

"Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it."
- Brian W. Kernighan

CSE341

Programming Languages

Lecture 6 – November 2019

Expressions

© 2012-2019 Yakup Genç

Chapter 6 slides are adapted from R.W. Sebesta, C. Li & W. He and V. Shmatikov

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

November 2019

CSE341 Lecture 6

3

Expression

- Constructed recursively:
 - Basic expression (literal, identifiers)
 - Operators, functions, special symbols
- Number of operands:
 - unary, binary, ternary operators
- Operator, function: equivalent concepts

<code>(3+4) * 5</code>	(infix notation)
<code>mul (add (3, 4), 5)</code>	
<code>"*" ("+" (3, 4), 5)</code>	(Ada, prefix notation)
<code>(* (+ 3 4) 5)</code>	(LISP, prefix notation)

November 2019

CSE341 Lecture 6

4

Postfix notation

- **PostScript:**

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

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

November 2019

CSE341 Lecture 6

5

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 sub-expressions does not matter (mathematical forms)
- Side effect: Order matters

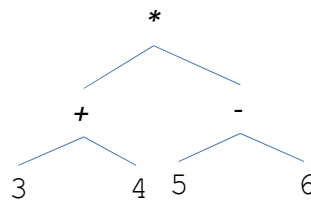
November 2019

CSE341 Lecture 6

6

Applicative Order Evaluation (Strict Evaluation)

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



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

November 2019

CSE341 Lecture 6

7

Order Matters

C:

```

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

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());
    }
}
  
```

4

3

Many languages don't specify the order, including C and Java

- C: usually right-to-left
- Java: always left-to-right, but not suggested to rely on that

November 2019

CSE341 Lecture 6

8

Expected Side Effect

Assignment (**expression in C, not statement**)

`x = (y = z)` (right-associative operator)

x++, ++x

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

Why?

November 2019

CSE341 Lecture 6

9

Sequence Operator

- `(expr1, expr2, ..., exprn)`
 - Left to right (this is indeed specified in C)
 - The return value is `exprn`

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

November 2019

CSE341 Lecture 6

10

Non-strict Evaluation

- Evaluating an expression without necessarily evaluating all the sub-expressions
- short-circuit Boolean expression
- if-expression, case-expression

November 2019

CSE341 Lecture 6

11

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 ...` ?

November 2019

CSE341 Lecture 6

12

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?

November 2019

CSE341 Lecture 6

13

case-expression

- ML:
case color of
 red => "R" |
 blue => "B" |
 green => "G" |
 _ => "AnyColor";

November 2019

CSE341 Lecture 6

14

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)) = ...`

Example

- `int double (int x) { return x+x; }`
- `int square (int x) { return x*x; }`
- Normal order evaluation: `square(double(2)) =`
 - `square(double(2))`
 - `double(2)*double(2)`
 - `(2+2)*(2+2) → (with left to right order) 4*(2+2) → 4*4 → 16`
- Applicative order evaluation: `square(double(2)) =`
 - `square(double(2))`
 - `square(2+2)`
 - `square(4)`
 - `4*4 → 16`

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;
}
```

November 2019

CSE341 Lecture 6

17

Examples

- Call by Name (Algol60)
- Macro

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

– What are the problems here?

November 2019

CSE341 Lecture 6

18

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);
}
```

November 2019

CSE341 Lecture 6

19

Assignments

- Central construct in imperative languages
 - Functional?
- General syntax:


```
<target_var> <assign_operator> <expression>
```
- Operator:


```
"=", "!="
```
- Watch it when overloaded

November 2019

CSE341 Lecture 6

20

Assignment: Conditional Targets

- Conditional targets (as in Perl)

```
($flag ? $count1 : $count2) = 0;
```

equivalent to:

```
if ($flag) {
    $count1 = 0;
} else {
    $count2 = 0;
}
```

- Declaration order?
- Binding?

November 2019

CSE341 Lecture 6

21

Assignment: Compound Operators

- A shorthand method for specifying commonly needed form of assignment

examples:

```
a = a + b
```

can be written as:

```
a += b
```

- Algol, C/C++, JavaScript, Perl, Python, Ruby.

November 2019

CSE341 Lecture 6

22

Unary Assignment

- Unary assignment operators in C-based languages combine increment and decrement with assignment

- Examples:

```
sum = ++count; // count inc. added to sum  
sum = count++; // count inc. added to sum
```

November 2019

CSE341 Lecture 6

23

Assignment as Expressions

- In C-based languages, Perl and JavaScript, assignment statements produce results and can be used as operands

- Examples:

```
while ((ch = getchar()) != EOF) { ... }
```

November 2019

CSE341 Lecture 6

24

Multiple/List Assignments

- Perl and Ruby support list assignments
- Example:

```
($first, $second, $third) = (20, 30, 40);  
($first, $second) = ($second, $third);
```

November 2019

CSE341 Lecture 6

25

Statements

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

November 2019

CSE341 Lecture 6

26