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)*5 (infix notation)
  - mul(add(3,4), 5)
    - "*"("+"(3,4),5) (Ada, prefix notation)
    - (* (+ 3 4) 5) (LISP, prefix notation)

Postfix notation

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

http://en.wikipedia.org/wiki/PostScript

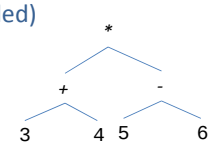
Expression and Side Effects

- 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

Applicative Order Evaluation (Strict Evaluation)

- Evaluate the operands first, then apply operators (bottom-up evaluation)

  (subexpressions evaluated, no matter whether they are needed)



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

Order Matters

C:

```
int x=1;
int f(void) {
   x=x+1;
   return x;
}

main(){
   printf("%d\n", x + f());
   return 0;
}
```

4

Java:

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

3

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)

Why?

- x++, ++x

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


Sequence Operator

- (expr1, expr2, …, expr*n*)
  - Left to right (this is indeed specified in C)
  - The return value is expr*n*

```
x=1;
y=2;
x = (x=x+1, y++, x+y);
printf("%d\n",x);
```


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)…`
  - No need to evaluate x, no matter x is true or false
- What is it good for?
  - `if (i <=lastindex and a[i]>=x)…`
  - `if (p != NULL and p->next==q)…`
- Ada: allow both short-circuit and non short-circuit.
  - `if (x /= 0) and then (y/x > 2) then ...`
  - `if (x /= 0) and (y/x > 2) then ...` ?
  - `if (ptr = null) or else (ptr.x = 0) then ...`
  - `if (ptr = null) or (ptr.x = 0) then ...` ?

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-statemnt?`

case-expression

- ML:

  case color of

  red => "R" |

  blue => "B" |

  green => "G" |

  _ => "AnyColor";

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; }
  ```

  Applicative order evaluation : square(double(2)) = …

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

What is it good for?

```
(p!=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);
```

- With side effect, it may hurt you:

```
int get_int(void) {
  int x;
  scanf("%d" &x);
  return x;
}
```


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;}

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

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

```
#define DOUBLE(x) {x+x;}

main() {
  int a;
  a = DOUBLE(get_int());
  printf("a=%d\n", a);
}
```

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

Statements

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

GOTO

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)

```
int f(int y);    //declaration

int f(int y) {
      int x;
      x=y+1;
      return x;
}
```

```
int f(int y){    //definition
      int x;
      x=y+1;
      return x;
}
```

Procedure Call

- Caller:

```
…
f(a);
…
```

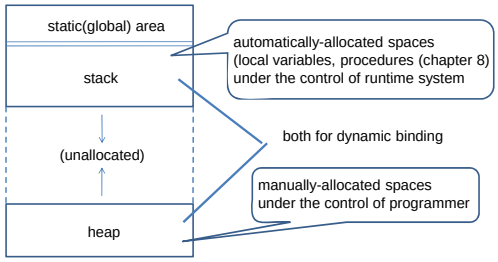
Callee:

```
int f(int y){
  int x;
  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

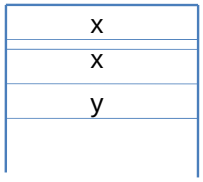
- Environment: binding from names to their attributes



Activation Record for Nested Blocks

- 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
-

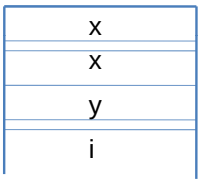
```
int x; //global
{
  int x,y;
  x = y*10;
  {
    int i;
    i = x/2;
  }
}
```



Activation Record for Nested Blocks

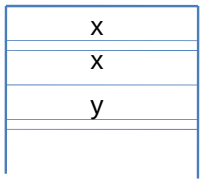
```
int x; //global
{
  int x,y;
  x = y*10;
  {
    int i;
    i = x/2;
  }
}
```

X: Nonlocal variable, in the surrounding activation record



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;
}
```

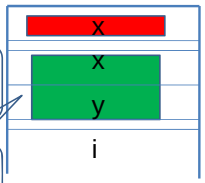


Activation Record for Procedures

```
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;
}
```

x: global variable in defining environment

Need to retain information in calling environment

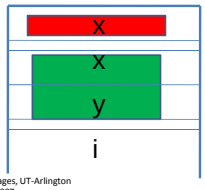


Activation Record for Procedures

```
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;
}
```

Can only access global variables in defining environment

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



Activation Record for Procedures

- Why not access global variables through parameters as well?