Chapter 6: Expressions

CMSC 124, 1st Semester, AY 2009-10

Chapter 6: Expressions

Definition

Expressions

- Fundamental means of specifying computations in a programming language.

Kinds of Expressions

1. Literals
2. Aggregates
3. Function Calls
4. Conditional Expressions
5. Constants and Variables

Chapter 6: Expressions

Literals and Aggregates

Literals

- Simplest kind of expression.
- Basic element of all other expressions.
- Fixed value of some type.

Eg: Literals in C

100

5.25

'c'

Aggregates

- Construct an aggregate value from its component values.

Eg: Aggregates in ML

{ y = 2009, m = "Sep", d = 26 }

Function Calls

- Basically do computation on arguments passed to the function and produce a result.
- All explicit f'n call has this form:

  f(actual parameters)

- An operator may be thought of as denoting a function.
- Given the expression

  `a + b * c - d`

  It is equivalent to

  `* (+ (a,b),-(c,d))`

Conditional Expressions

Conditional Expressions

- Allow several sub-expressions to be included in the expression but only one of these is chosen to be evaluated.
- Should not be confused with conditional commands.

- **Eg:** In ML (Con Expr)

```
val even :=
 if n mod 2 = 0 then
 true
 else false
```

- **Eg:** In Pascal (Con Cmd)

```
if n mod 2 = 0 then
     even := true
 else
     even := false;
```

Constants and Variables

Consider this: In Pascal, the constant and variable declaration.

```
const a = 5.25;
var b: real;
```

In the expression: **52.35 + a * b**

Replaced by 5.25

Replaced by the value of b

Chapter 6: Expressions

Syntax for Expressions

1. Prefix Notation

- The operator symbol is placed ahead of the operands in left-to-right order.

```
orator(orand1, orand2)
```

- **Eg:**

```
+( *(a,b), -(c,d) )
```

- Also, the notation used for most <u>monadic operations</u>.
  - Negatives: -a
  - Function calls: power(b,n)

Prefix Notation: Other Forms

Cambridge Polish

- Variant of prefix notation.
- Used in LISP, Scheme.
- **Syntax:**

  `(orator orand1 orand2)`
- **Eg:**

  `(+ (* a b) (- c d))`

Polish

- Another variant of prefix notation.
- Does not use parentheses.
- **Syntax:**

  `orator orand1 orand2`
- **Eg:**

  `+ * a b – c d`

Chapter 6: Expressions

Syntax for Expressions

2. Postfix Notation

- The opposite of prefix.
- The operator symbol follows that of the operands.

```
(orand1, orand2)orator
orand1 orand2 orator
```

- **Eg:**

```
((a,b)*,(c,d)-)+
a b * c d - +
```

- It is used extensively as the execution time representation of expressions.
- Since most PL's operate on stacks, eval of expressions becomes very easy.

3. Infix Notation

- Suitable for binary operations.
- The operator is placed between the two operands.
- **Eg:**

```
(a * b) + (c – d)
```

- The notation of mathematics, thus, supported by most PL's.

Chapter 6: Expressions

Infix Notation: Disadvantages

1. **It cannot be used for monadic operations.**

➢ **Solution:** Combine with either prefix or postfix to handle all kinds of expressions.

2. **It is inherently ambiguous.**

➢ **Consider this:**

`a – b / c`

The above expression can be interpreted as:

`(a – b) / c`

`a – (b / c)`

➢ **Solution:** Implement implicit control rules to dictate how expressions are to be evaluated.

Chapter 6: Expressions

Operator Precedence

- The value of an expression is dependent on the order of evaluation of operators involved in the expression.

- **5 + 2 * 3** will give us:
  - 21, if addition is done first.
  - 11, if multiplication is done first.

Chapter 6: Expressions

Operator Precedence

- **In mathematics,** there is the concept of associating operators with priorities.
  - **"Hierarchy of evaluation"**
- This concept of hierarchy has been adopted by PL designers.
- But, remember, each PL have their own operator precedence rules.

Chapter 6: Expressions

Operator Precedence Rules in C

From high priority to low priority the order for all C operators is:

Highest

```
( )  [  ]  -> .
!  - * & sizeof cast ++ --
```
(these are right->left associativity)
```
* / %
+ -
< <= >= >
== !=
&
^          |
&&
||
?:
```
(right->left associativity)
```
= += -=
```
(right->left associativity)
```
,   (comma)
```

Lowest

Chapter 6: Expressions

Operator Associativity

- When 2 adjacent occurrence of operators with the same level of precedence is found, as to which operator evaluated first is answered by **associativity rules**.

Chapter 6: Expressions

Operator Associativity

1. **Left-to-Right Evaluation**

- Prevalent in FORTRAN, C, Pascal, C++, ADA
- **Eg:** 10 – 5 + 4 = 9

2. **Right-to-Left Evaluation**

- In FORTRAN, the evaluation of ** (power) is from right to left.
- **Eg:** 3 ** 2 ** 2 = 81

Operator Associativity

3. Non-Associativity

- **Eg:** In ADA, it is illegal to have an expression like:

```
3 ** 2 ** 2
```

- It is illegal to write a expression having these operators side by side in the expression.

- The only way to have this type of computation is through the use of parentheses.

```
3 ** (2 ** 2) or  (3 ** 2) ** 2
```

Parentheses

- Precedence and associativity rules can be overridden using parentheses.
- **Eg 1:** `10 *  2 + 5 => 10 * (2 + 5)`
  The above example forces addition to be done first.
- **Eg 2 (In Pascal):** `2 ** 3 ** 2 OR 2 ** (3 ** 2)`
  Pascal, by default, employs left-to-right eval.
  The above example forces a right-to-left eval.
- **In APL,** <u>uniform associativity rule</u> applies to all operators.
  Programmer may use parentheses.

QUIZ

Get ¼: Consider the following operator precedence and associativity rules in C

Highest

( ) [ ] -> .
! - * & sizeof cast ++ -- *(right->left assoc)*
* / %
+ -
< <= >= >
== !=
&
^ |
&&
||
?: *(right->left associativity)*
= += -= *(right->left associativity)*
, (comma)

Lowest

Use <u>parentheses</u> to observe the operator precedence and associativity rules in C:

1. a < b && c * d < e
2. a = b = c / d + e

Implementing Expressions

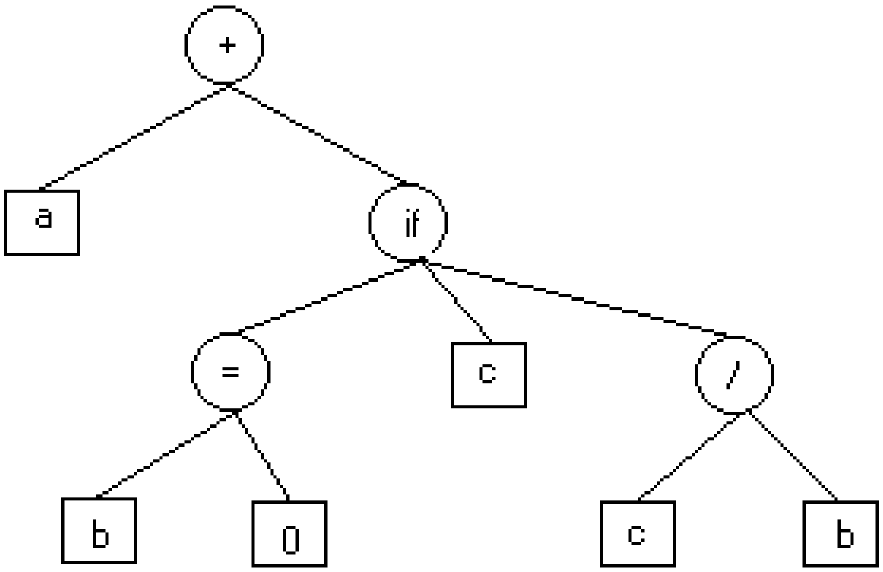
1. Generate the machine code directly.

2. Build the expression tree.

3. Transform the expression from its present notation to either prefix or postfix notations.

1. **Uniform evaluation rules do not apply for evaluating all types of expressions.**
   - ✓ **Consider this**:

```
a + if  b = 0 then c else c/b
```



2. **Side effects are possible.**
   - ✓ Side effect is a form of **ambiguity**.
   - ✓ **Functional side effect** - when the function changes either one of its parameters or global variables.
   - ✓ Given the following:

```
a = 1
f(a) // this function simply adds 1 to a.
```

   - ✓ **What is the output of a + f(a) * a?**
     - 3 – if a is fetched once.
     - 5 – if a is fetched twice, before and after evaluating f(a).

Approaches to Side Effects

1. **Disallow the user to access global variables from a function.**
   - Affects the flexibility and efficiency of programs.

2. **Allow and state explicitly how the evaluation will proceed.**
   - Disallows optimization methods used by compilers.

Implementation Problems

3. **No uniform way of handling error conditions.**
   - ✓ When an error is detected by the system on some primitive operations, control is passed to the operating system.
   - ✓ One common solution is to simply abort the execution without trying to recover.

4. **Short circuit evaluation may be necessary for some expressions.**
   - ✓ **Short circuit evaluation** - result is determined without evaluating all of the operands and/or operators.

     Eg 1:

     (a * 10) * ~~(20 + b / 2)~~

     // if a = 0

     Eg 2:

     a + if b = 0 ~~or c/b > 1~~ then c else d

     // if b = 0

Short Circuit Evaluation

Eg 3:

(In Java) Given that **list** has a **listlen** elements and **key** is the searched for value.

```
index = 1;
while ((index < listlen) && (list[index] <> key))
      index = index + 1;
```

What could happen if evaluation is not short circuit?

Solution: In ADA, the keywords "and then" and "or else".

```
i := 0;
while (i < listlen) and then (list(i) /= 2)
loop
    i := i + 1;
end loop
```

Eg 4:

Short circuit evaluation, can result in serious errors when side effect is also allowed in the language.

```
int x = 10;
int f() {
    x = 100;
}
main() {
    if (a > 0) && ((f()+10)>100)
        x = 0;
    else
        x = 1;
}
```

Translation Considerations

- **Design of the Translator**
  - **Translator 1**: Builds a parse tree and generates object code from the said tree. (optimized)
  - **Translator 2**: Generates object code directly from the source code. (faster implementation)
- **Number of Registers**
  - More registers - the more efficient!

Chapter 6: Expressions

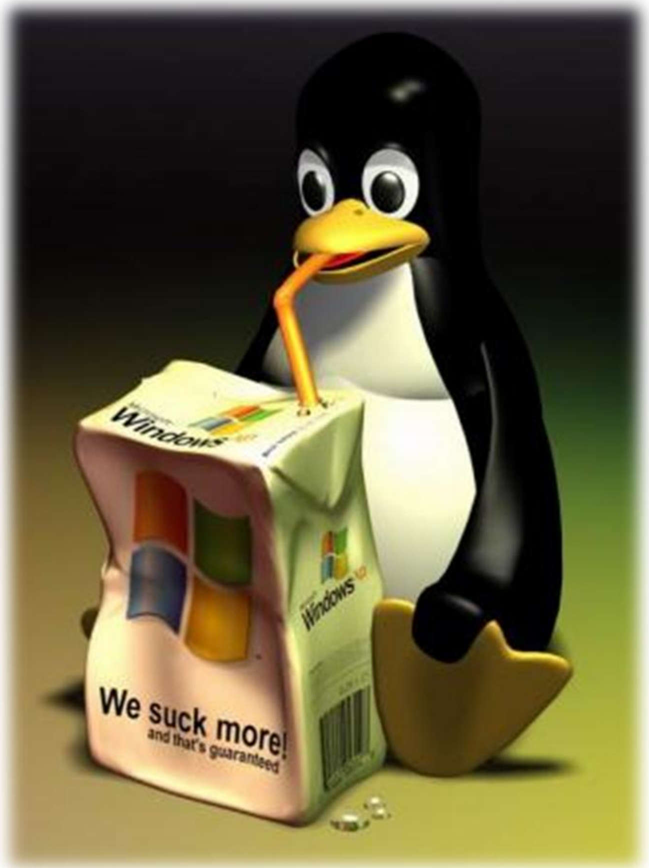
Approaches to Implementation of Expressions

1. Generating Code Directly
2. Generating Code from Postfix Expression
3. Generating Code from Parse Trees

Chapter 6: Expressions

Generating Code Directly

- A compiler generates the target machine code as it scans the source code.
- It reads one token in advance before it proceeds to generate the target machine code for the sub-expression it has already scanned.



Chapter 6: Expressions

Generating Code Directly

Consider this: a + b * c = d + e

Scanned Token	Lookahead	Compiler's Action
a	+	Load into the accumulator the value of a
+	b	Remember +
b	*	Push onto the stack the value a Load into the accumulator the value of b
*	c	Remember * ahead of +

Chapter 6: Expressions

Generating Code Directly

Consider this: a + b * c = d + e

c	=	Push onto the stack the value b Load into the accumulator the value of c Pop from the stack the value b and multiply it with the value c in the accumulator leaving b*c in the accumulator Pop from the stack the value a and add it with value b*c in the accumulator leaving a+b*c in the accumulator
=	d	Remember =

Generating Code Directly

Consider this: a + b * c = d + e

d	+	Push onto the stack the value a+b*c Load into the accumulator the value of d
+	e	Remember + ahead of =
e	end	Push onto the stack the value d Load into the accumulator the value e Pop from the stack the value d and add it with the value e in the accumulator leaving d+e in the accumulator Pop from the stack the value a+b*c and check if it is = with the value d+e in the accumulator leaving the result in the accumulator

Chapter 6: Expressions

Generating Code from Postfix Expression

- The compiler converts an expression to postfix notation.
- The postfix expression is used by the compiler to generate target machine code.

Generating Code from Postfix Expression

Consider this: a + b * c = d + e → ((a,(b,c)*)+, (d,e)+)=

Scanned Token	Lookahead	Compiler's Action
(	(	
(	a	
a	(	Load into the accumulator the value of a
(	b	
b	c	Push onto the stack the value a Load into the accumulator the value of b
c	)	Push onto the stack the value b Load into the accumulator the value of c

Generating Code from Postfix Expression

Consider this: a + b * c = d + e → ((a,(b,c)*)+, (d,e)+)=

)	*	
*	)	Pop from the stack the value b and multiply it with the value c in the accumulator leaving b*c in the accumulator
)	+	
+	(	Pop from the stack the value a and add it with value b*c in the accumulator leaving a+b*c in the accumulator
(	d	
d	e	Push onto the stack the value a+b*c Load into the accumulator the value of d

Generating Code from Postfix Expression

Consider this: a + b * c = d + e → ((a,(b,c)*)+, (d,e)+)=

e	)	Push onto the stack the value d Load into the accumulator the value of e
)	+	
+	)	Pop from the stack the value d and add it with the value e in the accumulator leaving d+e in the accumulator
)	=	
=	end	Pop from the stack the value a+b*c and check if it is = with the value d+e in the accumulator leaving the result in the accumulator

Chapter 6: Expressions

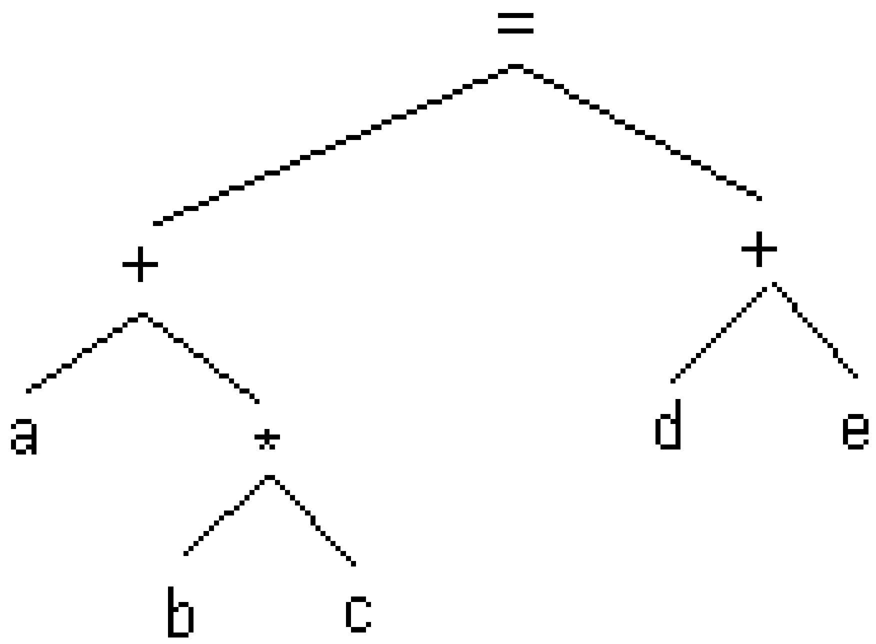
Generating Code from Parse Tree

- A compiler constructs a parse tree from an expression.
- Then, the parse tree is traversed by the compiler generating code while it is traversing the parse tree.

Chapter 6: Expressions

Generating Code from Parse Tree

- **Consider this:** a + b * c = d + e



Chapter 6: Expressions

Generating Code from Parse Tree

Node Visited	Compiler's Action
a	Load into the accumulator the value of a
b	Push onto the stack the value a Load into the accumulator the value b
c	Push onto the stack the value b Load into the accumulator the value c
*	Pop from the stack the value b and multiply it with the value c in the accumulator leaving b*c in the accumulator
+	Pop from the stack the value a and add it with the value b*c in the accumulator leaving a+b*c in the accumulator

Chapter 6: Expressions

Generating Code from Parse Tree

d	Push onto the stack the value a+b*c Load into the accumulator the value d
e	Push onto the stack the value d Load into the accumulator the value e
+	Pop from the stack the value d and add it with the value e in the accumulator leaving d+e in the accumulator
=	Pop from the stack the value a+b*c and check if it is = with the value d+e in the accumulator leaving the result in the accumulator