MASSEY UNIVERSITY
TE KUNENGA KI PŪREHUROA
UNIVERSITY OF NEW ZEALAND

159.341 Programming Languages, Algorithms & Concurrency

Imperative Programming (Part 2)

Daniel Playne
`d.p.playne@massey.ac.nz`

State

Two key parts of imperative programming languages are ***state*** and ***statements***.

The ***state*** of a program is represented by:

- ***Internal State*** - the current state of the program represented by the values of its variables.
- ***External State*** - the values stored in input/output buffers, files etc.

Statements are instructions or commands that change the internal or external state of the program. The state of the output can be observed and is called the ***result*** of the program.

Assignment Statements

Assignment statements modify the value that is currently stored in a variable
`i = (x + y)/2;`

The expression on the r.h.s. is called the ***source*** and the l.h.s. is the ***destination***.

Unlike mathematics (or functional languages) this does not express a relationship but instead is a one-off action.

Statements such as `i = i + 1;` are perfectly legal in imperative languages but would indicate a contradiction in mathematics.

Assignment Syntax

There are many different pieces of syntax used for assignment statements.

C uses = for assignment and == for an equality test while Ada uses := for assignment and = for equality.

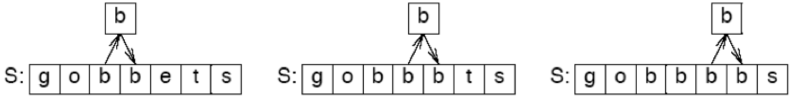
Other languages use additional keywords such as
`let x = x + 1` or `set y = 5`.

The transfer of data in an assignment statement is not performed until the expression on the r.h.s. is completely evaluated (very important for array slicing and expressions with side effects).

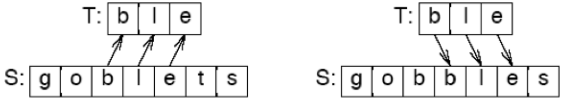
Array Slicing

Ada supports array slicing notation that can be used to read from and assign two subsections of an array.

The incorrect interpretation of the statement `S(4..6) := S(3..5)` is:



The correct interpretation uses a temporary array `T` that is created by the compiler.



Assignment Operators

Many assignment expressions have the form:

```
<dest> := <dest> <op> <exp>
```

For example:

```
I := I + 1;
a[i] = a[i] * 2;
```

Languages often provide shorthand syntax for these types of assignment statements:

```
I +:= 1;
a[i]*= 2;
```

Assignment Operators

In addition to the shorthand assignments, languages may (like C) provide ***post-increment*** `i++;` or ***pre-increment*** `++i;` operations.

Post-increment returns the old value of `i` and increments the value of `i`.
Pre-increment increments the value of `i` and returns the new value.

While these operators may seem odd and provide many different ways to write the same thing, they do aid compiler efficiency and can help produce faster code.

Simultaneous Assignments

Simultaneous assignments allow more than one value to be transferred (effectively) at the same time.

`i,j := 3,5;` Assigns 3 to `i` and 5 to `j`.
`i,j := j,i;` Swaps values of `i` and `j`.

Can be a useful tool but may also introduce some semantic problems:

`a[i], a[j] := 3,5;` What if `i` and `j` are equal?

Expressions

The ***source*** of a value in an assignment statement is an ***expression***.

Generally the expression syntax in most languages is designed to be as close to mathematical notation as possible.

Symbols and syntax are modified to turn notation into machine-readable text files. For example, the mathematical notation a_k becomes `a[k]` etc.

Infix Notation

Many expressions are written using operators in ***infix notation*** - the operators are written in-between the two operands. This notation requires operators to have ***precedence*** and ***associativity*** rules in order to correctly interpret expressions.

For example the expression `4+5*6` is interpreted as `4+(5*6)` because multiplication has a higher precedence than addition. These rules can be overridden using parentheses.

Infix Notation

Most languages will have precedence levels for the following:

Level	Operators	
0	names, literals, parenthesised expressions	
1	`()`, `[]`, `.`	(function calls, subscripting)
2	`+`, `-`	(unary plus and minus)
3	`^`	(exponentiation)
4	`*`, `/`, `mod`, `div`	(standard arithmetic)
5	`+`, `-`	(binary plus and minus)
6	`<`, `>`, `<=`, `=>`, `=`, $\neq$	(comparison)
7	$\neg$	(logical NOT)
8	`&`, |	(logical AND and OR)
9	`:=`	(assignment)

Operator Associativity

Operators will also have a convention for being left- or right-associative (or possible non-associative).

Left-associative operators are evaluated left-to-right. The expression `a-b-c-d` evaluates to `((a-b)-c)-d)`.

Assignment operators are ***right-associative*** so the expression `a:=b:=c:=3` evaluates to `a:=(b:=(c:=3))` (assuming the assignment operator returns the value of the expression).

Operators and Operands

Operators with a single ***operand*** argument are called ***unary*** or ***monadic***. Operators with two ***operands*** are called ***binary*** or ***dyadic***.

Most unary operators precede their operators and are called ***prefix*** operators. ***Postfix*** operators (such as the factorial operator `5!`) follow their operands.

Some languages consider special constants such as ***pi*** and ***e*** as ***zeroadic*** operators (operators with zero operands). The degree of an operator is known as its ***arity*** or ***adicity***.

Conditional Expressions

Some languages provide expressions that include some conditional result:

```
Q := IF X/=0 THEN 1/X ELSE 0;
q = (x != 0) ?  (1/x) :  (0);
```

This is an example of a ***ternary*** operator which has three operands, the condition and two expressions.

Operator Overloading

Traditionally operators have been heavily overloaded even in early languages to support mixed-mode arithmetic. Expressions such as:

`2 * 12.5` and `1.07 * 370.18`

Require different operators and type conversions (int-float vs float-float). Even early languages had *ad-hoc* overloading to support this while more modern languages introduced more support for operators and treat them in a more *orthogonal* manner.

Prefix & Postfix Notation

Computer languages tend to use ***infix*** notation as it is the notation humans are most familiar with. Some use ***prefix*** or ***postfix*** notation instead.

In ***prefix*** notation the operator always comes before the operand(s) and afterwards in ***postfix***. The ***arity*** of each operator determines how many of the following or preceding expressions it is applied to.

Prefix & Postfix Notation

Prefix notation is used extensively in some ***functional*** languages and postfix notation corresponds very closely to the order of machine instructions for evaluating expressions in computers.

Most computers use registers rather than a stack but control languages such as ***Forth*** and ***PostScript*** do use this stack approach.

You may also come across calculators that use postfix or ***reverse Polish notation***.

Lazy Evaluation

Lazy evaluation is the idea that an expression should only be executed if the result of that expression is actually needed. This is most commonly found in ***functional*** languages but does appear in a limited form in imperative languages. In the following statement:

```
IF I <= 10 AND a[I] > 0 THEN ...
```

The second part of the condition `a[I] > 0` only needs to be evaluated if the first part is true (otherwise we already know the condition will be false).

Lazy Evaluation

Some imperative languages support this by providing ***short-cut*** or ***short-circuit*** operators. These perform a similar role to the logical operators in the language but not evaluate operands that will not affect the final result.

In C the operator `&&` and `||` are short-cut operators while `&` and `|` are regular logical operators (they should not be used interchangeably).

External State

Assignment statements use the internal state to modify the internal state of the program.

Output statements (`printf`, `PUT` etc) can be used to modify the external state based on the internal state.

Likewise input statements (`scanf`, `GET` etc) modify the internal state of the program based on the external state.

A program that doesn't interact with the external state is not a very interesting program!

State Representation

Within the program, there is an agreed representation for the state - internal variables etc.

To provide input/output the representation must often be converted to an appropriate form - converting an integer into a string representation etc.

Many languages do not draw a distinction between the input/output and the formatting/conversion functionality.

Input/Output

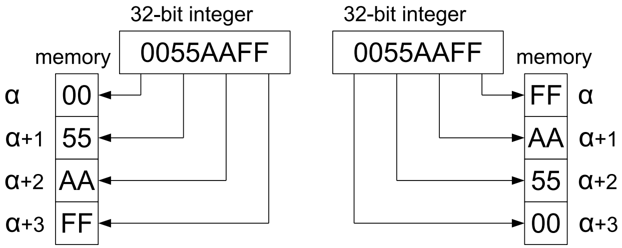
Formatted or character input/output is usually well supported by languages (such as `printf`, `fprintf`, `scanf` etc in C) but do not provide strong type checking. It is the programmers responsibility to provide the correct specifiers for numbers, decimals, characters etc (`%d, %f, %c`).

Modern languages will typically support formatted or unformatted I/O - must decide carefully which to use in our programs.

Binary Input/Output

Binary I/O is when the program reads or writes raw bytes of data that represent the values in memory. Must be very careful as this binary data represents the values on ***our*** machine and ***our*** operating system and may not be at all portable.

Big-endian/***little-endian*** byte conventions may differ on different operating systems.



Binary Input/Output

Persistence is the idea that the program's internal state can persist after the program is halted temporarily by the machine turning off. It turns out to be very hard to provide persistence that is portable across machines and operating systems.

Have been some attempts to use meta-type languages to provide a semi-portable format for widely used primitive types. Related to application checkpointing where a program will save its state at some point that it can recover from following a crash.

Flow Control

Imperative languages allow the programmer to control the order in which statements are executed with flow of control mechanisms. There are generally provided by:

- ***Sequencing*** - jumps, `goto` statements.
- ***Selection*** - conditional branching - `if-else`, `switch` statements.
- ***Repetition*** - loops - `for`, `while`, `repeat` statements.
- ***Routines*** - function and procedure calls.

Sequencing

The statements of an imperative are executed according to the ***program counter*** which is incremented by 1 after each statement. Sequencing statements allow the program counter to be set to a particular value.

These jump or `goto` statements are generally misused and tend to lead to *spaghetti code* (see Edgar Dijkstra: Go To Statement Considered Harmful).

They are used invisibly by the compiler to build other flow-control constructs.

Sequencing

When they are used directly by the programmer they usually take the form of jumping to a labeled instruction in the program.

Sequencing in C and Ada

```c
goto my_label;
...
my_label: x = a + 2;
```

```ada
goto My_Label;
...
<<My_Label>> X := A + 2;
```

The labels and goto statements compile down to program-counter jump instructions. They should generally be avoided and can be considered a relic of machine code (they do still have some very specific uses though).

Selection

Selection (`if`) statements choose one of two sets of instructions to execute next based on a boolean expression (condition).

Most languages allow a list of conditions/sets of instructions to be defined in the same construct (`if-else if-else`). These selection statements can be nested as the programmer sees fit.

Selection

Constructs such as `switch` or `case` statements are generally a special sort of selection that can be compiled to arithmetically computed jumps in the program counter. Often restricted to integer types.

Selection in C and Ada

```
switch(n) {             case N is
case 0:                    when 0 =>
    ...                        ...
    break;
case 1:                    when 1 =>
    ...                        ...
    break;
default:                    when others =>
    ...                        ...
    break;
}                         end case;
```

Selection

When designing `switch` statements, one must consider a number of possibilities:

- Same value appears twice (compile error in C and Ada).
- Value doesn't appear in list (default action).
- Action labelled with more than one value (allowed in C and Ada).
- Actions labelled with a range of values (allowed in Ada but not C).

Repetition

There are two main forms of repetition structures - looping over a predefined number of iterations (`for` loop) or repeating until some condition occurs (`while` loop).

These repetition structures often support controlled exits from within the loops (`break` and `continue`) statements rather than using `goto` statements. It is harder to break out of nested loops.

Repetition

In C, a `for` loop has three parts - initialisation, termination test and increment. Arguably this is really just a `while` loop in disguise as the condition is checked after each iteration and the control variable can be modified within the loop (with caution).

`for` loops in C and Ada

```c
sum = 0;
for(i = 0; i < n; i++) {
    sum += a[i];
}
```

```ada
Sum := 0;
for I in 1..N loop
   Sum := Sum + A(I);
end loop;
```

Most languages use a known range for `for` loop statements.

Repetition

Some language design questions for repetition statements:

- Responsibility for the control variable (normal variable in C, implicit in Ada)
- Value of control variable after loop (value that test failed on in C)
- Type of the control variable (user-defined in C, derived from range in Ada).
- Can control variable be modified (yes in C, no in Ada)
- Can range be empty or changed within the loop?

Repetition

The integer values `1,10` are the start/stop values and `CONTINUE` is a do-nothing statement in FORTRAN.

Repetition in old and modern FORTRAN

```
DO 1010, I=1,10
 ...
1010 CONTINUE
```

```
do i=1,10,2
   ...
enddo
```

What happens if the value of `I` is changed inside the loop? The behaviour is undefined.

Repetition

While loops are designed to repeat a section of code an unknown number of times as long as the condition is finished.

This type of loops has many variations such as testing for the condition at the end of the loop rather than at the start. Syntax varies with `repeat-until` and `do-while` constructs commonly appearing.

A `while` loop executes ***zero or more*** times whereas a `do-while` or `repeat-until` loops executes ***one or more*** times.

Run-Time Error Handling

Sometimes the conditional, loops and break/continue statements are enough to anticipate error conditions. Other times more control structures are needed to deal with them.

Common causes of errors in programs are:

- Domain/data errors
- Resource exhaustion
- Loss of facilities

Domain/Data Errors

Some programs will encounter operations with invalid values such as overflows, divide-by-zero, inverting a singular matrix etc. These are ***synchronous*** errors that will happen every time the program is run with the same data.

These are generally types of error that the programmer is able to detect and test for before attempting to apply an operation.

Resource Allocation

Computer resources are limited (memory, disk space etc) and the machine may not be able to supply a program with the requested resources. These resource may be detectable by failure codes returned by functions such as returning `NULL` from a failed `malloc` in C.Others may be implicit such as running out of stack space in recursive function calls.

This type of error can appear as both ***synchronous*** and ***asynchronous*** errors as they may depend on what resources have been requested by other programs.

Loss of Facilities

Loss of facilities includes a range of problems:

- Loss of power
- Network failure
- Disk crash
- Machine reboot

These types of ***asynchronous*** errors are outside the control of the program.

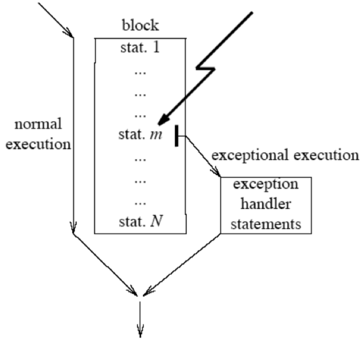
Signals

Signals are one way for a programmer to specify error handling for anticipated run-time errors. Tend to be more closely related to the operating system rather than the programming languages.

Signals are grouped into ***error conditions*** such as a constraint error in Ada for out of bounds or a ***SIGFPE*** floating point error in UNIX.

Exceptions

Exceptions provide a more complete solution to error handling by providing a set of handling statements for each error condition.



Exceptions can be attached to a particular block of statements (rather than the entire program) and have better access to the local data from the procedure that caused the error.

Exception Design

Some language design issues for exceptions to be considered:

- How are exceptions represented (names in Ada, class hierarchy in Java).
- Can new exceptions be defined (yes in Ada, Java)
- Information be extracted from exceptions? (yes in Ada, Java)
- Exception is not caught (propagates to enclosing unit of code in Ada, Java).
- Handlers accept multiple exceptions? (yes in Ada, Java).
- Can programmer raise an exception? (yes in Ada, Java).

Summary

- Internal/External State
- Operators & Associativity
- Flow-Control Statements