  - side effect, executed for it
  - no return value
  - Procedure call: statement
- No clear distinction made in most languages
  - C/C++: void
  - Ada/FORTRAN/Pascal: procedure/function

Syntax

- Terminology:
  - body
  - specification interface
  - Name
  - type of return value
  - parameters (names and types)

\[
\text{int f(int y); //declaration} \\
\text{int f(int y) { //definition} } \\
\text{  int x;} \\
\text{  x=y+1;} \\
\text{  return x;} \\
\text{ } \\
\text{}}
\]
Procedure Call

- Caller:
  ```
  ... int f(int y){
    int x;
    if (y==0) return 0;
    x=y+1;
    return x;
  }
  ```
- Callee:
  ```
  ...                   if (y==0) return 0;
  x=y+1;
  return x;
  ```
- Control transferred from caller to callee, at procedure call
- Transferred back to caller when execution reaches the end of body
- Can return early

Environment

- Environment: binding from names to their attributes
- static(global) area
- stack
- heap
- automatically-allocated spaces (local variables, procedures)
- manually-allocated spaces (unallocated)
- both for dynamic binding
- Environment: binding from names to their attributes
- Environment: binding from names to their attributes

Activation Record

- Activation record: memory allocated for the local objects of a block
  - Entering a block: activation record allocated
  - Exit from inner block to surrounding block: activation record released
- int x; //global
  ```
  {  
    int x,y;
    x = y*10;
    {    
      int i;
      i = x/2;
    } 
  }
  ```
- Activation Record for Nested Blocks

 Activation Record for Procedures

 int x; //global
 void B(void) {  
  int i;
  i = x/2;
 }  
 void A(void) {  
  int x,y;
  x = y*10;
  B();  
  main()} {  
  A();
  return 0;
 }
```c
int x; //global
void B(void) {
    i = x/2;
}
void A(void) {
    int x,y;
    x = y*10;
    B();
    main() {
        A();
        return 0;
    }
}
```

Activation Record for Procedures

Can only access global variables in defining environment

No direct access to the local variables in the calling environment
(Need to communicate through parameters)

• Why not access global variables through parameters as well?