Expressions

The simplicities of natural laws arise through the complexities of the language we use for their expression. -- Eugene Wigner

- Arithmetic Expressions 6.2, 6.3
  - precedence
  - side effects
  - conditional expressions
  - operator overloading
- Logical Expressions 6.5, 6.6
  - relational expressions
  - boolean expressions
- Assignment 6.7, 6.8

Arithmetic Expressions

We have seen how to use grammars to describe the *syntax* of some arithmetic expressions, but what is the *semantics* of these expressions? That is, what do these expressions *mean*?

- `x + y × z`
- `f(x) + g(x)`
- `x - -y`
- `[x, y] + [u, v]`
- `4.2 × 10`

Operator Precedence

Operator precedence refers to the order of evaluation of operators in expressions that contain more than one operator.

Operators are usually fall into groups having the same precedence.

- exponentiation
  - `**`
- unary
  - `abs`, `not`, `+`, `-`, `++`, `--`, etc.
- multiplicative
  - `*`, `/`, `div`, `mod`, etc.
- additive
  - `+`, `-`, `or`, etc.

Most languages have groups matching these: exponentiation having the highest precedence, additive operators the lowest.

Associativity...

Operator precedence rules determine the order of evaluation when operators come from different groups (exponentiation, unary, multiplicative, additive). We also need rules to decide the order of evaluation of multiple operators from the *same* group:

- `x + y - z`
  - `(x + y) - z` or `x + (y - z)` *no diff!*
- `x * y / z`
  - `(x * y) / z` or `x * (y / z)` *who cares!*
- `-x++`
  - `-(x++)` or `(-x)++` `-(5++) = -6` `(-5)++ = -4`
- `x ** y ** z`
  - `(x ** y) ** z` or `x ** (y ** z)`

`(2**3)**2 = 64` `2**(3**2) = 512`

More Associativity

Let θ be any binary operator.

- *left associative*
  - $x\ \theta\ y\ \theta\ z = (x\ \theta\ y)\ \theta\ z$
  - normal binary operators in Ada, Pascal, C, etc.
- *right associative*
  - $x\ \theta\ y\ \theta\ z = x\ \theta\ (y\ \theta\ z)$
  - unary increment operators in C++
- *nonassociative*
  - $x\ \theta\ y\ \theta\ z$ -- *illegal!*
  - must specify $(x\ \theta\ y)\ \theta\ z$ or $x\ \theta\ (y\ \theta\ z)$
- *fixed associativity*
  - APL always right to left
  - Smalltalk always left to right

Operand Evaluation

Even with operator precedence rules and associativity rules, the question of order of evaluation is not resolved.

- let θ be some binary operator
- let the operands be α and β
- what is the meaning of the expression: $\alpha\ \theta\ \beta$?
  - evaluate α
  - evaluate β
  - apply θ to α and β

  or is it
  - evaluate β
  - evaluate α
  - apply θ to α and β

What difference does it make?

Side Effects

Consider this harmless little Pascal function:

```
function sq(var x: real): real;
begin
  x := x * x;
  sq := x
end;
```

What is the meaning of these 2 expressions?

- `y + sq(y);`
- `sq(y) + y;`

The function `sq` has “secretly” changed the value of its argument. This is called a *side effect*.

Conditional Expressions

Most programming languages have both unary operators and binary operators that can be used in expressions.

A few programming languages also have a *ternary* operator (an operator that takes *three* operands). This operator is usually a *conditional operator*

Algol 60

```
begin
  integer x, y, z;

  ...
  z := (if y > 0 then x / y else 0);
  ...

end
```

C

```
{
 int x, y, z;

 ...
 z = (y > 0) ? x / y : 0;
 ...

}
```

Operator Overloading

Operator overloading is the practice of using the same symbol for more than one operation.

- Pascal
  - +
    - integer addition, floating-point addition, string concatenation, set union
  - *
    - integer multiplication, floating-point multiplication, set intersection
- C
  - *
    - integer multiplication, floating-point multiplication, pointer dereferencing
  - /
    - integer division, floating-point division

Operator Reference Resolution

If the same symbol is used for more than one operation, there must be some way of resolving which operation the symbol refers to.

- the rule is that all overloaded operators must be resolvable by context (that is, by the known types of the operands)
  - ```
    var x, y, z: integer;
        a, b, c: set of char;
    begin
    ...
    if x = y + z then
    ...
    if a = b + c then
    ...
    if x <= y then
    ...
    if a <= b then
    ...
    ```
    ```

Operator Overloading Gone Awry

Operator overloading often improves readability, writability, learnability, etc. But sometimes it doesn't:

PL/I

```
DECLARE (X, Y, Z) FIXED;

...
IF Y = 0 THEN
   Z = 0
ELSE
   Z = X / Y;
...
```

C

```
{
 int x, y, *p1, *p2;

 ...
 p1 = &x;
 p2 = &y;
 y = x * *p1 & *p2;
 ...
```

C

```
{
 int x, y;
 float z;
 ...
 x = 3;
 y = 2;
 z = x / y;
 z = float(x / y);
 ...
```

Could = be overloaded in C like it is in PL/I?

Programmer-Defined Operator Overloading

Some languages allow the programmer to overload existing operators, effectively *extending* the programming language.

- Ada, C++, Fortran90, etc.

- This is a perfect example of procedural abstraction because it allows us to hide the details of an operation under a simple symbol.
- It is also a good example of data abstraction because it enhances the power of user-defined data types.
  - (more later when we see Abstract Data Types)

Overloading Ada

```
type sport is (baseball, basketball, football, hockey, lacrosse, tennis);
type set_of_sports is array (0..5) of boolean;

noncontact, contact, collision, physical: set_of_sports :=
                                  (false, false, false, false, false, false);

function "+"(s1: in set_of_sports; s: in sport) return set_of_sports is
  s2: set_of_sports := s1;
begin
  s2(sport'pos(s)) := true;
  return s2;
end;
function "+"(s1, s2: in set_of_sports) return set_of_sports is
  s3: set_of_sports;
  i: integer;
begin
  for i in set_of_sports'range loop
    s3(i) := s1(i) or s2(i);
  end loop;
end;

noncontact := noncontact + tennis;
contact := contact + baseball;  contact := contact + basketball;
collision := collision + football;  collision := collision + hockey;
collision := collision + lacrosse;
physical := contact + collision;
```


Relational Expressions

Arithmetic expressions map one or more operand types to a target type; the target type is often the same as one of the operand types.

- `abs`:
  - integer → integer, real → real
- `**`:
  - integer × integer → integer; real × real → real; ...
- `/`:
  - integer × integer → real; ...

Relational expressions map two instances of the same type (or compatible types) to a *boolean*, the result of some *comparison* of the operands.

Relational Operators

There are six common relational operators:

Operation	Operator				Operands
equal	`=`	`==`		`.EQ.`	any type (usually)
not equal	`<>`	`/=`	`!=`	`.NE.`	any type (usually)
less than	`<`			`.LT.`	scalars (+ possibly strings)
less than or equal	`<=`	`=<`		`.LE.`	scalars (+ possibly strings) (+ sets in Pascal/Modula)
greater than	`>`			`.GT.`	scalars (+ possibly strings)
greater than or equal	`>=`	`=>`		`.GE.`	scalars (+ possibly strings) (+ sets in Pascal/Modula)

Scalar types are the numeric types (integers, floating points) and the ordinal types (integers, enumerated types, characters).

Boolean Expressions

Boolean expressions use boolean operators to map boolean operands to a boolean.

- boolean operators
  - `and`, `or`, `not`
- boolean operands
  - boolean variables, boolean constants, expressions that result in booleans

Boolean operator precedence:

- exponentiation
  - `**`
- unary
  - `abs`, **`not`**, `+`, `-`, `++`, `--`, etc.
- multiplicative
  - `*`, `/`, **`and`**, `div`, `mod`, etc.
- additive
  - `+`, `-`, **`or`**, etc.

Operator Oddities

- All of Ada's boolean operators (except `not`) have the same precedence:
  - `X and Y or Z` Illegal
  - `(X and Y) or Z`
    `X and (Y or Z)`
- Pascal's boolean operators have higher precedence than the relational operators:
  - `X < 0 or X > 10` Illegal
  - `(X < 0) or (X > 10)`
- C has no boolean type so 0 is false and any positive or negative number is true:
  - `while(X--)`
  - `if(mystring[len] || counter - 10)`
- C and C++ probably have the richest operator sets:
  - more than 50 operators
  - 17 levels of precedence

Boolean Operand Evaluation

When discussing arithmetic operations, we noted that the order in which the *operands* are evaluated is significant:

- does the expression $\alpha\ \theta\ \beta$ mean
  - evaluate α, evaluate β, apply θ?

  or
  - evaluate β, evaluate α, apply θ?

The same question applies if θ is a binary boolean operator (`and` or `or`). But with the boolean operators we know two extra facts:

- `true or anything = true`
- `false and anything = false`

Short-Circuit Evaluation

If we can determine the value of some expression without evaluating all the operands and operators in the expression, we can do a *short-circuit evaluation*.

- ```
  X := -1;
  if (X > 0) and (Y > 10) then
  ...
  ```

We can also have short-circuited arithmetic expressions:

- ```
  X := 0;
  Z := (X * Y) * (Y + 6);
  ...
  ```

Short-circuiting is not always just for efficiency:

- ```
  X := 0;
  if (X <> 0) and (Y / X > 1) then
  ...
  ```


Operators for Explicit Short-Circuiting

Depending on the language implementation, this expression may evaluate all of `A`, `B`, `C`, or it may stop after the first `false`.

  - `A and B and C`

- There may be times when we want to force evaluation of all operands:
  - `if (sq(X) > 100) and (sq(Y) > 50)`
- There may be times when we want to force short-circuiting:
  - `if (X <> 0) and (Y / X > 1)`
- In Ada:
  - `and`, `or` are *not* short-circuiting
  - `and then`, `or else` *force* short-circuiting
  ```

Assignment Statements

The *assignment statement* is what distinguishes imperative programming from declarative programming; it is how data is stored, how state is recorded and how things change.

The most basic form of assignment is as follows:

- `target` ← `expression`
  (“`target` *gets* `expression`”)
- `expression` and `target` are evaluated in some order
  - `expression` is evaluated to a value
  - `target` is evaluated to an address
- the value of `expression` is stored in the address of `target`

Multiple Targets

The basic form assignment is the *only* form of assignment in many languages (Pascal, Ada, etc.)

Other languages allow more than one target:

- `x, y, z ← 0`
- `target1, target2, ... ← expression`
  (“`target1` *gets* `expression`, `target2` *gets* `expression`, ...”)
- `expression` and `targetlist` are evaluated in some order
  - `expression` is evaluated to a value
  - `target1` is evaluated to an address
  - `target2` is evaluated to an address
  - ...
- the value of `expression` is stored in the address of `target1` and in the address of `target2`, etc.

Conditional Targets

C++ and Java allow the target to be a conditional expression:

- `ctarget` ← `expression`
- `expression` and `ctarget` are evaluated in some order
  - `expression` is evaluated to a value
  - `ctarget` is evaluated to an address
- the value of `expression` is stored in the address of `ctarget`

The difference is in the evaluation of `ctarget`:

- `condition` ? `targetT` : `targetF` = `expression`
- `(x > y) ? big : little = 1;`

Compound Assignment

Compound assignment is simply a notational extension that eliminates a certain redundancy in assignment statements:

- `myvariablewithalongname := myvariablewithalongname + 1;`
- `myCvariablewithalongname = myCvariablewithalongname + 1;`

Faster and less error-prone:

- `myCvariablewithalongname += 1;`
- `myCvariablewithalongname %= 100;`
- `myCmaskwithalongnametoo &= 0x7FFF;`

Even faster for the special case of `+= 1`:

- `myCvariablewithalongname++;`

Assignment Expressions

The examples so far have all been assignment *statements* (standalone commands to be executed one-by-one). Some languages allow assignments to be used as expressions.

The assignment expression can itself be evaluated; the value of the entire assignment expression is the value at the address of `target` after being assigned the value of `expression`.

- `target ← expression`
- `if (target ← expression) > 10 ...`
- `target1 ← target2 ← expression`

Assignment as a Binary Operator

But now we have a problem. What does it mean in C to say:

- `myvar = x = y * 3;`
  - `myvar = (x = y) * 3;`
  - `myvar = x = (y * 3);`
- `index = 13; myarray[index] = index = 42;`
  - `index = 13; myarray[13] = 42; index = 42;`
  - `index = 13; index = 42; myarray[42] = 42;`

In C the assignment operator (=) is treated as any other binary operator. As such, we need precedence rules and associativity rules.

- assignment has very low precedence (lower than additive)
- assignment is right-associative