# Principles of Programming Languages

CS F372

## Language evaluation

### Readability

The readability of a language is largely dependent on the following characteristics:

- **Simplicity:** The larger a language and the more *constructs* it uses, the more difficult it is to learn. Programmers will only learn a subset of the program. If another programmer learns a different subset, the code will be unreadable to one another. *Feature multiplicity* (e.g. different ways of incrementing a number) and *operator overloading* also contribute to this.
- **Orthogonality**: A language is orthogonal if it has a relatively small set of primitive constructs that can be combined in a relatively small number of ways to build the control and data structures of the language. An example of a lack of orthogonality could be the example that it is impossible to return arrays from a function, but `structs` can be returned. Context dependence also displays a lack of orthogonality - Consider `a+b`. `b` will be treated differently if `a` is a pointer
- **Data Types**: The presence of adequate facilities to define data types and data structures aids readability. E.g. `bool b = true` vs. `int b = 1`
- **Syntax Design:** Syntactic constructs like curly braces or compound statements or special words like `while` greatly affect the readability. Using something like `end loop` is apparently more readable. Moreover, they must be clear - unlike commands like `ls` or `grep` which need prior knowledge to understand

### Writability

Besides simplicity and orthogonality, the following characteristics also affect the writabiliy of a language:

- **Support for abstraction:** Abstraction is defined the same way as in OOP.
- **Expressivity**

### Reliability

A program is reliable if it performs to its specifications under all conditions.

- **Type checking**
- **Exception handling**
- **Aliasing:** This is when two pointers refer to the same variable. Think references in C++
- **Readability and Writability**

### Cost

The cost of a programming language is a function of many characteristics:

- Cost of learning language
- Cost of writing language
- Cost of compiling
- Cost of execution

## Programming paradigms

### Imperative Programming

These languages have been designed based on the popular von Neumann architecture. In the von Neumann architecture, the data and programs are stored in the same memory, but the CPU is separate from the memory. So, the data goes to the CPU and is computed upon, following which it is moved back. The architecture works on a fetch-execute cycle.

Examples: C, Fortran, Pascal

### Functional Programming

These are designed primarily for symbolic data processing. It has been used for symbolic calculations in differential and integral calculus, mathematical logic, etc.;

Examples: LISP, Haskell

### Object Oriented Programming

Very obvious, we had a course on this!

Examples: Java, C++, SmallTalk

### Logic Programming

It uses a specialized form of logical reasoning to answer queries.

Examples : Prolog

# Language Implementation

## Compilation

Here, we translate the high level program code into machine code which the machine can execute. It has several phases:

- Lexical Analysis: Converts characters into lexical units (like operators, identifiers, etc.)
- Syntax Analysis: Transforms lexical units into parse trees which represent the syntactic structure of the program.
- Intermediate code generation and semantic analysis: generate intermediate code and do type checking
- Optimization
- Code generation: machine code is generated

The **load module** is the user and system code together, forming the executable

**Linking and loading** is the process of collecting system program units and linking them to a user program

## Interpretation

Here, we do not translate. The interpreter acts as a virtual machine and runs the language directly. This leads to easier implementation, but it is far slower and needs more space.

While it is slower, it also allows more flexibility and can deal with dynamic properties.

Examples: LISP, JS, PHP

## Hybrid Implementation Systems

It is a compromise between compilers and pure interpreters. Here, we translate into an intermediate language which is then interpreted.

Examples: Perl

## Just In Time Compilation

This is what Java uses. It translates the programs into some intermediate language, and then compiles the intermediate language into machine code when they are called.

# Syntax and Semantics

## Terminology

**Lexeme** A lexeme is the smallest meaningful syntactic unit. E.g. =, index, 2, etc.

**Tokens** A category of lexemes is known as a token. E.g. identifiers, int_literal, plus_op, etc.

## Language Recognizers and Generators

Suppose we have a language $L$ and an alphabet $\sum$. A recognizer $R$ is a mechanism that is capable of reading strings of characters from $\sum$ and indicating whether or not they are part of $L$. The syntax analysis part of a compiler usually acts as a recognizer, determining whether or not the given programs are in the language.

A language generator is a device that can generate sentences in $L$. One use of this is determining whether the syntax of a sentence is correct by comparing it to the output of a generator.

## Backus-Naur Form

In BNF, abstractions are used to represent classes of syntactic structures. They are called **nonterminal symbols**, while lexemes and tokens are called **terminal symbols**. Nonterminals are normally enclosed with angle brackets, like `<if_stmst>`

A **rule** or production has an LHS, which is a nonterminal, and an RHS, which is a string of terminals and/or nonterminals. An example of a rule is:

`<if_stmt>` → if `<logic_expr>` then `<stmt>`

A nonterminal can also have more than one definitions. These can be represented in a single rule by separating the definitions with an | symbol. An example of this is:

`<if_stmt>` → if ( `<logic_expr>` ) `<stmt>` \

| if ( `<logical_expr>`) `<stmt>` else `<stmt>`

### Describing Lists

Syntactic lists can be described with recursion

```
<ident_list> -> identifier | identifier, <ident_list>
```

## Grammars and Derivations

A grammar is a generative device for defining languages. The sentences of the language are generated though a sequence of application of the rules, beginning with a special

nonterminal called the **start symbol**. This sequence of rule applications is called a **derivation**, where we start with a start symbol and end with a sentence consisting entirely of terminal symbols.

Every string of symbols in a derivation is called a **sentential form**, and a sentence is a sentential form consisting entirely of terminal symbols.

A **leftmost derivation** is one in which the leftmost nonterminal in each sentential form is the one that is expanded, while a **rightmost derivation** is one in which the rightmost nonterminal in each sentential form is the one that is expanded. This derivation order has no effect on the language generated by the grammar.

A simple example is the following grammar:

```
<program> -> begin <stmt_list> end
<stmt_list> -> <stmt> | <stmt>;<stmt_list>
<stmt> -> <var> = <expression>
<var> -> A | B |C
<expression> -> <var> + <var> | <var> - <var> | <var>
```

Can this language accept the set of sentences:

```
begin
A=B+C; B=C
end
```

We can prove this by the means of the following derivation

```
<program> -> begin <stmt_list> end
-> begin <stmt> ; <stmt_list> end
-> begin <var> = <expression>; <stmt_list> end
-> begin A = <expression>; <stmt_list> end
-> begin A = <var> + <var>; <stmt_list> end
-> begin A = B + C; <stmt_list> end
-> begin A = B + C; <stmt> emd
-> begin A = B + C; <var> = <expression> end
-> begin A = B + C; B = <var> end
-> begin A = B + C; B = C end
```

This derivation can be represented hierarchically by the means of a parse tree

*Get image from slides*

### Ambiguity in grammar

A grammar that generates two or more distinct parse trees is said to be ambiguous

```
<assign> -> <id> = <expr>
<id> -> A|B|C
```

```
<expr -> <id>|<expr>+<expr>|<expr>*<expr>|(<expr)
```

Try to derive A = B + C * A

If you try to check, this gives us two different parse trees depending on what kind of derivations we do!!

How to we resolve this? Via precedence! If we say that * is precedent over +, then we can get only one parse tree. The lower we are in the parse tree, the higher the precedence.

Now to the next problem - how do we express this in the terms of rules? We start to split up the rules. See the example below

```
<assign> -> <id> = <expr>
<id>  = A|B|C
<expr> -> <expr> + <term> | <term>
<term> -> <term>*<factor> | <factor>
<factor> -> (<expr>) | <id>
```

We might also want to support associativity as well. Operators may be right associative (A+(B+C)) or left associative ((A+B)+C). To indicate right associativity, a right recursive approach is use where the LHS appears at the right end of the RHS. An example is

```
<factor> -> <exp> ** <factor> | <exp>
<exp> -> (<expr) | id
```

# Imperative Programming

A **program** is a concise representation of the computation that occurs when it runs, allowing large sequences of computations to be condensed into a few lines of code.

The behavior of a program may be **static** or **dynamic**. The static form of a program is a simple sequence of instructions written in a file, while the program in execution is represented by the dynamic form of the program. An example is:

```
if (x < 10)
  y = x+z;
else
  y = x-z*2;
```

This is the static form, while in dynamic form it would look like:

```
read x as 13
y = x-z*2
```

Note that the dynamic computation could be longer or shorter than the static form.

**Structured programming** is a paradigm wherein the structure of the program text explains what the program does. The advantages of a structured program is that it is easy to read and modify and easy to tune for efficiency.

Imperative languages are action oriented, in that computations are viewed as a sequence of actions, where an action is a change in the value of a variable.

The **program state** is defied as the tuple of variable values which changes at each computation.

A language is **Turing complete** if it can be used to implement any algorithm (Alan Turing). Any imperative language is Turing complete as long as it supports integers, arithmetic operators, assignment, sequencing, looping and branching.

Any imperative program must have a driver function. It may have more than one function, but there may not be any other module other than the driver function. Every module contains a set of imperative statements.

A **construct** is a piece of code that is derived (constructed) from the underlying grammar that defines the programming language. Examples of this is an if statement, a switch clause, datatypes, operators, etc.

# Grammar

A **grammar** is defined as a four valued tuple $(N, T, S, P)$ where $N$ is the set of non-terminals, $T$ is the set of terminals, $S$ is the start symbol and $P$ is the set of production rules.

An **expression** can be a term or a combination of two terms with a binary operator. This is what we call an expression construct.

The grammar rules to derive an expression are:

```
<expression> -> <term>
<expression> -> <term> <operator> <term>
<term> -> <expression>
<term> -> ID
<operator> -> PLUS,MINUS,MUL
```

This grammar G = ({<expression>, <term>, <operator>}, {ID, PLUS, MINUS, MUL}, <expression>, P)

We can use this grammar to derive expressions! However, it is ambiguous. It is possible to redefine to get a proper grammar.

The design issues for expression construct are:

- Associativity rules

- Which operators to include
- Precedence rules
- Operator overloading
- Type rules

The **statements construct** is used for computation, changing execution flow, and many other uses.

## Computation based statements

The most general form of an assignment statement has at least one variable on the LHS, and the RHS gets the computed value to be stored in the LHS. The RHS could be an expression or a function call.

The syntax of the assignment statement is:

```
<assignment_stmt> -> ID ASSIGNOP <expression>
```

The LHS represents the nonterminal construct . Semantically, this means we must compute the value of and copy the value in the memory location bound to the variable identifier, ID.

The syntax of the function call statement is:

```
<functioncall_stmt> -> ID ASSIGNOP FUNCTION_ID <actual para_list>
```

The semantics indicate that we must copy the value returned by the function in the memory location bound to the variable identifier ID.

## Control flow statements

These are also called control statements, and exhibit a single entry single exit control flow. Examples of this are a switch, if, while, etc.

The design issues are:

- Should be evident from the syntax
- What should be the form and type of expressions that control selection?
- What are the type and scope of loop variables? Can it be reassigned within the loop? How often should the parameter be evaluated?

The syntax of the iterative FOR statement is:

```
<iterative_for_stmt>-> FOR OP <c1> semicolon <c2> semicolon <c3> CP <statements>
<c1>->ID ASSIGNOP <expression>
<c2>-><expression> <relationalOp> <expression>
<c3>-><assignment_stmt>
```

`<relationalOp>->LT|GT|EQ|LE|GE|NE`

The semantics are that we initialize and execute only once, then repeat execution of based on . We execute every time before leaving the loop.

# Variables

A program variable is an abstraction of a computer memory cell or collection of cells. They also have a type which specifies the permissible values and characteristics of the variable. Associating a variables with its type also defines the possible operations that can be performed on the values. Listing them, the advantages of doing this are:

- Increased readability
- Protection through type consistency checks at compile time
- Size of data object can be inferred from its type
- Unsafe/Invalid ops are avoided

By definition, a **type** is a set of values such that there exists a common collection of operations on these values, and the values share a common representation.

**Primitive data objects** are those directly manipulated by the underlying machine. Operations for these are built into the language, and are the building blocks for programmer defined data types. The range of values is normally decided by the machine word size. Values associated with primitive data types can be used freely.

In a strongly typed imperative language, the location to value binding is done at run time, while the name to location binding is done at compile time.

The layout is a plan for fixing relative locations logically corresponding to the variable names. The values are laid out physically at runtime by using the machine representation of the values at actual locations. On most machines, a char is one byte, an integer is one word long, and a real number fits in two contiguous words.

Primitive data types also have their own operations, which can be mathematically described with the form `+: integer x integer -> integer`.

**Enumerated datatypes** are used to defined identifiers which would behave as constants in the location. The values and variables in enumerated types are usually implemented as fixed length bit-strings, with format and size compatible with some integer type.

**Composite data types** are those constructed in a program using its programming language's primitive data types and other composite data types. Examples of these are `struct`, `union`, arrays, pointers, etc. In arrays, we have a collection of elements of the same type, laid out in consecutive machine locations.

# Arrays

An array allows random access of its indices within a range. We can computer the address of `A[i]` with `base + (i)*w`. Here, `w` is the width of an element, and base is `A[low]`. `w` is known at compile time, but `i` is computed at runtime.

In Ada and Pascal, we can have array subranges as [low...high]. Now, the base address will be `A[i] = base + (i-low)*w`.

The type expression for an array is declared as (array,<low,high>,type).

Dynamic arrays are array variables whose index ranges may be expanded at any time after creation, without changing the values of its current elements. This is allowed in Perl, but not in C. Notice, `realloc` does not mean C arrays are dynamic, since you still need to define the size.

Arrays can also be multidimensional. They could have two memory layouts - row major (rows appear side by side) or column major (columns appear side by side).

We could also have **jagged arrays**, where every row is not of the same size. Think of `vector<vector<int>>` in C++.

Depending on the layout, we could calculate the address of a given array element. For instance, in a 2D array, A[i][j] is addressed by `base + [i*n + j]` where `n` is the number of elements in a row. To generalize this concept, let us see the formula for a 4D array at location `A[m][n][p][k]` - $base + (m * n_{R_2} * n_{R_3} * n_{R_4} + n * n_{R_3} * n_{R_4} + p * n_{R_4} + k)$.

Arrays could be of multiple types:

- **Static Array**: Subscript ranges are statically bound and storage allocation is static. This applies to C arrays that have the `static` modifier.
- **Fixed stack dynamic array**: Subscript ranges are statically bound, but allocation is done at elaboration time during execution. Arrays in C without the `static` modifier are fixed stack dynamic arrays.
- **Stack dynamic array**: Subscript ranges are dynamically bound, while storage allocation is dynamic during execution. Once bound they remain fixed during variable lifetime. Ada arrays are an examples of this.
- **Fixed heap dynamic array**: Subscript ranges are dynamically bound and storage allocation is dynamic, but they are both fixed after storage is allocated. Bindings are done when user requests them, not at elaboration time. Storage is on the heap. C provides this via `malloc`, `calloc`, etc. All arrays in Java are fixed heap dynamic.
- **Heap dynamic array**: Subscript ranges are dynamically bound, storage allocation is dynamic, and it can change any number of times during the array's lifetime. ArrayList from C# is an example of this.

# Records

Here, data is described with various attributes of various types. Through this, we can group together relevant information into a single data structure. The operation on record is selection of a field by name. All records are laid as contiguous blocks of memory, sometimes with padding to conform to word boundaries.

The type expression of a record is the Cartesian product of its fields. For instance, the struct below has the type signature `int x int`.

```
struct complex {
  int real;
  int imaginary;
}
```

In an array of structs, the relative offset is computed at compile time.

A `union` is a different construct from a `struct`. Unlike `struct`, `union` only allocates as much memory as the largest member of the `union`. So, in a `union` we can technically only use only one variable from it at a time - changing, say, `union.y` would also change the value of `union.x`, since they use the same memory! These are also called **free unions**. The type expression of a `union` looks like `<a|b|c>`.

A **tagged union** or a **discriminated union** is a data structure used to hold a value that could take on several different fixed types. Only one of these can be in use at a time, and it has a tag field that explicitly indicates which one is of use. This is also called it's **discriminant**. In C, this can be implemented as a `struct` which has a nested `union` and tags to indicate which variable is in use.

Tagged unions are directly supported in Ada, ML, Haskell and F#. In Ada, the tagged union looks something like this:

```
procedure shapedemo is
  type shape is (circle, rectangle, triangle);
  type colors is (red,green,blue);
  type figure (form : shape) is
    record
      filled : boolean;
      color: colors;
      case form is
        when circle=>
          diameter : float;
        when triangle =>
          left_side: integer;
          right_side : integer;
          angle : float;
```

```
        when rectangle =>
          side_1 : integer;
          side_2 : integer;
      end case;
    end record;
    f1: figure(form=> rectange);
    f2: figure(form=> triangle);
begin
  f1:= (filled=>true,color=>blue, form=> rectangle, side_1=>12, side_2=>3);
end shapedemo;
```

Here the type of `form` would be `<circle | triangle | rectangle>`. So, the type of `figure` is `boolean x triangle x <circle | triangle | rectangle>`.

The type expression of `f1` is `boolean x triangle x <circle|triangle|rectangle> x (tag=rectangle)`.

The type checking could be done statically, in the case where we declare `f1` to be constrained to a triangle. In these cases, we can announce type errors at compile time. There is also an option for dynamic type checking, when `f1` is declared without specifying its tag. In these cases, we would do the checking at runtime.

Assume we declared `F1` as `figure`. Then, it is an **unconstrained variant record** that has no initial value, and its type can be changed by assignment of a whole record, including its discriminant. However, if we declare `F2` as `Figure(form => Triangle)`, it is constrained to be a triangle and cannot be changed to another variant.

# Sets

Sets of $n$ numbers, each less than $n$ are implemented by a bit vector of $n$ bits. If it is set, we surmise that the number it present in the set. This, however, only works for relatively small numbers.

Operations on sets are implemented as bitwise operations. This is not supported in C, but it is supported in Pascal.

# Pointers

A pointer type is a value that provides indirect access to elements of a known type. They can take up any address value, as well as NULL. The only operations that can be done on it are `=`, `+`, `-`, `->`, `*`.

The grammar for C like pointer declaration would look like:

```
<declarationStmt> -> <Type> <list>
```

```
<list> -> ID COMMA <List> |ID|STAR ID COMMA <List> | STAR ID
```

Pointers can provide reference to **heap dynamic variables**. Heap dynamic variables are those that are dynamically allocated from the heap. These do not have identifiers associated with them, and hence are also called **anonymous variables**.

Pointers could result in dangling pointer problems (dereferencing a deallocated variable) or memory leaks (losing access to allocated memory).

Ada prevents this by not allowing users to explicitly deallocate. This reduces the possibilities of dangling pointers, but can cause memory leaks.

FORTRAN 95+ has implicit dereferencing of the pointer, by judging the type.

# Functions

A **function definition** construct has a specific name and formal parameters to communicate with the calling procedure. A **function call** is a construct that uses name of existing procedures/functions through the actual parameters.

Functions are allowed to make nested calls to other functions, or even itself. The activations of procedures during execution of the program is represented by a tree known as an **activation tree**. In the code snippet:

```
p()
{
  q();
  r();
}

q()
{
  u();
  v();
}
```

We would get the activation tree:

```
p
|  \
q    r
| \
u  v
```

Each live activation of a procedure call is maintained by a data structure called an **activation record**. When a procedure is called, its activation record is pushed onto the stack. Once it terminates, it gets popped off the stack. Non overlapping functions

can even share stack space. The **calling sequence** is the compiler code that allocates activation records onto the stack.

When calling a function, the caller and callee split the work of allocating and filling in the fields in the callee's frame. In C, the caller is responsible for placing the actual parameters in the callee's activation record. It also stores information needed to restart execution of the caller (like the instruction pointer.)

The formal parameters, actual parameters and return value are all stored in the beginning of the callee's activation record. The middle of the activation record normally contains fixed length items (control link, access link, m/c status fields), while local variables are placed at the end.

The **control link** is the link used at runtime to maintain address of the calling function's activation record. The **access link** is used to implement the nested scoped languages following static scope rules. It points to the activation record of the function whose code text has the variable declarations.

Some languages allow **nested procedures**, which means that a function is defined within a function.

Names are bound to variables by the means of scope. A variable may be local (available only within the scope of the code block/function/etc.) or non-local (visible in this code bloack, but defined outside it). Scope can either by static or dynamic. In static scoping, the scope of the variable is computed at compile time, and the code segment holding the non-local variable's declaration is known as **static parent**. Dynamic scoping is based on the calling sequence of the functions. In other words, the association of a named occurrence of a variable to its declaration does not depend on the position in the code text, but rather on the text segment of the function that called it.

# Lambda Calculus

Pure lambda calculus consists of just 3 constructs - variables, function application, and function creation. It's syntax is:

$$e ::= x|\lambda x.e|e_1e_2|(e)$$

Here, $x$ is a variable. $\lambda x.e$ is called a function abstraction, where $x$ is the input and $e$ is the body of the function. An example of this could be $\lambda x.(x^2 + 1)$. $e_1e_2$ is called a function application. It "applies" $e_2$ to the function $e_1$, which is like passing the parameters $e_2$ to $e_1$. $(e)$ is a bracketed expression.

By convention, the application of functions is **left associative**. This means that in the case of the expression $e_1e_2e_3$, we get $(e_1e_2)e_3$.

The scope of a variable expands as far right as possible. This means that if we have the expression $\lambda x.x\lambda y.xy$, this would give us the bracketed expression $\lambda x.(x\lambda y.(xy))$.

Here is an interesting example : $\lambda x.\lambda yx$. This means that we have a function that takes $x$ as an input, which returns a function that takes $y$ as an input, which returns $x$. Another way to look at it is that it takes 2 arguments $x$ and $y$ and returns the first argument $x$.

Another example is $\lambda f.\lambda x.f(fx)$ . This is a function that takes in $f$ and $x$, and then returns $f(f(x))$

In the function $\lambda x.xy$, we say that $x$ is **bound** and $y$ is **free**. $\lambda x$ acts as the **binder** for $x$.

## Alpha renaming

Something that may happen is that variables occur twice in a function. Consider the expression $\lambda x.x(\lambda x.x)x$. It becomes more and more confusing which variable is bound to which binder. To remedy this, we use $\alpha$ **renaming**. The expressions $\lambda x.x$ and $\lambda y.y$ are equivalent, so we rename different bound variables. The expression would hence become $\lambda x.x(\lambda y.y)x$.

## De Bruijin Index

The same general idea that variable names don't matter results in the **de Bruijin notation**. We define the **de Bruijin index** as the number of $\lambda$s that separate the occurrence from the binder. This results in an expression like $\lambda x.\lambda y.xy$ having $\lambda.\lambda.1.0$ as an equivalent De Bruijin expression.

## Beta reduction

Let us try to understand the semantics of application. Consider the expression $(\lambda x.e_1)e_2$, where $e_1$ is a body with many occurrences of $x$. This is equivalent to replace every (free?) occurrence of $x$ in $e_1$ with $e_2$. This is denoted by $[x \to e_2]e_1$. This is a type of substitution called $\beta$ **reduction**. Some examples are:

- $[x \to S]x = S$
- $[x \to S]y = y$
- $[x \to S](\lambda y.e) = \lambda y.[x \to S]e$
- $[x \to S](e_1e_2) = ([x \to S]e_1)([x \to S]e_2)$

Let us see a more involved example - $((\lambda a.a)\lambda b.\lambda c.b)(x)\lambda e.f$. We can remove $\lambda a.a$ with the following expression with $\beta$ reduction, giving us $(\lambda b.\lambda c.b)(x)\lambda e.f$. Now doing the reduction with $x$, we get $(\lambda c.x)(\lambda e.f)$. Finally, we can reduce to $x$ as there is no occurrence of $c$. This is called the **beta normal form**. Whenever there is more than

one $\beta$ reduction, it will always result in the same final $\beta$ normal form (Church Rosser Theorem).

There are in general two ways to go about this reduction - **call by value** or **call by name**. In call by value, we evaluate the argument of a function before passing it. In call by name, we do not evaluate the argument of a function. Instead it is demand driven, and only evaluated when needed. For instance, in the expression $(\lambda y.(\lambda x.x)y)((\lambda u.u)(\lambda v.v))$ we would not evaluate the lambdas on $u$ or $v$. We however would reduce it in call by value.

Another odd examples is $[x \to z](\lambda z.x)$. We cannot directly do the replacement, since that would result in the free variable $x$ becoming a bound variable $z$. Instead, we should do $\alpha$ renaming, renaming $z$ to $w$, and then do the substitution.

## Combinators

A **combinator** refers to any $\lambda$ term without any free variables.

The first combinator is the **identity combinator**. It is given by $I := \lambda a.a$. Of course, $Ix$ always returns $x$.

Another combinator is the **mockingbird operator**, given by $M := \lambda f.ff$. For example, let us look at the result of $MI$. This becomes $\lambda f.ff\lambda x.x$. By Beta reduction, it becomes obvious the result is $I$. Unsurprisingly, $MM$ leads to a stack overflow, as it results in the same expression.

The **kestrel operator** is given by $K := \lambda ab.a$ or $K := \lambda a.\lambda b.a$. This is kind of an absorption function - it takes two parameters and returns the first one. What is the result of , say, $KMI$? By intuition or beta reduction we would get $M$.

The **KITE operator** is given by $KI := \lambda ab.b$. It is also absorption, but the opposite of the Kestrel!

The **CARDINAL operator** is given by $C := \lambda fab.fba$. For example, $CKIM$ would become $KMI$, and the final result would be $M$. One interesting result is that $CK$ is equivalent to $KI$.

## Boolean Logic

Let us say that $T = (\lambda x.\lambda y.x)$ (the kestrel) and $F = (\lambda x.\lambda y.y)$ (the kite).

The AND operation would be given by $\lambda a.\lambda b.abF$. As we can see, AND T T would given us T (by beta reduction).

The NOT operation is given by $\lambda a.aFT$. If we pass T, by beta reduction, we get $TFT$ which we know is $F$. Passing $F$ gives us $FFT$, which gives us $T$. This is the same as the Cardinal Operation!

We can simulate branches (IF) with lambda expressions. Branches of the form:

```
if c then
  a
else
  b
```

are given by $IF\ cab$. Of course, $IF\ Tab$ should give $a$, and $IF\ Fab$ should return $b$. This can simply be done with $\lambda a.a$, the identity combinator.

## Church's Numerals

We can also simulate numbers as lambda expressions!. 0 is given the same way as FALSE in Boolean logic - $\lambda f.\lambda x.x$, once again the Kite operator. With just this we can show any number!.

$1 = \lambda f.\lambda x.fx$

$2 = \lambda f.\lambda x.f(fx)$

$3 = \lambda f.\lambda x.f(f(fx))$

and so on...

We can simulate the ++ operator with the **successor function**. It is given by $succ = \lambda n.\lambda f.\lambda x.f(nfx)$. So, succ 0 = 1 !

This can help us create an ADD function. ADD(n,m) is given by $\lambda n.\lambda m.\lambda f.\lambda x.nf(mfx)$. This replicates a process where the successor function is called on $m$, $n$ times.

Finally, we can look at multiplication! Just like addition, we can define MULT(n,m) as adding $n$ $m$ times, given by $\lambda n.\lambda m.m(ADDn)0$

Finally, let us look at computing factorial through recursion. Let us assume that we have a function isZero and a function pred. Then, we can define FACT as

$\lambda n.$ if (isZero $n$) (1) (MULT n (FACT ( pred $n$ )))

However, we cannot write this function as we are recursively defining it!. For this we need the **Y combinator**, given by $(\lambda x.\lambda y.y(xxy))(\lambda x.\lambda y.y(xxy))$. Let us apply this to some variable foo, as in (Y foo). By beta reduction we can see, we get foo(Y foo)!.

Finally, we can define factorial as:

$Y(\lambda f.\lambda n.$ if (isZero $n$) (1) (MULT $n$ ($f$ ( pred $n$ ))))

## Programs

Lambda calculus is implemented in Haskell

Haskell uses **currying**. Every function in Haskell takes one argument, and a multi-argument function is a sequence of single argument function, like e.g. `sum m n = (m+n)`. This is in fact application. The type of this function would be `int ->(int -> int)`. As we can see, the types are right associative, but the currying is left associative.

A list is built up from [] (empty list) by append operator `:`. The list [1,2,3,4] is internally implemented as `1:(2:(3:[]))`. Any list `[1,2,3]` is also equivalent to things like `1:2:3:[]`.

`head` returns the first element of a list, and `tail` returns the list without the first element. `head(x:xs) = x`, and `tail(x:xs) = xs`.

Some more built in functions are:

- `init` : returns the list without the last element
- `last` : returns the last element
- `reverse` : reverses the list
- `length` : returns the length
- `attach` : concatenates two lists
- `take` : returns the first $n$ elements of list $l$
- `drop` : leaves the last $n$ elements of list $l$ (check if true)

How do we define functions? The syntax is:

```
fname :: Type
fname definition
```

An example is:

```
attach :: [Int] -> [Int] -> [Int]
attach [] l = l
attach (x:xs) l = x : (attach xs l)
```

We could pass a function as an argument. These functions are known as **higher order functions**. Assume we have `apply f x => f x`. It applies first argument to second argument. If the type of `f` is `a->b`, then the type of `apply` is `apply :: (a->b)->a->b`

# Object Oriented Programming

An Object Oriented Programming Language has the following characteristics:

- Abstract Data Types (ADT)
- Inheritance
- Dynamic Binding

Generally, programming with ADTs has the following issues:

- Future requirements change and require modification in the code

- Program problem space is difficult to manage and difficult to organize program code

To solve this, OOP languages use inheritance. ADTs supported with inheritance are allowed to inherit data and functionality, as well as add new entities without changing other reused data types. The issue with this approach is that it creates dependencies in the form of some inheritance hierarchy.

OOP Languages face multiple design issues:

- **Exclusivity of Objects:** In OOP, everything is an object. The advantage of this is its elegance and purity, but the issue is that operations become much slower even on simple objects. One alternative is to have an imperative language with object oriented support. This makes the language larger and confusing to all but the experts. Java chooses to have primitive scalar types, but have structured types as Objects. This leads to some complications because invariably, we will need to mix the primitives with objects, creating a need for wrapper classes
- **Are Subclasses Subtypes?:** Does an IS-A relationship hold between a parent class object and an object of the subclass? A derived class is called a subtype if it has an IS-A relationship with its parent class. The methods of the subclass that override parent class methods must be type compatible with their corresponding overridden methods. This means that a call to an overriding methods can replace any call to the overriden methods in any appearance in the client program without causing type errors.
- **Multiple Inheritance:** This allows a new class to inherit from two or more classes. The issue is that it creates more language and implementation complexity, as well as potential inefficiency when dynamic binding happens.
- **Allocation and Deallocation:** Objects could be allocated on the stack or the heap. Let us consider a subclass B of class A, and the assignment `a1 = b1` where the former is an object of A and the latter is an object of B. If they are heap-allocated, this is simply changing the memory location the pointer is pointing to. If they are stack allocated we face the problem of **object slicing**. Here, the size of `b1` could be larger than `a1`, forcing us to reallocate space.
- **Dynamic and Static Binding:** Should all binding of messages to methods be dynamic? Should the user be allowed to specify?
- **Nested Classes:** Can the new class be nested inside the class that uses it? Which facilities of the nesting class should be visible to the nested class?
- **Initialization of Objects:** Are objects initialized to values when they are created? How are parent class members initialized when a subclass object is created.

## Support for OOP in C++

C++ has a mixed typing system, with imperative and OOP styles present. Destructors are used for explicit deallocation of memory in heap dynamic objects.

The access controls are:

- `private` (visible only in class and friends)
- `public` (visible in subclasses and clients)
- `protected` (visible in class and in subclasses, but not clients)

The subclassing process can be declared with access controls (private or public), which define potential changes in access by subclasses:

- Private derivation : inherited members are private in the subclass
- Public derivation : public and protected members are also public and protected in subclasses

Private derived classes cannot be subtypes. A member that is non accessible in a subclass can be declared to be visible using the `::` operator.

Multiple inheritance is supported in C++, along with support for dynamic binding. This is done through the use of the `virtual` keyword. A very clear example of the use of this keyword is:

```
class base {
public:
    virtual void print()  {
        cout << "print base class" << endl;
    }
    void show() {
        cout << "show base class" << endl;
    }
};

class derived : public base {
public:
    void print() {
        cout << "print derived class" << endl;
    }
    void show() {
        cout << "show derived class" << endl;
    }
};

int main()
{
```

```
    base* bptr;
    derived d;
    bptr = &d;
    // virtual function, bound at runtime
    bptr->print();
    // Non-virtual function, bound at compile time
    bptr->show();
}
```

The output of the above code is:

```
print derived class
show base class
```

Dynamic binding in C++ is faster than dynamic binding in SmallTalk, since binding a virtual member function call in C++ to a function definition has a fixed cost, but in SmallTalk it depends on how far away in the inheritance hierarchy the correct method is.

### Implementing OOP Constructs

**Class instance records** store the storage structure of the instance variables of a class, that is, the state of an object. This is built at compile time. If a class has a parent, the subclass instance variables are added to the parent CIR. Since the CIR is static, access to all instance variables is done using constant offsets from the beginning of the CIR instance as it is in records.

Dynamically bound methods must have entries in the CIR. The calls are connected via a pointer in the CIR. This storage structure is called a **virtual method table**. Polymorphic variables of a base class always reference the CIR of the correct type object thereby binding to the correct version of the dynamically bound method. Static methods do not have entries in the CIR.

## Logical Programming Languages

Logic programming languages are declarative in nature. It consists of two important pieces of knowledge, called facts and rules. A program is used to prove if a statement is true and to answer the queries.

Logic programming languages need a unified way to state propositions. One relatively simple one is the **clausal form**, which can be used to represent any proposition. It's general syntax is:

$$B_1 \cup B_2 \cup ... \cup B_n \subset A_1 \cap A_2 \cap ... \cap A_m$$

The meaning of this is that if all $A_i$ are true, then at least one of $B_j$ is true. This way, we have no use for existential or universal quantifiers and only need conjunction and disjunction. Conjunction and disjunction can also be found on separate sides of the implication. The right side of this is called the antecedent, while the left side is consequent.

**Resolution** is an inference rule that allows inferred propositions to be computed from given propositions. If we have all of our propositions in clausal form, we can generate a new proposition from two propositions by finding the conjunction of the left side and the disjunction of the right side. However, it is necessary to find values for variables which allows matching of propositions to succeed, which is done in a step called **unification**.

Prolog is an example of such a language. Prolog has a single datatype called a **term**. Terms can be:

- **Atom** : An atom is a general purpose name with no meaning.
- **Number** : A number can either be a float or an integer. Most major Prolog systems support arbitrary length integer numbers.
- **Variables** : Variables are denoted by a string consisting of letters, numbers and underscore characters. They always begin with an uppercase letter or an underscore.
- **Compound terms**: This is composed of an atom called a functor and a number of terms called arguments. Examples of this include lists and strings.

Prolog programs consists of relations, which may be **rules** or **facts**. They are always Horn clauses. A rule is of the form `Head :- Body`, and is read as "Head is true if Body is true". A fact is a clause with an empty body, for example `cat(tom)`, which is equivalent to a rule `cat(tom) :- true`. Rules are also called "Headed Horn Clauses" while facts are called "Headless Horn Clauses" (I know, weird).

Prolog uses SLD resolution on the negated query (called the goal) to answer queries. This involves a depth first search with backtracking. Prolog uses something called **top down resolution**.

*Note : This was not covered extensively so my notes suck*