COURSE CODE: **CSC 411**
COURSE TITLE: **Organization of Programming Languages**
NUMBER OF Units: **3 Units**

Course Duration: **Three hours per week**

COURSE DETAILS:
Course Lecturer: **Dr. O. Folorunso**
B.Sc(UNAAB)., M.Sc(UNILAG)., PhD(UNAAB)
Email: folorunsolusegun@yahoo.com, folorunsoo@unaab.edu.ng
Office Location: Room B201, COLNAS
Consultation hours: 12-2pm, Wednesdays & Fridays
Lecture Note developed by: The Department of Computer Science, University of Agriculture, Abeokuta
Head of Department: Dr. A. F. Adekoya

Course Content: Introduction and brief history of programming languages, Imperative programming, Generative Grammars, Lexical and syntactic analysis, variables, bindings and scope, data types and type checking, functional programming scheme, expression of assignments, program statements, program units, logic programming.

Course Description: The course is designed to describe the fundamental concepts of programming language by discussing the design issues of the various language constructs, examining the design choices for these constructs in some of the most common languages, and critically comparing design alternatives. The course also introduces the fundamental syntactic and semantic concepts underlying modern programming languages. The imperative, functional and logic programming paradigms will be discussed, with illustrative examples in C/C++, Java, Ada, Scheme, Python and Prolog, major topics include:

- Context Free Grammars, Lexical Analysis and Syntactic Parsing
- Bindings, Type Checking and Scopes
- Expressions, Control Structures and Functions

Course Justification:

Any serious study of programming languages requires an examination of some related topics among which are formal methods of describing the syntax and semantics of programming languages and its implementation techniques. The need to use programming language to solve our day-to-day problems grows every year. Students should be able to familiar with popular programming languages and the advantage they have over each other. They should be able to know which programming language solves a particular problem better. The theoretical and practical knowledge acquired from this course will give the students a foundation from which they can appreciate the relevant and the interrelationships of different programming languages.

Course Objectives:

The general objective of the course as an integral part of the Bachelor Degree for Computer Science Students in University of Agriculture, Abeokuta, is to:

- To increase capacity of computer science students to express ideas
- Improve their background for choosing appropriate languages
- Increase the ability to learn new languages
- To better understand the significance of programming implementation
- Overall advancement of computing

Course Requirements:

This is a compulsory course for all computer science students in the University. In view of this, students are expected to participate in all the course activities and have minimum of 75% attendance to be able to write the final examination.

Reading List:

1. Michael L. Scoot. Programming Language Pragmatics. Morgan Kaufmann, 2006.
2. Allen B. Tucker and Robert Noonan. Programming Languages Principles and Paradigm. Mcgraw-Hill Higher Education, 2006
3. Robert W. Sebesta. Concepts of Programming Languages. Addison Wesley, 2011.
4. Aho, A. V., J. E. Hopcroft, and J. D. Ullman. The design and analysis of computer algorithms. Boston: Addison-Wesley, 2007.
5. www.Wikipedia.com
6. Kasabov NK, (1998) Foundations of Neural Networks, Fuzzy Systems, and . Knowledge Engineering. The MIT Press Cambridge.
7. Bezem M (2010) A Prolog Compendium. Department of Informatics, University of Bergen. Bergen, Norway
8. Rowe NC. (1988) Artificial Intelligence through Prolog. Prentice-Hall
9. Bramer M (2005) Logic Programming with Prolog. Springer. USA
10. Prolog Development Center (2001) Visual Prolog Version 5.x: Language Tutorial. Copenhagen, Denmark

LECTURE CONTENT

Week 1:

Reasons for studying concepts of programming language, Application domains, criteria for language evaluation, influences on language design, language paradigms, language design trade-offs, implementation method.

Objective:

1. To increase the student's capacity to use different constructs
2. To enable students to choose language more intelligently
3. To make learning new languages easier

Description:

The course will lay emphasis on programming evaluation criteria like readability, writ ability, reliability and cost, Major influence of language design relating to machine architecture and software development methodology will be exhaustively discussed Major implementation methods include: compilation, pure interpretation, hybrid and just-in-time will be discussed.

Lecture Note:

Reason for studying concepts of programming language, Application domains, critical for language evaluation, influences on language design, language paradigms, language design trade-offs, implementation methods.

REASONS FOR STUDYING CONCEPTS OF PROGRAMMING LANGUAGES

1. **Increased ability to express ideas/algorithms**

   In Natural language, the depth at which people think is influenced by the expressive power of the language they use. In programming language, the complexity of the algorithms that people

   Implement is influenced by the set of constructs available in the programming language

2. **Improved background for choosing appropriate Languages**

   Many programmers use the language with which they are most familiar, even though poorly suited for their new project. It is ideal to use the most appropriate language.

3. **Increased ability to learn new languages**

   For instance, knowing the concept s of object oriented programming OOP makes learning Java significantly easier and also, knowing the grammar of one's native language makes it easier to learn another language.

4. **Better Understanding of Significance of implementation**

5. **Better use of languages that are already known**

6. **The overall advancement of computing**

APPLICATION DOMAINS

1. Scientific Applications
2. Data processing Applications
3. Text processing Applications
4. Artificial intelligence Applications
5. Systems Programming   Applications
6. Web software

   SCIENTIFIC APPLICATIONS can be characterized as those which predominantly manipulate numbers and arrays of numbers, using mathematical and statistical principles as a basis for the algorithms. These algorithms encompass such problem as statistical significance test, linear programming, regression analysis and numerical approximations for the solution of differential and integral equations. FORTRAN, Pascal, Meth lab are programming languages that can be used here.

DATA PROCESSING APPLICATIONS can be characterized as those programming problems whose predominant interest is in the creation, maintenance, extraction and summarization of data in records and files. COBOL is a programming language that can be used for data processing applications.

TEXT PROCESSING APPLICATIONS are characterized as those whose principal activity involves the manipulation of natural language text, rather than numbers as their data. SNOBOL and C language have strong text processing capabilities

ARTIFICIAL INTELLIGENCE APPLICATIONS are characterized as those programs which are designed principally to emulate intelligent behavior. They include game playing algorithms such as chess, natural language understanding programs, computer vision, robotics and expert systems. LISP has been the predominant AI programming language, and also PROLOG using the principle of ‘’Logic programming’’ Lately AI applications are written in Java, c++ and python.

SYSTEMS PROGRAMMING APPLICATIONS involve developing those programs that interface the c omputer system (the hardware) with the programmer and the operator. These programs include compilers, assembles, interpreters, input-output routines, program management facilities and schedules for utilizing and serving the various resources that comprise the system. Ada and Modula – 2 are examples of programming languages used here. Also is C.

WEB SOFTWARE

Edectio collection of languages which include:

- Markup (e.g. XHTML)
- Scripting for dynamic content under which we have the
  - Client side, using scripts embedded in the XHTML documents e.g. Javascript, PHP
  - Server side, using the commonGateway interface e.g. JSP, ASP, PHP
- General- purpose, executed on the web server through cGI e.g. Java, C++.

CRITERIA FOR LANGUAGE EVALUATION AND COMPARISION

1. Expressivity
2. Well – Definedness
3. Data types and structures
4. Modularity
5. Input-Output facilities
6. Portability
7. Efficiency
8. Pedagogy
9. Generality

Expressivity means the ability of a language to clearly reflect the meaning intended by the algorithm designer (the programmer). Thus an "expressive" language permits an utterance to be compactly stated, and encourages the use of statement forms associated with structured programming (usually "while "loops and "if – then – else" statements).

By "well-definiteness", we mean that the language's syntax and semantics are free of ambiguity, are internally consistent and complete. Thus the implementer of a well-defined language should have, within its definition a complete specification of all the language's expressive forms and their meanings. The programmer, by the same virtue should be able to predict exactly the behavior of each expression before it is actually executed.

By "Data types and Structures", we mean the ability of a language to support a variety of data values (integers, real, strings, pointers etc.) and non elementary collect ions of these.

Modularity has two aspects: the language's support for sub-programming and the language's extensibility in the sense of allowing programmer – defined operators and data types. By sub programming, we mean the ability to define independent procedures and functions (subprograms), and communicate via parameters or global variables with the invoking program.

In evaluating a language's "Input-Output facilities" we are looking at its support for sequential, indexed, and random access files, as well as its support for database and information retrieval functions.

A language which has "portability" is one which is implemented on a variety of computers. That is, its design is relatively"machine – independent". Languages which are well- defined tend to be more portable than others.

An "efficient" language is one which permits fast compilation and execution on the machines where it is implemented. Traditionally, FORTRAN and COBOL have been relatively efficient languages in their respective application areas.

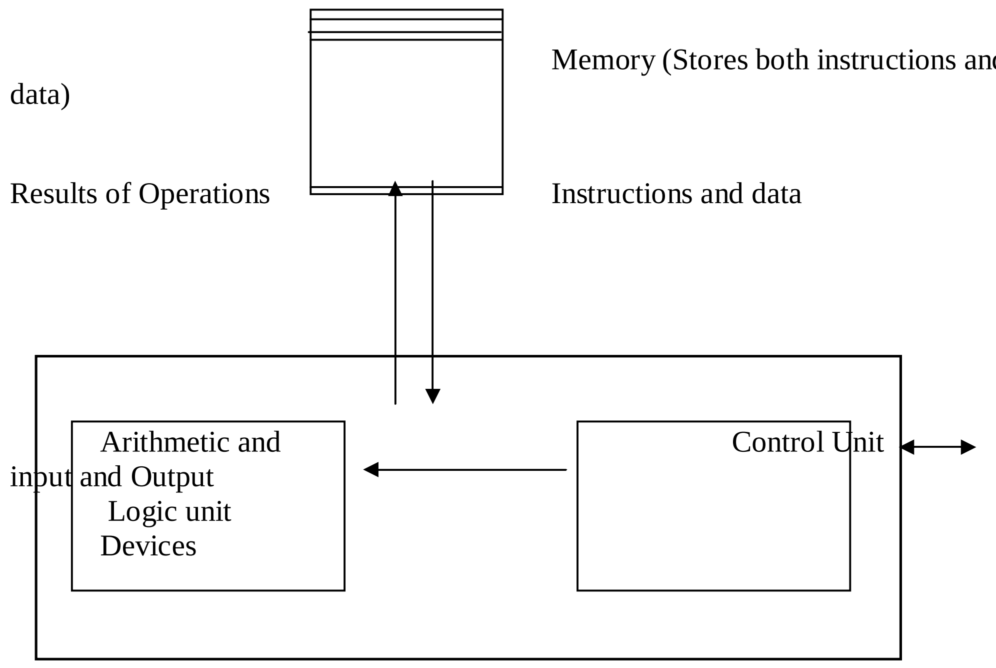
Some languages have better "pedagogy" than others. That is, they are intrinsically easier to teach and to learn, they have better textbooks; they are implemented in a better program development environment, they are widely known and used by the best programmers in an application area.

Generality: Means that a language is useful in a wide range of programming applications. For instance, APL has been used in mathematical applications involving matrix algebra and in business applications as well.

INFLUENCES ON LANGUAGE DESIGN

1. Computer Architecture: Languages are developed around the prevalent computer architure, known as the Von Neumann architecture (the most prevalent computer architecture).

Fig; Von Neumann Architecture



The connection speed between a computer's memory and its processor determines the speed of that computer. Program instructions often can be executed much faster than the speed of the connection; the connection speed thus, results in a bottleneck (Von Neumann bottleneck). It is the primary limiting factor in the speed of computers.

2. **Programming Methodologies:** New software development methodologies (e.g. object Oriented Software Development) led to new paradigms in programming and by extension, to new programming languages.

LANGUAGE PARADIGMS (Developments in Programming Methodology)

1. Imperative

This is designed around the Von Neumann architecture. Computation is performed through statements that change a program's state. Central features are variables, assignment statements and ileration, sequency of commands, explicit state update via assignment. Examples of such languages are Fortran, Algol, Pascal, e/c++, Java, Perl, Javascript, Visual BASIC.NET.

2. Functional

Here, the main means of making computations is by applying functions to parameters. Examples are LISP, Scheme, ML, Haskell. It may also include OO (Object Oriented) concepts.

3. Logic

This is Rule-based (rules are specified in no particular order). Computations here are made through a logical inference process. Examples are PROLOG and CLIPS. This may also include OO concepts.

TRADE-OFFS IN LANGUAGE DESIGN

1. Reliability Vs. Cost of Execution: For example, Java demands that all references to array elements be checked for proper indexing, which leads to increased execution costs.
2. Readability vs. Writability: - APL provides many powerful operators land a large number of new symbols), allowing complex computations to be written in a compact program but at the cost of poor readability.
3. Writability (Flexibility) vs. reliability: The pointers in c++ for instance are powerful and very flexible but are unreliable.

IMPLEMENTATION METHODS

1. Compilation – Programs are translated into machine Language & System calls
2. Interpretation – Programs are interpreted by another program (an interpreter)
3. Hybrid – Programs translated into an intermediate language for easy interpretation
4. Just –in-time – Hybrid implementation, then compile sub programs code the first time they are called.

COMPILATION

- Translated high level program (source language) into machine code (machine language)
- Slow translation, fast execution
- Compilation process has several phases
    - Lexical analysis converts characters in the source program into lexical units (e.g. identifiers, operators, keywords).
    - Syntactic analysis: transforms lexical units into parse trees which represent the syntactic structure of the program.
    - Semantics analysis check for errors hard to detect during syntactic analysis; generate intermediate code.
    - Code generation – Machine code is generated
- **INTERPRETATION**
- - Easier implementation of programs (run-time errors can easily and immediately be displayed).
- Slower execution (10 to 100 times slower than compiled programs)
- Often requires more memory space and is now rare3 for traditional high-level languages.
- Significant comeback with some Web scripting languages like PHP and JavaScript.
- Interpreters usually implement as a read-eval-print loop:
    - Read expression in the input language (usually translating it in some internal form)
    - Evaluates the internal forms of the expression
    - Print the result of the evaluation
    - Loops and reads the next input expression until exit
- Interpreters act as a virtual machine for the source language:
    - Fetch execute cycle replaced by the read-eval-print loop
    - Usually has a core component, called the interpreter “run-time” that is a compile program running on the native machine.

HYBRID IMPLEMENTAITON

- This involves a compromise between compilers and pure interpreters. A high level program is translated to an intermediate language that allows easy interpretation.
- Hybrid implementation is faster than pure interpretation. Examples of the implementation occur in Perl and Java.
    - Perl programs are partially compiled to detect errors before interpretation.
    - Initial implementat6ions of Java were hybrid. The intermediate form, byte code, provides portability to any machine that has a byte code interpreter and a run time system (together, these are called Java Virtual Machine).

JUST-IN-TIME IMPLEMENTATION

This implementation initially translates programs to an intermediate language then compile the intermediate language of the subprograms into machine code when they are called.

- Machine code version is kept for subsequent calls. Just-in-time systems are widely used for Java programs. Also .NET languages are implemented with a JIT system.

Study Questions:
1. Why is it useful for a programmer to have some background in language design, even though he or she may never actually design a programming language?
2. What does it mean for a program to be reliable?
3. What is: Aliasing?; What is exceptional handling?
4. Why is readability important in writability?
5. What are the three fundamental features of an object-oriented language?
6. What role does symbol table play in compiler?
7. What does a linker do?
8. What are the advantages in implementing a language with pure interpreter?
9. What are the three general methods of implementing a programming language?
10. Which produces faster program execution, a compiler or a pure interpreter and how?

Reading List:
1. Robert W. Sebesta. Concepts of Programming Languages. Addison Wesley, 2011.
2. Aho, A. V., J. E. Hopcroft, and J. D. Ullman. The design and analysis of computer algorithms. Boston: Addison-Wesley, 2007.
3. www.Wikipedia.com

Week 2:

A brief history of programming language with emphasis on Machine language and Assembly language, Evolution of Fortran language, Lisp and pure Lisp, Algol, Simula 67, Ada, C, C++, Java, Prolog, Scripting Language for the Web.

Objective:

The objective is for the student to understand the evolution of various programming languages, compare and contrast and examine the advantages and disadvantages one language has over the other. Student should be able to write simple programs in various languages.

Description:
Conversion of Simple programs from Assembly Language to Machine Language and to various other high level languages. Description of Object Oriented Programming using C++, Java, C# etc.

Lecture Note: Algol, c++, c#

A brief history of programming language with emphasis on Machine language and Assembly language, Evolution of Fortran language, LISP and pure LISP, Algol, simula 67, Ada, C, C++, Java, Prolog, Scripting language for the Web.

A BRIEF HISTORY OF PROGRAMMING LANGUAGES.

- Assembly languages
- IBM 704 and Fortran – FORmula TRANslation
- LISP – LISt Processing
- ALGOL 60 – International Algorithmic language
- Simul 67 – First object oriented language
- Ada – history's largest design effort
- C++ - Combining Imperaive and Object – Oriented Features
- Java – An Imperative – Based Object – Oriented language
- Prolog – Logic Programming

ALGOL

ALGOL 68 (short for **ALGO**rithmic **L**anguage 19**68**) is an imperative computer programming language that was conceived as a successor to the ALGOL 60 programming language, designed with the goal of a much wider scope of application and more rigorously defined syntax and semantics. ALGOL 68 features include expression-based syntax, user-declared types and structures/tagged-unions, a reference model of variables and reference parameters, string and array and matrix slicing and also concurrency.

ALGOL 68 was designed by IFIP Working Group 2.1. On December 20, 1968 the language was formally adopted by Working Group 2.1 and subsequently approved for publication by the General Assembly of IFIP.

ALGOL 68 was defined using a two-level grammar formalism invented by Adriaan van Wijngaarden. Van Wijngaarden grammars use a context-free grammar to generate an infinite set of productions that will recognize a particular ALGOL 68 program; notably, they are able to express the kind of requirements that in many other programming language standards are labelled.

Notable language elements

Bold symbols and reserved words

There are 61 such reserved words (some with "brief symbol" equivalents) in the standard sub-language:

mode, **op**, **prio**, **proc**,
flex, **heap**, **loc**, **long**, **ref**, **short**,
bits, **bool**, **bytes**, **char**, **compl**, **int**, **real**, **sema**, **string**, **void**,
channel, **file**, **format**, **struct**, **union**,
of, **at** "@", **is** ":=:", **isnt** ":/=:", ":≠:", **true**, **false**, **empty**, **nil** "○", **skip** "~",
co comment "¢", **pr**, **pragmat**,
case in ouse *in* out esac "(~ | ~ |: ~ | ~ | ~)",
for from to by while do od,
if then elif *then* else fi "(~ | ~ |: ~ | ~ | ~)",
par begin end "(~)", **go *to***, **goto**, **exit** ".".

mode: Declarations

The basic data types (called **mode**s in ALGOL 68 parlance) are **real**, **int**, **compl** (complex number), **bool**, **char**, **bits** and **bytes**. For example:

int n = 2;
co n is a fixed constant of 2.**co**
int m := 3;
co m is a newly created local variable whose value is initially set to 3.
    This is short for **ref int** m = **loc int** := 3; **co**
real avogadro = 6.0221415□23; **co** Avogadro's number **co**
long long real pi = 3.14159 26535 89793 23846 26433 83279 50288 41971 69399 37510;
compl square root of minus one = 0 ⊥ 1
However, the declaration **real** x; is just syntactic sugar for **ref real** x = **loc real**;. That is, x is really the *constant identifier* for a *reference to* a newly generated local **real** variable.

Special characters

Most of Algol's "special" characters (×, ÷, ≤, ≥, ≠, ¬, ∨, ∧, □, →, ↓, ↑, ◻, ⌊, ⌈, ⎩, ⎧, ○, ⊥ and ¢) can be found on the IBM 2741 keyboard with the APL "golf-ball" print head inserted, these became available in the mid 1960s while ALGOL 68 was being drafted. These characters are also part of the unicode standard and most of them are available in several popular fonts.

Transput: Input and output

Transput is the term used to refer to ALGOL 68's input and output facilities. There are pre-defined procedures for unformatted, formatted and binary transput. Files and other transput devices are handled in a consistent and machine-independent manner.

The following example prints out some unformatted output to the standard output device:

```
 print ((newpage, "Title", newline, "Value of i is ",
   i, "and x[i] is ", x[i], newline))
```

Note the predefined procedures newpage and newline passed as arguments.

C++

C++ (pronounced "see plus plus") is a statically typed, free-form, multi-paradigm, compiled, general-purpose programming language. It is regarded as an intermediate-level language, as it comprises a combination of both high-level and low-level language features. It was developed by Bjarne Stroustrup starting in 1979 at Bell Labs as an enhancement to the C language and originally named C with Classes. It was renamed C++ in 1983.

C++ is one of the most popular programming languages and its application domains include systems software (such as Microsoft Windows), application software, device drivers, embedded software, high-performance server and client applications, and entertainment software such as video games.

C++ is sometimes called a hybrid language; it is possible to write object oriented or procedural code in the same program in C++. This has caused some concern that some C++ programmers are still writing procedural code, but are under the impression that it is object orientated, simply because they are using C++. Often it is an amalgamation of the two. This usually causes most problems when the code is revisited or the task is taken over by another coder.
C++ continues to be used and is one of the preferred programming languages to develop professional applications.

Language features

C++ inherits most of C's syntax. The following is Bjarne Stroustrup's version of the Hello world program that uses the C++ standard library stream facility to write a message to standard output:

```
#include <iostream>

int main()
{
   std::cout << "Hello, world!\n";
}
```

Within functions that define a non-void return type, failure to return a value before control reaches the end of the function results in undefined behaviour (compilers typically provide the means to issue a diagnostic in such a case). The sole exception to this rule is the main function, which implicitly returns a value of zero.

Operators and operator overloading

C++ provides more than 35 operators, covering basic arithmetic, bit manipulation, indirection, comparisons, logical operations and others. Almost all operators can be overloaded for user-defined types, with a few notable exceptions such as member access (. and .*). The rich set of overloadable operators is central to using C++ as a domain-specific language. The overloadable operators are also an essential part of many advanced C++ programming techniques, such as smart pointers. Overloading an operator does not change the precedence of calculations involving the operator, nor does it change the number of operands that the operator uses (any operand may however be ignored by the operator, though it will be evaluated prior to execution). Overloaded "&&" and "||" operators lose their short-circuit evaluation property.

C#

During the development of the .NET Framework, the class libraries were originally written using a managed code compiler system called Simple Managed C (SMC). In January 1999, Anders Hejlsberg formed a team to build a new language at the time called Cool, which stood for "C-like Object Oriented Language". Microsoft had considered keeping the name "Cool" as the final name of the language, but chose not to do so for trademark reasons. By the time the .NET project was publicly announced at the July 2000 Professional Developers Conference, the language had been renamed C#, and the class libraries and ASP.NET runtime had been ported to C#.

Some notable distinguishing features of C# are:

- It has no global variables or functions. All methods and members must be declared within classes. Static members of public classes can substitute for global variables and functions.
- Local variables cannot shadow variables of the enclosing block, unlike C and C++. Variable shadowing is often considered confusing by C++ texts.
- C# supports a strict Boolean data type, bool. Statements that take conditions, such as while and if, require an expression of a type that implements the true operator, such as the boolean type. While C++ also has a boolean type, it can be freely converted to and from integers, and expressions such as if(a) require only that a is convertible to bool, allowing a to be an int, or a pointer. C# disallows this "integer meaning true or false" approach, on the grounds that forcing programmers to use expressions that return exactly bool can prevent certain types of common programming mistakes in C or C++ such as if (a = b) (use of assignment = instead of equality ==).
- In addition to the try...catch construct to handle exceptions, C# has a try...finally construct to guarantee execution of the code in the finally block.
- C# currently (as of version 4.0) has 77 reserved words.
- Multiple inheritances are not supported, although a class can implement any number of interfaces. This was a design decision by the language's lead architect to avoid complication and simplify architectural requirements throughout CLI.

Common Type System (CTS)

C# has a unified type system. This unified type system is called Common Type System (CTS). A unified type system implies that all types, including primitives such as integers, are subclasses of the System.Object class. For example, every type inherits a ToString() method. For performance reasons, primitive types (and value types in general) are internally allocated on the stack.

Categories of data types

CTS separate data types into two categories:

1. **Value types:** they are plain aggregations of data. Instances of value types do not have referential identity nor a referential comparison semantics - equality and inequality comparisons for value types compare the actual data values within the instances, unless the corresponding operators are overloaded. Value types are derived from System. ValueType, always have a default value, and can always be created and copied. Some other limitations on value types are that they cannot derive from each other (but can implement interfaces) and cannot have an explicit default (parameterless) constructor. Examples of value types are all primitive types, such as int (a signed 32-bit integer), float (a 32-bit IEEE floating-point number), char (a 16-bit Unicode code unit), and System.DateTime (identifies a specific point in time with nanosecond precision). Other examples are enum (enumerations) and struct (user defined structures).
2. **Reference types:** they have the notion of referential identity - each instance of a reference type is inherently distinct from every other instance, even if the data within both instances is the same. This is reflected in default equality and inequality comparisons for reference types, which test for referential rather than structural equality, unless the corresponding operators are overloaded (such as the case for System.String). In general, it is not always possible to create an instance of a reference type, nor to copy an existing instance, or perform a value comparison on two existing instances, though specific reference types can provide such services by exposing a public constructor or implementing a corresponding interface (such as ICloneable or IComparable). Examples of reference types are object (the ultimate base class for all other C# classes), System.String (a string of Unicode characters), and System.Array (a base class for all C# arrays). Both type categories are extensible with user-defined types.

Study Questions
1. Make an educated guess as to the most common syntax error in Lisp programs.
2. Describe in detail the three most important reasons, in your opinion, why ALGOL 60 did not become a very used language.
3. Do you think language design committee is a good idea? Support your opinion.
4. Give a brief general description of a markup/programming hybrid language
5. Why in your opinion, do new scripting languages appear more frequently than new compiled languages?
6. Write a program to implement N factorial in; Machine Language, Assembly Language, Scripting Language and any other five high level languages.

Week 3:

What is a programming language? , Implementation method, the compilation process flow chart, Syntax Analysis, Semantic Analysis, Grammars, Syntactic ambiguity, Operator precedence, Top down and Bottom up parsing, recursive descent parsing.

Objective:
The objective of the week lecture is for the student to be able to understand Generative Grammars, Lexical Analysis, Syntactic Analysis and Finite Automata.

Description:
Representation of Finite State Automata (FSA) using transition diagram, An FSA for recognising integer literals, identifiers and reserved words

Lecture Note: Programming Language

A programming language is an artificial language designed to express computations that can be performed by a machine, particularly a computer. Programming languages can be used to create programs that control the behavior of a machine and/or to express algorithms precisely.

A programming language is usually split into the two components of syntax (form) and semantics (meaning). Some languages are defined by a specification document (for example, the C programming language is specified by an ISO Standard), while other languages, such as Perl, have a dominant implementation that is used as a reference.

Elements

All programming languages have some primitive building blocks for the description of data and the processes or transformations applied to them (like the addition of two numbers or the selection of an item from a collection). These primitives are defined by syntactic and semantic rules which describe their structure and meaning respectively.

SYNTAX

A programming language's surface form is known as its syntax. The syntax of a language describes the possible combinations of symbols that form a syntactically correct program. The meaning given to a combination of symbols is handled by semantics (either formal or hard-coded in a reference implementation).

Programming language syntax is usually defined using a combination of regular expressions (for lexical structure) and Backus–Naur Form (for grammatical structure).

Semantics

The term semantics refers to the meaning of languages, as opposed to their form (syntax).

Static semantics

The static semantics defines restrictions on the structure of valid texts that are hard or impossible to express in standard syntactic formalisms.For compiled languages, static semantics essentially include those semantic rules that can be checked at compile time. Examples include checking that every identifier is declared before it is used (in languages that require such declarations) or that the labels on the arms of a case statement are distinct. Many important restrictions of this type, like checking that identifiers are used in the appropriate context (e.g. not adding an integer to a function name), or that subroutine calls have the appropriate number and type of arguments, can be enforced by defining them as rules in a logic called a type system. Other forms of static analyses like data flow analysis may also be part of static semantics. Newer programming languages like Java and C# have definite assignment analysis, a form of data flow analysis, as part of their static semantics.

Dynamic semantics

Once data has been specified, the machine must be instructed to perform operations on the data. For example, the semantics may define the strategy by which expressions are evaluated to values, or the manner in which control structures conditionally execute statements. The dynamic semantics (also known as execution semantics) of a language defines how and when the various constructs of a language should produce a program behaviour. There are many ways of defining execution semantics. Natural language is often used to specify the execution semantics of languages commonly used in practice. A significant amount of academic research went into formal semantics of programming languages, which allow execution semantics to be specified in a formal manner. Results from this field of research have seen limited application to programming language design and implementation outside academia.

Grammar

Below is a simple grammar, based on Lisp:

```
expression ::= atom | list
atom ::= number | symbol
number ::= [+-]?['0'-'9']+
symbol ::= ['A'-'Z''a'-'z'].*
list ::= '(' expression* ')'
```

This grammar specifies the following:

- An expression is either an atom or a list;
- An atom is either a number or a symbol;
- A number is an unbroken sequence of one or more decimal digits, optionally preceded by a plus or minus sign;
- A symbol is a letter followed by zero or more of any characters (excluding whitespace); and
- A list is a matched pair of parentheses, with zero or more expressions inside it.

Syntactic ambiguity

Syntactic ambiguity is a property of sentences which may be reasonably interpreted in more than one way, or reasonably interpreted to mean more than one thing. Ambiguity may or may not involve one word having two parts of speech or homonyms.

Syntactic ambiguity arises not from the range of meanings of single words, but from the relationship between the words and clauses of a sentence, and the sentence structure implied thereby. When a reader can reasonably interpret the same sentence as having more than one possible structure, the text is equivocal and meets the definition of syntactic ambiguity.

Operator Precedence

When several operations occur in an expression, each part is evaluated and resolved in a predetermined order called operator precedence. Parentheses can be used to override the order of precedence and force some parts of an expression to be evaluated before other parts. Operations within parentheses are always performed before those outside. Within parentheses, however, normal operator precedence is maintained.

When expressions contain operators from more than one category, arithmetic operators are evaluated first, comparison operators are evaluated next, and logical operators are evaluated last. Comparison operators all have equal precedence; that is, they are evaluated in the left-to-right order in which they appear. Arithmetic and logical operators are evaluated in the following order of precedence:

Arithmetic	Comparison	Logical
Exponentiation (**^**)	Equality (**=**)	**Not**
Negation (**-**)	Inequality (**<>**)	**And**
Multiplication and division (*****, **/**)	Less than (**<**)	**Or**
Integer division (\)	Greater than (**>**)	**Xor**
Modulus arithmetic (**Mod**)	Less than or equal to (**<=**)	**Eqv**
Addition and subtraction (**+**, **-**)	Greater than or equal to (**>=**)	**Imp**
String concatenation (**&**)	**Is**	**&**

When multiplication and division occur together in an expression, each operation is evaluated as it occurs from left to right. Likewise, when addition and subtraction occur together in an expression, each operation is evaluated in order of appearance from left to right.

The string concatenation operator (**&**) is not an arithmetic operator, but in precedence it does fall after all arithmetic operators and before all comparison operators. The **Is** operator is an object reference comparison operator. It does not compare objects or their values; it checks only to determine if two object references refer to the same object.

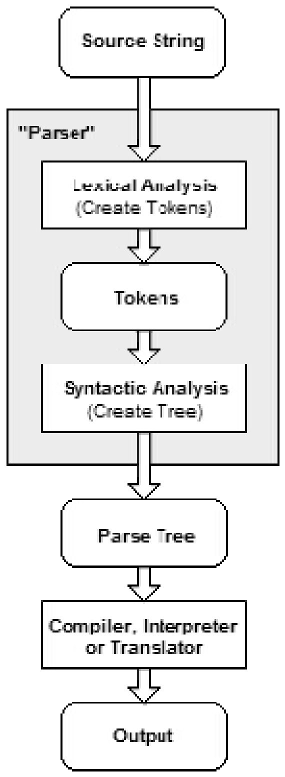
Parsing

In linguistics, parsing is the process of analyzing a text, made of a sequence of tokens (for example, words), to determine its grammatical structure with respect to a given (more or less) formal grammar. Parsing can also be used as a linguistic term, especially in reference to how phrases are divided up in garden path sentences.

Parser

In computing, a parser is one of the components in an interpreter or compiler, which checks for correct syntax and builds a data structure (often some kind of parse tree, abstract syntax tree or other hierarchical structure) implicit in the input tokens. The parser often uses a separate lexical analyser to create tokens from the sequence of input characters. Parsers may be programmed by hand or may be (semi-)automatically generated (in some programming languages) by a tool.

Overview of process



Types of parser

The task of the parser is essentially to determine if and how the input can be derived from the start symbol of the grammar. This can be done in essentially two ways:

- Top-down parsing- Top-down parsing can be viewed as an attempt to find left-most derivations of an input-stream by searching for parse trees using a top-down expansion of the given formal grammar rules. Tokens are consumed from left to right. Inclusive choice is used to accommodate ambiguity by expanding all alternative right-hand-sides of grammar rules. Examples includes: Recursive descent parser, LL parser (Left-to-right, Leftmost derivation), and so on.
- Bottom-up parsing - A parser can start with the input and attempt to rewrite it to the start symbol. Intuitively, the parser attempts to locate the most basic elements, then the elements containing these, and so on. LR parsers are examples of bottom-up parsers. Another term used for this type of parser is Shift-Reduce parsing.

Study questions:
1. Define syntax and semantic
2. What is the primary use of attribute grammars?
3. Define a left recursive grammar
4. The two mathematical models of Language description are generation and recognition. Describe how each can define the syntax of a programming language
5. Write EBNF description for the following:
A Java class definition header statements
A Java method call statement
A C switch statement
A C Union definition
C float literals
6. Prove that the following grammar is ambiguous:
<S> -> <A>
<A> -> <A> + <A>|<id>
<id> -> a|b|c
7. Which of the following sentences are in the language generated by this grammar
Baab, bbbab, bbaaaaa, bbaab

Week 4:

Names, Variables, The concepts of binding, scope, scope and lifetime, Referencing Environments, Named constants

Objective:
To understand the fundamental semantic issues of variables, nature of names and special words in programming languages

Description:
Attributes of variables, including type, address and value will be discussed.

Lecture Note: Variables, scope, and Binding

Variables

It is a symbolic name given to some known or unknown quantity or information, for the purpose of allowing the name to be used independently of the information it represents. A variable name in computer source code is usually associated with a data storage location and thus also its contents.

Compilers have to replace variables' symbolic names with the actual locations of the data. While the variable name, type, and location generally remain fixed, the data stored in the location may get altered during program execution.

Naming conventions

Unlike their mathematical counterparts, programming variables and constants commonly take multiple-character names, e.g. COST or total. Single-character names are most commonly used only for auxiliary variables; for instance, i, j, k for array index variables.

Some naming conventions are enforced at the language level as part of the language syntax and involve the format of valid identifiers. In almost all languages, variable names cannot start with a digit (0-9) and cannot contain whitespace characters. Whether, which, and when punctuation marks are permitted in variable names varies from language to language; many languages only permit the underscore (_) in variable names and forbid all other punctuation. In some programming languages, specific (often punctuation) characters (known as sigils) are prefixed or appended to variable identifiers to indicate the variable's type.

Case-sensitivity of variable names also varies between languages and some languages require the use of a certain case in naming certain entities; most modern languages are case-sensitive; some older languages are not. Some languages reserve certain forms of variable names for their own internal use; in many languages, names beginning with 2 underscores ("__") often fall under this category.

Binding

Binding describes how a variable is created and used (or "bound") by and within the given program, and, possibly, by other programs, as well.

There are two types of binding; Dynamic, and Static binding.

- Dynamic Binding: (Also known as Dynamic Dispatch) is the process of mapping a message to a specific sequence of code (method) at runtime. This is done to support the cases where the appropriate method cannot be determined at compile-time. It occurs first during execution, or can change during execution of the program.
- Static Binding: It occurs first before run time and remains unchanged throughout program execution

Scope

The scope of a variable describes where in a program's text, the variable may be used, while the extent (or lifetime) describes when in a program's execution a variable has a (meaningful) value. Scope is a lexical aspect of a variable. Most languages define a specific scope for each variable (as well as any other named entity), which may differ within a given program. The scope of a variable is the portion of the program code for which the variable's name has meaning and for which the variable is said to be "visible". It is also of two type; static and dynamic scope.

- Static Scope: The static scope of a variable is the most immediately enclosing block, excluding any enclosed blocks where the variable has been re-declared. The static scope of a variable in a program can be determined by simply studying the text of the program. Static scope is not affected by the order in which procedures are called during the execution of the program.
- Dynamic Scope: The dynamic scope of a variable extends to all the procedures called thereafter during program execution, until the first procedure to be called that re-declares the variable.

Referencing

The referencing environment is the collection of variable which can be used. In a static scoped language, one can only reference the variables in the static reference environment.
A function in a static scoped language does have dynamic ancestors (i.e. its callers), but cannot reference any variables declared in that ancestor.

Study questions:
1. Some programming languages are typeless. What are the obvious advantages and disadvantages of having no types in a language?
2. What are the advantages and disadvantages of dynamic scoping?
3. Define static binding and dynamic binding
4. In what ways are reserved words better than keywords?
5. Define lifetime, scope, static scope, and dynamic scope

Week 5:

Primitive data types, character string types, user-defined ordinal types, array types, associative arrays, record types, union types, pointer and reference types, type checking, strong typing, type equivalence, theory and data types.

Objective:
To introduce the concept of a data type and the characteristics of the common primitive data types, designs of enumeration and subrange types will be discussed. Details of structured and subrange types will be discussed. Details of structured data types specifically arrays records and unions are investigated. Indepth look at pointers and references.

Description:

Variables, name, address, value, type, lifetime, scope
Binding = association of attributes with program entities
Type binding, variables based on storage binding
Type checking

Lecture Note: Data Types

Data type is a classification identifying one of various types of data, such as floating-point, integer, or Boolean, that determines the possible values for that type; the operations that can be done on values of that type; and the way values of that type can be stored. There are several classifications of data types, some of which include:

Primitive Data Type

It is a basic data type which is provided by a programming language as a basic building block. Most languages allow more complicated composite types to be recursively constructed starting from basic types. It also a built-in data type for which the programming language provides built-in support.

Character String Types

A string data type is a data type modelled on the idea of a formal string. Strings are such an important and useful data type that they are implemented in nearly every programming language. In some languages they are available as primitive types and in others as composite types. The syntax of most high-level programming languages allows for a string, usually quoted in some way, to represent an instance of a string data type; such a meta-string is called a literal or string literal. A string is traditionally a sequence of characters, either as a literal constant or as some kind of variable.

Arrays

An array is a systematic arrangement of objects, usually in rows and columns. It is a collection of variables of the same type under one name. Each variable, called an element, is accessed using a subscript (or index).

Study questions:
1. What are the design issues of Arrays, Unions and Pointer types?
2. What are the common problems with pointers?
3. Define strongly typed
4. Why is C, C++, and Java not strongly typed?
5. How does a decimal value waste memory space?
6. Explain how coercion rules can weaken the beneficial effect of strong typing
7. Write a program in the language of your choice that behaves differently if the language used name equivalence than if it used structural equivalence

Week 6:

Arithmetic expression, overloaded operators, type conversions, relational and Boolean expressions, short-circuit evaluation, assignment statements, mixed-mode assignment

Objectives:
To encourage the students to understand control flow, arithmetic expressions and design issues in arithmetic expression.

Description:
Type conversions, mixed mode expression, short circuit evaluation, simple assignment statements, assignment with conditional targets, compound assignment operators, urinary assignment operators, list assignment and mixed mode assignment.

Lecture Note: Arithmetic expression

Programming languages generally support a set of operators that are similar to operations in mathematics. A language may contain a fixed number of built-in operators (e.g. + - * = in C and C++), or it may allow the creation of programmer-defined operators (e.g. Haskell). Some programming languages restrict operator symbols to special characters like + or := while others allow also names like div (e.g. Pascal).

Overloaded operators

In some programming languages an operator may be ad-hoc polymorphic, that is, have definitions for more than one kind of data, (such as in Java where the + operator is used both for the addition of numbers and for the concatenation of strings). Such an operator is said to be overloaded. In languages that support operator overloading by the programmer but have a limited set of operators, operator overloading is often used to define customized uses for operators.

Relational and Boolean Expression

Relational operator is a programming language construct or operator that test or define some kind of relation between two entities. These include numerical equality (e.g., $5 = 5$) and inequalities (e.g., $4 \geq 3$). In programming languages that include a distinct Boolean data type in their type system, like Java, these operators return true or false, depending on whether the conditional relationship between the two operands holds or not.

An expression created using a relational operator forms what is known as a relational expression or a condition. Relational operators are also used in technical literature instead of words. Relational operators are usually written in infix notation, if supported by the programming language, which means that they appear between their operands (the two expressions being related).

Short-Circuit Evaluation

Short-circuit evaluation, minimal evaluation, or McCarthy evaluation denotes the semantics of some Boolean operators in some programming languages in which the second argument is only executed or evaluated if the first argument does not suffice to determine the value of the expression: when the first argument of the AND function evaluates to false, the overall value must be false; and when the first argument of the OR function evaluates to true, the overall value must be true. In some programming languages (Lisp), the usual Boolean operators are short - circuit. In others (Java, Ada), both short-circuit and standard Boolean operators are available.

Study questions:

1. Define operator precedence and operator associativity
2. What is Coercion?
3. What is overloaded operator?
4. Describe a situation in which the operator in a programming language would not be i] Associative ii] Commutative
5. Why does Java specify that operands in expressions are all evaluated in left to right order?

Week 7:

Selection Statements, Iterative Statements, Unconditional Branching, Guarded Commands

Objective:
To introduce students to control flow and execution sequence in different programming languages.

Description:
Structured control flow, selection statements, two way selection statements, nested selectors, multiple way selection statements in C/C++/Java/C#/Ada/Python
Iteration Statements, indefinite iteration, Iteration construct in C/C++/Java/C#/Ada/Python
Lecture Note: Selection, and Iterative Statement

Selection Statement

It is a control statement which conditionally chooses between two or more alternate paths of execution, they are also referred to as conditional statements. There are various types of selection statement, such as;

- Two-Way Selection Statements

The basic two-way selection statement is the Boolean if statement; in its simplest form a controlled statement is executed iff a control expression is true: e.g.,

```
  if (x > 4) {
    y = 0;
  }
```

In this example, if x is 5, the value of y is changed to 0; if x is 3, the value of y is not changed.

- Multiple Selection Statements

A multiple selection statement executes one of a list of statement sections, depending on the value of an expression.

The effect of a multi-way selection statement can also be provided with nested if statements, e.g.

```
if (Count == 1)
  cout << "Only one\n";
else if (Count == 2)
  cout << "A pair\n";
else if (Count == 3)
  cout << "Three\n";
else
  cout << "Many\n";
```

Iterative Statement

It is a control statement which conditionally executes a section of code zero or more times. Iteration statements are also referred to as loop statements. It is of various type;

- Counter Controlled Loops

```
FOR I = 1 TO 5 DO BEGIN
    J := J + I
   END
```

- Logically Controlled Loops

```
i = 1;
while (i <= 5) {
 j = j + 1;
 ++i;
}
```

- User-Located Loop Control Mechanisms
- Iteration Based on Data Structures

Unconditional branch statement

This is a statement which always transfers control to a specified location in a program. The most common unconditional branch statement is the go to statement, which in some cases, increase the execution efficiency of a program.

Study questions:
1. What is the definition of control structure?
2. What contemporary languages do not include goto statement?
3. Explain the advantages and disadvantages of the Java "for" statement compared to Ada's "for"
4. Rewrite the following pseudocode segment using a loop structure in the specified languages:

```
K = (j + 13)/27
Loop:
If k > 10 then goto out
K = k + 1
I = 3 * k-1
Goto loop
Out: ...
```

a. Fortran 95
b. Ada
c. C, C++, Java, C#
d. Python
e. Ruby

Week 8:

Fundamentals of subprograms, design issues for subprograms, local referencing environments, parameter-passing methods, parameters that are subprograms, overloaded subprograms, generic subprograms, design issues for functions, user-defined overload operators, subroutines.

Objective:

- To introduce students to subprograms in different programming languages as the fundamental building blocks of programs
- To explore the design concepts including parameter-passing methods, local referencing environment, overload subprograms, generic subprograms and the aliasing and side effect problems.

Description:
Control flow, abstraction, subprogram general declaration, procedure versus functions, formal/actual parameters, local referencing, semantic modes of parameter passing, pass by value, pass by name, pass by result, pass by reference, type checking, overload, polymorphisms, subroutines.

Lecture Note: Fundamentals of subprograms.

Subprogram

Subroutine (also called procedure, function, routine, method, or subprogram) is a portion of code within a larger program that performs a specific task and is relatively independent of the remaining code.

A subroutine is often coded so that it can be started ("called") several times and/or from several places during a single execution of the program, including from other subroutines, and then branch back (return) to the next instruction after the "call" once the subroutine's task is done.

A subroutine may be written so that it expects to obtain one or more data values from the calling program (its parameters or arguments). It may also return a computed value to its caller (its return value), or provide various result values or out(put) parameters. Indeed, a common use of subroutines is to implement mathematical functions, in which the purpose of the subroutine is purely to compute one or more results whose values are entirely determined by the parameters passed to the subroutine. (Examples might include computing the logarithm of a number or the determinant of a matrix.)

The advantages of breaking a program into subroutines include:

- Decomposition of a complex programming task into simpler steps: this is one of the two main tools of structured programming, along with data structures.
- reducing the duplication of code within a program,
- enabling the reuse of code across multiple programs,
- Hiding implementation details from users of the subroutine.

Disadvantages

- The invocation of a subroutine (rather than using in-line code) imposes some computational overhead in the call mechanism itself.
- The subroutine typically requires standard housekeeping code—both at entry to, and exit from, the function (function prologue and epilogue—usually saving general purpose registers and return address as a minimum).

Parameter –passing Methods

Parameter is a special kind of variable, used in a subroutine to refer to one of the pieces of data provided as input to the subroutine. These pieces of data are called arguments. An ordered list of parameters is usually included in the definition of a subroutine, so that, each time the subroutine is called, its arguments for that call can be assigned to the corresponding parameters.

Study questions:
1. What are the design issues for subprograms?
2. What are the three semantic models of parameter passing?
3. What are the design issues for functions?
4. In most Fortran IV implementations, parameters were passed by reference, using access path transmission only. State both the advantages and disadvantages of this design choice
5. Write a program, using the syntax of whatever language you like that produces different behaviour depending on whether pass by reference or pass by value result is used in its parameter passing.

Week 9:

Concept of Abstraction, Introduction to data abstraction, design issues for abstract data types, language examples, parameterized abstract data types, encapsulations.

Objective:
- To explore programming language constructs that support data abstraction and
- To discuss the general concept of abstraction in programming and programming languages.

Description:
Functional versus imperative, mathematical functions, lambda expressions, functional forms, syntax and naming conventions, simple expressions, procedures that return procedures, functional forms in scheme, the Fibonacci numbers, the sum of two infinite lists.

Study questions:
1. Compare Java's package with Ruby's modules
2. Describe the three ways a client can reference a name from a namespace in C++
3. Design an abstract data type for a matrix with integer elements in a language that you know, including operations for addition, subtraction, and matrix multiplication.

Week 10:

Object Oriented programming, Design issues for object oriented language, smalltalk, C++, Java, C3, ada 95, Ruby, Implementation of Object Oriented constructs, Scripting Language i.e Python.

Objective:
To introduce Object Oriented programming followed by an extended discussion of the primary design issues for inheritance and dynamic binding, support for object oriented

programming in smallytalk, c++, Java, c#, Ada 95 and Ruby. Implementation of dynamic bindings of method calls to methods in Object Oriented language.

Description:
Important features of O-O language, the python example: interpreter, keywords, modules, class, inheritance, built-in types basic and composite, integers, Booleans, floating point, strings, lists, tuples, dictionaries, files, statement and functions, assignment forms, compound statements, conditional statement, error handler, exception handling.

Study questions:
1. Describe the three characteristics features of Object Oriented Language
2. What is?
a. Multiple Inheritance
b. Polymorphic variables,
c. Nesting class
3. Compare the class entity control of C++ and Java
4. Explain type checking in smalltalk
5. What is the purpose of finalize clause in Java
6. Compare the type error detection for instance variables in Java and Ruby.

Week 11:
Predicate Calculus, Influence Rules, Logic programming Languages, Element of Prolog, applications of logic programming.

Objective:
The objectives are to introduce the concepts of logic programming and logic programming languages, including a brief description of a subset of prolog.

Description:
Introduction to predicative calculus as the basis for logic programming language; followed by discussion of how predicative calculus can be used for automatic theorem-proving systems, general overview of logic programming, basic of prolog programming language, including arithmetic list processing, and a trace tool that can be used to help debug programs, illustration of how the prolog system works, application areas in which prolog has been used.

Lecture Note: PROLOG
Logic Programming is the name given to a distinctive style of programming, very different from that of conventional programming languages such as C++ and Java. Fans of Logic Programming would say that 'different' means clearer, simpler and generally better! Although there are other Logic Programming languages, by far the most widely used is Prolog. The name stands for Programming in Logic. Prolog is based on research by computer scientists in Europe in the 1960s and 1970s, notably at the Universities of Marseilles, London and Edinburgh. The first implementation was at the University of Marseilles in the early 1970s. Further development at the University of Edinburgh led to a de facto standard version, now known as Edinburgh Prolog. Prolog has been widely used for developing complex applications, especially in the field of Artificial Intelligence. Although it is a general-purpose language, its main strengths are for symbolic rather than for numerical computation. The

developers of the language were researchers working on automating mathematical theorem proving. This field is often known as computational logic. The idea that the methods developed by computational logicians could be used as the basis for a powerful general purpose programming language was revolutionary 40 years ago. Unfortunately most other programming languages have not yet caught up. Developed in Europe in the 1970s, the language Prolog has steadily gained enthusiastic converts, bolstered by its surprise choice as the initial language of the Japanese Fifth Generation Computer project. Prolog has three positive features that give it key advantages over Lisp. First, Prolog syntax and semantics are much closer to formal logic, the most common way of representing facts and reasoning methods used in the artificial intelligence literature. Second, Prolog provides automatic backtracking, a feature making for considerably easier "search", the most central of all artificial intelligence techniques. Third, Prolog supports multidirectional (or multiuse) reasoning, in which arguments to a procedure can freely be designated inputs and outputs in different ways in different procedure calls, so that the same procedure definition can be used for many different kinds of reasoning. Besides this, new implementation techniques have given current versions
of Prolog close in speed to Lisp implementations, so efficiency is no longer a reason to prefer Lisp.

Prolog has a straightforward uniform syntax, programs that are equivalent to a database of facts and rules, a built-in theorem prover with *automatic backtracking, list processing, recursion and facilities for modifying programs (or databases) at run-time*. Prolog lends itself to a style of programming making particular use of two powerful techniques: recursion and list processing. In many cases algorithms that would require substantial amounts of coding in other languages can be implemented in a few lines in Prolog.

Available versions of PROLOG

There are many versions of Prolog available for PC, Macintosh and Unix systems, including versions for Microsoft Windows, to link Prolog to an Oracle relational database and for use with 'object-oriented' program design. These range from commercial systems with many features to public domain and 'freeware' versions. Some of these are listed (in alphabetical order) below, together with web addresses at which more information can be found.

1. Amzi! Prolog ; http://www.amzi.com/products/prolog_products.htm
2. B-Prolog -http://www.probp.com/
3. Ciao Prolog - http://clip.dia.fi.upm.es/Software/Ciao/
4. GNU Prolog - http://gnu-prolog.inria.fr/
5. Logic Programming Associates Prolog (versions for Windows, DOS and Macintosh) -
6. http://www.lpa.co.uk
7. Open Prolog (for Apple Macintosh) - http://www.cs.tcd.ie/open-prolog/
8. PD Prolog (a public domain version for MS-DOS only) - http://www-2.cs.cmu.edu/afs/cs/project/ ai-repository/ai/lang/prolog/impl/prolog/pdprolog/0.html
9. SICStus Prolog - http://www.sics.se/isl/sicstuswww/site/index.html
10. SWI Prolog - http://www.swi-prolog.org/

11. Turbo Prolog (an old version that only runs in MS-DOS) - http://www.fraber.de/university/prolog/tprolog.html
12. Visual Prolog - http://www.visual-prolog.com/
13. W-Prolog (a Prolog-like language that runs in a web browser) - http://goanna.cs.rmit.edu.au/~winikoff/wp/
14. YAP Prolog - http://www.ncc.up.pt/~vsc/Yap/

Starting PROLOG

Starting the Prolog system is usually straightforward, but the precise details will vary from one version to another. Consult the documentation if necessary. Starting Prolog will generally produce a number of lines of headings followed by a line containing just

?-

This is the system prompt. (In some versions of Prolog a slightly different combination of characters may be used.)

The prompt indicates that the Prolog system is ready for the user to enter a sequence of one or more goals, which must be terminated by a full stop, for example:

```
?- write('Hello World'),nl,write('Welcome to Prolog'),nl.
```

The above line does not have any effect until the 'return' key is pressed. Doing so produces the output

```
Hello World
Welcome to Prolog
yes
```

followed by a further system prompt

?-.

In the above example, the user has entered a sequence of four goals:

write('Hello World'), nl (twice) and write('Welcome to Prolog').

The commas separating the goals signify 'and'. In order for the sequence of goals

write('Hello World'),nl,write('Welcome to Prolog'),nl

to succeed each of the following goals has to succeed in order:

write('Hello World')

Hello World has to be displayed on the user's screen

nl

a new line has to be output to the user's screen

write('Welcome to Prolog')

Welcome to Prolog has to be displayed on the user's screen

nl

A new line has to be output to the user's screen.

The Prolog system can achieve all these goals simply by outputting lines of text to the user's screen. It does so and then outputs **yes** to indicate that the sequence of goals has succeeded.

From the system's point of view, the important issue is whether or not the sequence of goals entered by the user succeeds. The generation of output to the screen is considered much less important and is described as (merely) a side effect of evaluating the goals write('Hello World') etc. The meanings of write and nl are pre-defined by the Prolog system. They are known as built-in predicates, sometimes abbreviated to BIPs. Two other built-in predicates that are provided as standard in almost all versions of Prolog are halt and statistics.

```
?-halt.
```

causes the Prolog system to terminate.

```
?- statistics.
```

causes system statistics (of value mainly to more experienced users) such as the following to be generated.

```
Memory Statistics   Free Bytes   Total Bytes   Percent
Reset Space         65536        65536         (100%)
Heap Space          261850       262130        (100%)
Input Space         65536        65536         (100%)
Output Space        65536        65536         (100%)
Total Elapsed Time  626.871      (100%)
Active Processing   2.704        ( 0%)
Waiting for Input   623.977      (100%)
yes
```

Note that this output ends with the word yes, signifying that the goal has succeeded, as statistics, halt and many other built-in predicates always do. Their value lies in the side effects (generating statistics etc.) produced when they are evaluated.

A sequence of one or more goals entered by the user at the prompt is often called a query. I may use the two terms interchangeably in this Term Paper.

Prolog Programs

Entering a goal or a sequence of goals at the system prompt using only built-in predicates would be of little value in itself. The normal way of working is for the user to load a program written in the Prolog language and then enter a sequence of one or more goals at the prompt, or possibly several sequences in succession, to make use of the information that has been loaded into the database. The simplest (and most usual) way to create a Prolog program is to type it into a text editor and save it as a text file, say prog1.pl.

This is a simple example of a Prolog program. It has three components, known as clauses, each terminated by a full stop. Note the use of blank lines to improve readability – they are ignored.

```
dog(fido).
cat(felix).
animal(X):-dog(X).
```

The program can then be loaded for use by the Prolog system using the built-in predicate consult.

```
?-consult('prog1.pl').
```

Provided that the file prog1.pl exists and the program is syntactically correct, i.e. contains valid clauses, the goal will succeed and as a side effect produce one or more lines of output to confirm that the program has been read correctly, e.g.

```
?-
# 0.00 seconds to consult prog1.pl
?-
```

Loading a program simply causes the clauses to be placed in a storage area called the Prolog database. Entering a sequence of one or more goals in response to the system prompt causes Prolog to search for and use the clauses necessary to evaluate the goal(s). Once placed in the database the clauses generally remain there until the user exits from the Prolog system and so can be used to evaluate further goals entered by the user.
In the program above the three lines are all clauses. Each clause is terminated by a full stop. Apart from comments and blank lines, Prolog programs consist only of a sequence of clauses. All clauses are either facts or rules.

dog(fido) and cat(felix) are examples of facts. They can be interpreted in a natural way as meaning 'fido is a dog' and 'felix is a cat'.

dog is called a predicate.
It has one argument, the word **fido** enclosed in ( ).
fido is called an atom (meaning a constant which is not a number).

The final line of the program

```
animal(X):-dog(X).
```

is a rule. The **:-** character (colon and hyphen) can be read as '**if**'.
X is called a variable which in this context, represents any value, as long as it is the same value both times.
The rule can be read in a natural way as

X is an animal if X is a dog (for any X).

From the above clauses it is simple (for humans) to deduce that fido is an animal. Prolog can also make such deductions:

```
?- animal(fido).
yes
```

However there is no evidence to imply that felix is an animal:

```
?- animal(felix).
no
```

Data Objects in Prolog: Prolog Terms

The data objects in Prolog are called terms. Examples of terms that have been used in Prolog programs so far in this term paper are **fido**, **dog(henry)**, **X** and **cat(X)**. There are several different types of term, which are listed below.

(1) Numbers

All versions of Prolog allow the use of integers (whole numbers). They are written as any sequence of numerals from 0 to 9, optionally preceded by a + or - sign, for example:

623
-47
+5
025

Most versions of Prolog also allow the use of numbers with decimal points. They are written in the same way as integers, but contain a single decimal point, anywhere except before an optional + or - sign, e.g.

6.43
-.245
+256.

(2) Atoms

Atoms are constants that do not have numerical values. There are three ways in which atoms can be written.

(a) Any sequence of one or more letters (upper or lower case), numerals and underscores, beginning with a lower case letter, e.g.

john
today_is_Tuesday
fred_jones
a32_BCD
but not
Today
today-is-Tuesday
32abc

(b) Any sequence of characters enclosed in single quotes, including spaces and upper case letters, e.g.

'Today is Tuesday'
'today-is-Tuesday'
'32abc'

(c) Any sequence of one or more special characters from a list that includes the following + - * / > < = & # @ :

Examples

+++
>=
>
+--

(3) Variables

In a query a variable is a name used to stand for a term that is to be determined, e.g. variable X may stand for atom dog, the number 12.3, or a compound term or a list (both to be described below). The name of a variable is denoted by any sequence of one or more letters (upper or lower case), numerals and underscores, beginning with an upper case letter or underscore, e.g.

X
Author
Person_A

_123A

but not

45_ABC
Person-A
author

Note: The variable _ which consists of just a single underscore is known as the anonymous variable and is reserved for a special purpose.

(4) Compound Terms

Compound terms are of fundamental importance in writing Prolog programs. A compound term is a structured data type that begins with an atom, known here as a functor. The functor is followed by a sequence of one or more arguments, which are enclosed in brackets and separated by commas. The general form is

functor(t1,t2, … ,tn) n≥1

If you are familiar with other programming languages, you may find it helpful to think of a compound term as representing a record structure. The functor represents the name of the record, while the arguments represent the record fields.
The number of arguments a compound term has is called its **arity**. Some examples of compound terms are:

likes(paul,prolog)
read(X)
dog(henry)
cat(X)
>(3,2)
person('john smith',32,doctor,london)

Each argument of a compound term must be a term, which can be of any kind including a compound term. Thus some more complex examples of compound terms are:

likes(dog(henry),Y)
pred3(alpha,beta,gamma,Q)
pred(A,B,likes(X,Y),-4,pred2(3,pred3(alpha,beta,gamma,Q)))

(5) Lists

A list is at times is considered to be a special type of compound term and at times as a separate type of data object. Lists are written as an unlimited number of arguments (known as list elements) enclosed in square brackets and separated by commas, e.g. [dog,cat,fish,man]. Unlike the arity of a compound term, the number of elements a list has does not have to be decided in advance when a program is written. This can be extremely useful.

An element of a list may be a term of any kind, including a compound term or another list, e.g.

[dog,cat,y,mypred(A,b,c),[p,q,R],z]
[[john,28],[mary,56,teacher],robert,parent(victoria,albert),[a,b,[c,d,e],f],29]
[[portsmouth,edinburgh,london,dover],[portsmouth,london,edinburgh],[glasgow]]

A list with no elements is known as the empty list. It is written as [].

(6) Other Types of Term

Some dialects of Prolog allow other types of term, e.g. character strings. However, it is possible to use atoms to perform a rudimentary type of string processing. Atoms and compound terms have a special importance in Prolog clauses and are known collectively as call terms.

Clauses and Predicates

Clauses

Apart from comments and blank lines, which are ignored, a Prolog program consists of a succession of clauses. A clause can run over more than one line or there may be several on the same line. A clause is terminated by a dot character, followed by at least one 'white space' character, e.g. a space or a carriage return. There are two types of clause: **facts and rules**.
Facts are of the form
head.
head is called the *head of the clause*. It takes the same form as a goal entered by the user at the prompt, i.e. it must be an atom or a compound term. Atoms and compound terms are known collectively as call terms.
Some examples of facts are:

```
christmas.
likes(john,mary).
likes(X,prolog).
dog(fido).
```

Rules are of the form:
head:-t1,t2, … , tk. (k>=1)
head is called the *head of the clause (or the head of the rule)* and, as for facts, must be a call term, i.e. an atom or a compound term.
:- is called the *neck of the clause (or the 'neck operator')*. It is read as *'if'*.
t1,t2, … , tk is called the *body of the clause (or the body of the rule)*. It specifies the conditions that must be met in order for the conclusion, represented by the head, to be satisfied. The body consists of one or more components, separated by commas. The components are goals and the commas are read as 'and'. Each goal must be a call term, i.e. an atom or a compound term. A rule can be read as

'head is true if t1, t2, …, tk are all true'.

The head of a rule can also be viewed as a goal with the components of its body viewed as subgoals. Thus another reading of a rule is 'to achieve goal head, it is necessary to achieve subgoals t1, t2,…, tk in turn'.
Some examples of rules are:

```
large_animal(X):-animal(X),large(X).
grandparent(X,Y):-father(X,Z),parent(Z,Y).
go:-write('hello world'),nl.
```

Here is another version of the animals program, which includes both facts and

rules.
/* Animals Program 2 */

```
    dog(fido).
    large(fido).
    cat(mary).
    large(mary).
    dog(rover).
    dog(jane).
    dog(tom).
    large(tom).
    cat(harry).
    dog(fred).
    dog(henry).
    cat(bill).
    cat(steve).
    small(henry).
    large(fred).
    large(steve).
    large(jim).
    large(mike).
    large_animal(X):- dog(X),large(X).
    large_animal(Z):- cat(Z),large(Z).
```

fido, mary, jane etc. are atoms, i.e. constants, indicated by their initial lower case letters. X and Y are variables, indicated by their initial capital letters. The first 18 clauses are facts. The final two clauses are rules.

Predicates

The following simple program has five clauses. For each of the first three clauses, the head is a compound term with functor parent and arity 2 (i.e. two arguments).

```
    parent(victoria,albert).
    parent(X,Y):-father(X,Y).
    parent(X,Y):-mother(X,Y).
    father(john,henry).
    mother(jane,henry).
```

It is possible (although likely to cause confusion) for the program also to include clauses for which the head has functor parent, but a different arity, for example

```
    parent(john).
    parent(X):-son(X,Y).
```

/* X is a parent if X has a son Y */
It is also possible for parent to be used as an atom in the same program, for example in the fact

```
    animal(parent).
```

but this too is likely to cause confusion.

All the clauses (facts and rules) for which the head has a given combination of functor and arity comprise a definition of a predicate. The clauses do not have to appear as consecutive lines of a program but it makes programs easier to read if they do.

The clauses given above define two predicates with the name parent, one with arity two and the other with arity one. These can be written (in textbooks, reference manuals etc., not in programs) as parent/2 and parent/1, to distinguish between them. When there is no risk of ambiguity, it is customary to refer to a predicate as just dog, large_animal etc.
An atom appearing as a fact or as the head of a rule, e.g.

```
christmas.
go:-parent(john,B),
write('john has a child named '),
write(B),nl.
```

can be regarded as a predicate with no arguments, e.g. go/0.
There are five predicates defined in Animals Program 2: dog/1, cat/1, large/1, small/1 and large_animal/1. The first 18 clauses are facts defining the predicates dog/1, cat/1, large/1 and small/1 (6, 4, 7 and 1 clauses, respectively). The final two clauses are rules, which together define the predicate large_animal/1.

Declarative and Procedural Interpretations of Rules

Rules have both a declarative and a procedural interpretation. For example, the declarative interpretation of the rule

```
chases(X,Y):-dog(X),cat(Y),write(X),
write(' chases '),write(Y),nl.
```

is:
'chases(X,Y) is true if dog(X) is true and cat(Y) is true and write(X) is true, etc.'
The procedural interpretation is
'To satisfy chases(X,Y), first satisfy dog(X), then satisfy cat(Y), then satisfy write(X), etc.'
Facts are generally interpreted declaratively, e.g.

```
dog(fido).
```

is read as *'fido is a dog'*.
The order of the clauses defining a predicate and the order of the goals in the body of each rule are irrelevant to the declarative interpretation but of vital importance to the procedural interpretation and thus to determining whether or not the sequence of goals entered by the user at the system prompt is satisfied. When evaluating a goal, the clauses in the database are examined from top to bottom. Where necessary, the goals in the body of a rule are examined from left to right.
A user's program comprises facts and rules that define new predicates. These are called user-defined predicates. In addition there are standard predicates predefined by the Prolog system. These are known as built-in predicates (BIPs) and may not be redefined by a user program. Some examples are: write/1, nl/0, repeat/0, member/2, append/3, consult/1, halt/0. Some BIPs are common to all versions of Prolog. Others are version-dependent. Two of the most commonly used built-in predicates are write/1 and nl/0. The write/1 predicate takes a term as its argument, e.g.

```
write(hello)
write(X)
write('hello world')
```

Providing its argument is a valid term, the write predicate always succeeds and as a side effect writes the value of the term to the user's screen. To be more precise it is output to the current output stream, which by default will be assumed to be the user's screen. If the argument is a quoted atom, e.g. 'hello world', the quotes are not output.

The nl/0 predicate is an atom, i.e. a predicate that takes no arguments. The predicate always succeeds and as a side effect starts a new line on the user's screen.
The name of a user-defined predicate (the functor) can be any atom, with a few exceptions, except that you may not redefine any of the Prolog system's built-in predicates. You are most unlikely to want to redefine the write/1 predicate by putting a clause such as

write(27).
or
write(X):-dog(X).

in your programs, but if you do the system will give an error message such as 'illegal attempt to redefine a built-in predicate'.
It would be permitted to define a predicate with the same functor and a different arity, e.g. write/3 but this is definitely best avoided.

Simplifying Entry of Goals
In developing or testing programs it can be tedious to enter repeatedly at the system prompt a lengthy sequence of goals such as

?-dog(X),large(X),write(X),write(' is a large dog'),nl.

A commonly used programming technique is to define a predicate such as go/0 or start/0, with the above sequence of goals as the right-hand side of a rule, e.g.

go:-dog(X),large(X),write(X),
write(' is a large dog'),nl.

This enables goals entered at the prompt to be kept brief, e.g.

?-go.

Recursion
An important technique for defining predicates, which will be used frequently later in this term paper is to define them in terms of themselves. This is known as a recursive definition. There are two forms of recursion.
(a) Direct recursion. Predicate pred1 is defined in terms of itself.
(b) Indirect recursion. Predicate pred1 is defined using pred2, which is defined using pred3, …, which is defined using pred1.
The first form is more common. An example of it is

likes(john,X):-likes(X,Y),dog(Y).

which can be interpreted as 'john likes anyone who likes at least one dog'.

Predicates and Functions
The use of the term 'predicate' in Prolog is closely related to its use in mathematics. A predicate can be thought of as a relationship between a number of values (its arguments) such as likes(henry,mary) or X=Y, which can be either true or false. This contrasts with a function, such as 6+4, the square root of 64 or the first three characters of 'hello world', which can evaluate to a number, a string of characters or some other value as well as true and false. Prolog does not make use of functions except in arithmetic expressions.

Operators and Arithmetic

Operators

Up to now, the notation used for predicates in this term paper is the standard one of a functor followed by a number of arguments in parentheses, e.g.

likes(john,mary).

As an alternative, any user-defined predicate with two arguments (a binary predicate) can be converted to an infix operator. This enables the functor (predicate name) to be written between the two arguments with no parentheses, e.g.

john likes mary

Some Prolog users may find this easier to read. Others may prefer the standard notation.

Any user-defined predicate with one argument (a unary predicate) can be converted to a prefix operator. This enables the functor to be written before the argument with no parentheses, e.g.

isa_dog fred

instead of

isa_dog(fred)

Alternatively, a unary predicate can be converted to a postfix operator. This enables the functor to be written after the argument, e.g.

fred isa_dog

Operator notation can also be used with rules to aid readability. Some people may find a rule such as

likes(john,X):-is_female(X),owns(X,Y),isa_cat(Y).

easier to understand if it is written as

john likes X:- X is_female, X owns Y, Y isa_cat.

is a valid form of the previous rule.

Any user-defined predicate with one or two arguments can be converted to an operator by entering a goal using the op predicate at the system prompt. This predicate takes three arguments, for example

?-op(150,xfy,likes).

The first argument is the 'operator precedence', which is an integer from 0 upwards. The range of numbers used depends on the particular implementation. The lower the number, the higher the precedence. Operator precedence values are used to determine the order in which operators will be applied when more than one is used in a term. The most important practical use of this is for operators used for arithmetic, as will be explained later. In most other cases it will suffice to use an arbitrary value such as 150. The second argument should normally be one of the following three atoms:

xfy meaning that the predicate is binary and is to be converted to an infix operator

fy meaning that the predicate is unary and is to be converted to an prefix operator

xf meaning that the predicate is unary and is to be converted to a postfix operator

The third argument specifies the name of the predicate that is to be converted to an operator.

A predicate can also be converted to an operator by placing a line such as

?-op(150,xfy,likes).

in a Prolog program file to be loaded using consult or reconsult. Note that the prompt (the two characters ?-) must be included. When a goal is used in this way, the entire line is known as a directive. In this case, the directive must be placed in the file before the first clause that uses the operator likes. Several built-in predicates have been pre-defined as operators. These include

relational operators for comparing numerical values, including

< denoting 'less than' and

> denoting 'greater than'.

Thus the following are valid terms, which may be included in the body of a rule:

```
X>4
Y<Z
A=B
```

Bracketed notation may also be used with built-in predicates that are defined as operators, e.g.

>(X,4) instead of X>4.

Arithmetic

Although the examples used in previous sections of this term paper are non-numerical (animals which are mammals etc.), Prolog also provides facilities for doing arithmetic using a notation similar to that which will already be familiar to many users from basic algebra.

This is achieved using the built-in predicate **is/2**, which is predefined as an infix operator and thus is written between its two arguments.

The most common way of using is/2 is where the first argument is an unbound variable. Evaluating the goal

X is –6.5

will cause X to be bound to the number –6.5 and the goal to succeed.

The second argument can be either a number or an arithmetic expression e.g.

X is 6*Y+Z-3.2+P-Q/4 (* denotes multiplication).

Any variables appearing in an arithmetic expression must already be bound (as a result of evaluating a previous goal) and their values must be numerical. Provided they are, the goal will always succeed and the variable that forms the first argument will be bound to the value of the arithmetic expression. If not, an error message will result.

```
?- X is 10.5+4.7*2.
X = 19.9
?- Y is 10,Z is Y+1.
Y = 10 ,
Z = 11
```

Symbols such as + - * / in arithmetic expressions are a special type of infix operator known as arithmetic operators. Unlike operators used elsewhere in Prolog they are not predicates but functions, which return a numerical value.

As well as numbers, variables and operators, arithmetic expressions can include arithmetic functions, written with their arguments in parentheses (i.e. not as operators). Like arithmetic operators these return numerical values, e.g. to find the square root of 36:

```
?- X is sqrt(36).
X = 6
```

The arithmetic operator - can be used not only as a binary infix operator to denote the difference of two numerical values, e.g. X-6, but also as a unary prefix operator to denote the negative of a numerical value, e.g.

```
?- X is 10,Y is -X-2.
X = 10 ,
Y = -12
```

The table below shows some of the arithmetic operators and arithmetic functions available in Prolog.

X+Y	the sum of X and Y
X-Y	the difference of X and Y
X*Y	the product of X and Y
X/Y	the quotient of X and Y
X//Y	the 'integer quotient' of X and Y (the result is truncated to the nearest integer between it and zero)
X^Y	X to the power of Y
-X	the negative of X
abs(X)	the absolute value of X
sin(X)	the sine of X (for X measured in degrees)
cos(X)	the cosine of X (for X measured in degrees)
max(X,Y)	the larger of X and Y
sqrt(X)	the square root of X

Example

```
?- X is 30,Y is 5,Z is X+Y+X*Y+sin(X).
X = 30 ,
Y = 5 ,
Z = 185.5
```

Although the is predicate is normally used in the way described here, the first argument can also be a number or a bound variable with a numerical value. In this case, the numerical values of the two arguments are calculated. The goal succeeds if these are equal. If not, it fails.

```
?- X is 7,X is 6+1.
X = 7
?- 10 is 7+13-11+9.
no
?- 18 is 7+13-11+9.
yes
```

Operator Precedence in Arithmetic Expressions

When there is more than one operator in an arithmetic expression, e.g.

A+B*C-D

Prolog needs a means of deciding the order in which the operators will be applied. For the basic operators such as + - * and / it is highly desirable that this is the customary 'mathematical' order, i.e. the expression A+B*C-D should be interpreted as 'calculate the product of B and C, add it to A and then subtract D', not as 'add A and B, then multiply by C and subtract D'. Prolog achieves this by giving each operator a numerical precedence value. Operators with relatively high precedence such as * and / are applied before those with lower precedence such as + and -. Operators with the same precedence (e.g. + and -, * and /) are applied from left to right. The effect is to give an expression such as A+B*C-D the meaning that a user who is familiar with algebra would expect it to have, i.e. A+(B*C)-D. If a different order of evaluation is required this can be achieved by the use of brackets, e.g.

X is (A+B)*(C-D).

Bracketed expressions are always evaluated first.

Relational Operators

The infix operators =:= =\= > >= < =< are a special type known as relational operators. They are used to compare the value of two arithmetic expressions. The goal succeeds if the value of the first expression is equal to, not equal to, greater than, greater than or equal to, less than or less than or equal to the value of the second expression, respectively. Both arguments must be numbers, bound variables or arithmetic expressions (in which any variables are bound to numerical values).

```
?- 88+15-3=:=110-5*2.
yes
?- 100=\=99.
yes
```

Equality Operators

There are three types of relational operator for testing equality and inequality available in Prolog. The first type is used to compare the values of arithmetic expressions. The other two types are used to compare terms.

Arithmetic Expression Equality **=:=**

E1=:=E2 succeeds if the arithmetic expressions E1 and E2 evaluate to the same value.

```
?- 6+4=:=6*3-8.
yes
?- sqrt(36)+4=:=5*11-45.
yes
```

To check whether an integer is odd or even we can use the checkeven/1 predicate defined below.

```
checkeven(N):-M is N//2,N=:=2*M.
?- checkeven(12).
yes
?- checkeven(23).
no
?- checkeven(-11).
no
?- checkeven(-30).
yes
```

The integer quotient operator // divides its first argument by its second and truncates the result to the nearest integer between it and zero. So 12//2 is 6, 23//2 is 11, -11//2 is -5 and -30//2 is -15. Dividing an integer by 2 using // and multiplying it by 2 again will give the original integer if it is even, but not otherwise.

Arithmetic Expression Inequality =\=

E1=\=E2 succeeds if the arithmetic expressions E1 and E2 do not evaluate to the same value

```
?- 10=\=8+3.
yes
```

Terms Identical ==

Both arguments of the infix operator == must be terms. The goal Term1==Term2 succeeds if and only if Term1 is identical to Term2. Any variables used in the terms may or may not already be bound, but no variables are bound as a result of evaluating the goal.

```
?- likes(X,prolog)==likes(X,prolog).
X = _
```

```
?- likes(X,prolog)==likes(Y,prolog).
no
(X and Y are different variables)
?- X is 10,pred1(X)==pred1(10).
X = 10
?- X==0.
no
?- 6+4==3+7.
no
```

The value of an arithmetic expression is only evaluated when used with the is/2 operator. Here 6+4 is simply a term with functor + and arguments 6 and 4. This is entirely different from the term 3+7.

Terms Not Identical \==

Term1\==Term2 tests whether Term1 is not identical to Term2. The goal succeeds if Term1==Term2 fails. Otherwise it fails.

```
?- pred1(X)\==pred1(Y).
X = _ ,
Y = _
```

Logical Operators

This section gives a brief description of two operators that take arguments that are call terms, i.e. terms that can be regarded as goals.

The not Operator

The prefix operator not/1 can be placed before any goal to give its negation. The negated goal succeeds if the original goal fails and fails if the original goal succeeds.
The following examples illustrate the use of not/1. It is assumed that the database contains the single clause

```
dog(fido).
?- not dog(fido).
no
?- dog(fred).
no
?- not dog(fred).
yes
?- X=0,X is 0.
X = 0
?- X=0,not X is 0.
no
```

The Disjunction Operator

The disjunction operator ;/2 (written as a semicolon character) is used to represent 'or'. It is an infix operator that takes two arguments, both of which are goals.

Goal1;Goal2 succeeds if either Goal1 or Goal2 succeeds.

```
?- 6<3;7 is 5+2.
yes
?- 6*6=:=36;10=8+3.
yes
```

Study questions:
1. Explain the difference between procedural and non-procedural languages
2. What is the relationship between resolution and unification in prolog
3. Compare the concept of data typing in Ada with that of Prolog
4. Write a prolog program that finds the maximum of a list of numbers
5. Write a Prolog program that implement quicksort

Week 12:

Revisions and Examinations

Objective:
The objective of the week lecture is for the student to be able to revise all they have been taught so far.

Description:
All the objectives for the course should be seriously overviewed