CMSC 124

Design and Implementation of Programming Languages

CNM Peralta

EXPRESSIONS

Expressions

Fundamental way of specifying computations in a programming language.

Expressions can be classified into **five different categories**.

1.

Literals

Most basic expressions; **fixed** with a **value** of a certain **type**.

Example

```
100 //integer literal
5.25 //floating-point literal
'c' //character literal
```

2.

Aggregates

Value expressions that are made up of component values.

Example

```
[y=2000, m="Jan", d=1]
```

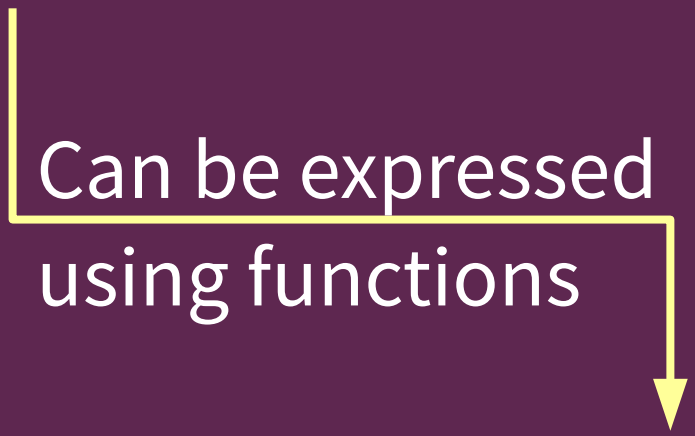
3.

Function Calls

Pass arguments to a **function** that produces a **result**, which is **returned** to the caller.

Example

```
A + B * C - D
```




```
- (+(A, *(B, C)), D)
```

4.

Conditional Expressions

Restricts execution of its subexpressions, **subject to a condition**.

Example

```
if (condition) {
...
} else {
...
}
```

5.

Constants and Variables

Also considered/evaluated as expressions.

Example

```
if (condition) {
...
} else {
...
}
```

The **syntax** of **expressions** can be classified into **three forms**.

1.

Prefix notation

The **operation/function call** is placed **ahead** of its **operands/parameters**; specified from left to right.

Cambridge-Polish Notation

Variant of prefix notation used by LISP; the **function call** is placed **inside** the **parentheses** instead of outside, and commas are removed.

```
swap(&x, &y)
```

Normal Prefix Notation

```
(swap &x &y)
```

Cambridge-Polish Notation

Parentheses can also be **dropped** entirely.

```
swap &x &y
```

Example

The following are examples of prefix notation (normal and both Cambdrige-Polish Forms

```
-  (+(A, *(B, C)), D)

        (-  (+ A (* B C)) D)

              -  +  A  *  B  C  D
```

2.

Infix notation

Most common notation for **binary** expressions; the **operation** is placed **between** the two operands.

Disadvantage

Infix notation **can't represent unary** operators.

&a	???	a&
Prefix	**Infix**	**Postfix**

Thus, infix notation is used in **conjunction** with either **prefix** or **postfix** notation.

Example: C

```
A = B + C; //Infix
B++; //Postfix
++C; //Prefix
```

Disadvantage

On its own, infix notation is **ambiguous**.

Operator precedence and **associativity** need to be **applied implicitly**.

3.

Postfix notation

Places the operator after its operands.

Though **not often used** in **programming languages**, it is the notation **used in executable form**.

Why?

Expressions are **evaluated using stacks**.

Example

```
1 2 + 3 4 - *
```

In infix, this is:

```
(1 + 2) * (3 - 4)
```

There are three **approaches** to **implementing expressions**.

1.

Generate machine code directly from source code.

2.

Build the **expression (parse) tree** first, then **translate the tree** to an **executable** sequence of **instructions**.

If **interpretation** is employed, the **expression tree** can be **directly executed**.

3.

Translate expression to either **prefix** or **postfix** notation and **execute using stacks** or **translate to machine code**.

Implementing expressions have their fair share of **problems**.

1.

Side effects

are possible.

Example: Pascal

```
a = 1;
a + f(a) * a;
```

What if f changes the value of a to 2 (and returns 2 as well)?

Example: Pascal

```
a = 1;                          = 1 + 2 * 1
a + f(a) * a;                   = 3
```

What if f changes the value of a to 2 (and returns 2 as well)?

Example: Pascal

```
a = 1;                          = 1 + 2 * 2
a + f(a) * a;                   = 5
```

What if `f` changes the value of `a` to `2` (and returns 2 as well)?

Solution

Either **disallow side effects** (**difficult**) or include in the **language definition which side effect is legal.**

2.

Error handling when executing expressions, e.g. division by 0, Not-A-Number errors.

When an **error occurs**, execution is usually aborted and **control** is **returned** to the **operating system**.

3.

Short-circuit evaluation

may be **necessary**.

Short-circuit evaluation

Allows **subexpressions** to be **skipped** if they **no longer need to be computed**.

Example: C

```
ptr = head;
while(ptr->next!=NULL &&
      ptr->next->x < x) {
  ptr = ptr->next;
}
```

Problems

What if the subexpression that is **skipped** is **necessary** for **future computations**?

Considerations when translating expressions include:

1.

Translator design

Will a parse tree be used? Or will executable code be generated directly from source code?

2.

Number of registers

The more registers, the more efficient.