SABANCI UNIVERSITY

FACULTY OF ENGINEERING AND NATURAL SCIENCES

CS305 – Programming Languages

Lecture Notes

compiled on March 8, 2023

Hüsnü Yenigün

Contents

Chapter 1

Introduction to the course

1.1 Textbook and reading list

[1] "Programming Languages: Concepts and Constructs" by Ravi Sethi

[2] "Concepts of Programmnig Languages" by Robert W. Sebesta

[3] "Comparative Programming Languages" by Leslie B. Wilson and Robert G. Clark

[4] "Programming Languages: Principles and Paradigms" by Allen Tucker and Robert Noonan

[5] "Essentials of Programming Languages" by Daniel Friedman, Mitchell Wand, and Christopher T. Haynes

We will provide a lecture notes document prepared based on the sources given above.

1.2 Tentative Outline

- Introduction
- Programming paradigms
- Language description
- Language implementation
- Statements
- Types
- Procedures
- Functional Programming – Scheme
- Logic Programming – Prolog
- Concurrent Programming – Ada

1.3 Why study PL (concepts)?

It is natural for you to wonder why to study PL concepts. The students' expectation from a course named "Programming Languages" can sometimes be to learn some number of new programming languages. Given this somewhat natural expectation and the fact that this course's main aim is not to teach you new programming languages, you may feel unmotivated to take this course. Below is a list of items that we hope that will motivate you to study PL concepts and hence motivate you to take this course.

1.3.1 Increased capacity to express ideas

- It is widely believed that the depth at which we think is influenced by the expressiveness level of the language that we use to communicate.
- It also depends on how well we know and use the language.
- Those with a limited capability in a natural language are also limited in the complexity of their thoughts.
- If we cannot write or verbally state something, then it is hard to conceptualize it.
- We all are somehow experienced in the natural languages we speak. Hence the effect of our language abilities on what we can think and write may not be obvious.
- When we consider the programming languages, however, the effect of our language abilities is more apparent.
- Programmers are really constrained by how well they know and use a programming language when they write a program in that language.
- The language in which the program is being developed places the limits on the kinds of control constructs, data structures and algorithms that can be used.
- When one is aware of concepts and constructs of PL in general, even if these concepts and constructs are not available in a language, they can be simulated.
- This should not encourage to learn a small number of languages, and to simulate the necessary constructs in the language being used. Usually, if a construct is native to a language, then it is implemented more efficiently.

1.3.2 Improved background for choosing appropriate languages

- Many programmers have a little formal education in PL.
- They learn a couple of languages on their own.
- When starting a new project, they tend to stick with the languages they know, even if the language is not actually suitable for the project.
- Knowing more languages, and especially particular features of those languages, helps to make a better and more informed decision while picking the language to use.

1.3.3 Increased ability to learn new languages

- Computer science and programming is a young discipline compared to other engineering disciplines.
- New languages are being developed continuously.
- One cannot survive without learning new languages.
- However, in terms of concepts in these languages, the change is not as fast. Over the history of programming languages which spans around last 50-60 years, there have been only a few set of concepts (which are actually associated with programming languages paradigms) whereas probably there have been hundreds of programming languages.
- Knowing the concepts simplifies the learning process. Learning a new language having the same concepts as a language you already know would practically take no time.

1.3.4 Better understanding of the implementation

- While learning the concepts, it is interesting and necessary to go into the implementation details of these concepts.
- When a programmer knows the implementation details of concepts and constructs, s/he decides better on the choices of the constructs to use (both for efficiency and for correctness). For example, if you know how recursion is actually implemented in a language, you would be able to decide when and whether to use recursion (or iteration) for efficiency and correctness.
- Certain kinds of bugs can only be understood if the implementation details are known. For example, you may expect that every time the following statement executed

```
if ((x==0) && (++y != z))
```

  the value of the variable "`y`" would be incremented by one. Whether or not this incrementation will be performed every time this statement is executed actually depends on the boolean value evaluation approach of the implementation. If so called "short–circuit evaluation" is used for the evaluation of the boolean expressions, then when the value of the variable "`x`" is non–zero, the first part of the condition will evaluate to false, hence the value of the entire boolean condition can be declared as false (it is not necessary to calculate the rest of the boolean expression hence it is not necessary to calculate the value of the expression "`(++y != z)`"). So in an implementation using short–circuit evaluation for boolean expressions, the value of the variable "`y`" will not be incremented every time this if statement executed, and this can only be apparent to a developer who knows this implementation detail of the compiler for the language.

1.3.5 Increased ability to design new languages

- It may seem to a student that to design a new programming language will never be necessary.

- Most professional programmers occasionally need to design simple languages, especially for user interaction of the complex systems developed.
- Knowing the general concepts of PLs, simplifies and provides a guidance for the design of these pet languages.

1.3.6 Overall advancement of computing

- General belief is that the popularity of programming languages are due to lack of knowledge of the people at managerial positions who decide the language that will be used.
- FORTRAN vs Algol60: Although Algol introduced some new and very important structures, the SW project managers at key positions have failed to understand the advantages of Algol over FORTRAN.
- So we need people at these managerial positions (hopefully you) to know better about programming languages so that we do not lose time with poor languages.

1.4 Programming Language Paradigms

1.4.1 Imperative

Languages within the imperative paradigm are action oriented. They actually were influenced by and reflect the way microprocessors work. A microprocessor performs computations by executing instructions one after another. The execution semantics for the imperative programming languages is similarly based on the execution of the statements in the programs sequentially.

The first programming language that is known to be implemented is an imperative programming language : FORTRAN. Other than FORTRAN, other famous languages that belong to imperative paradigm are Pascal and C.

When FORTRAN was first proposed as a programming language, people used to have concerns for using a programming language other than machine language. They believed the automatic translation from a higher level language like FORTRAN into machine language would not be as efficient as writing the programs directly in the machine language. However, FORTRAN was already very close to the machine language, and the translations turned out to be not as bad as expected. At least, the benefits of using FORTRAN which allowed programming at a higher level were considered to be more than what is lost by not programming directly at the machine code level.

Although it was not intended by the designers of FORTRAN that this language was to be used any machine other than IBM704 (this is the machine for which FORTRAN was supposed to be used on), the mathematical notation approach used by the designers for the syntax of the language, allowed implementations of FORTRAN for other IBM machines, and for machines from other vendors as well.

Algol language is another member of imperative paradigm. Interestingly, this language was not born to be a language in which people would write programs. The only purpose of the designer of Algol was to design a language that can be used to describe algorithms in scientific publications. However, Algol 60 became the most dominant programming language through 60s.

Another imperative language which was not designed to be a full–fledged programming language is Pascal. It was designed to be a programming language to be used at colleges to teach how to do programming. However, it became one of the most influential programming languages.

However, none of these languages become used more than C, which is the most important representative of imperative class of programming languages. It was first designed to be a language for implementing systems programs for UNIX operating system. However, it became the language for most of the companies for implementation and also for universities for educational purposes. Since the introduction of C and acceleration of software production more or less coincide, and since C has led to very efficient implementations and translation, it became one of the most widely used programming language ever.

1.4.2 Functional

The first functional language LISP (acronym for LISt Processor) was born in 1958, for the purposes of application in artifical intelligence.

There have been quite a number of other successful functional languages and dialects of these languages introduced after LISP, e.g. ML, Miranda, Haskell, Scheme, Common LISP, Standard ML.

The functional languages have been used to perform symbolic calculations (rather than being used for numeric calculations). One of the typical applications of functional languages turned to be the study of programming languages.

1.4.3 Object Oriented

With the introduction of object orientation, the way the programmers have to think and organize the programs had to change. The notion of class of objects first existed in Simula. Hence, object oriented programming owes a lot to this language. Simula was originally designed to describe and program simulations.

C++ and Smalltalk are among the most popular object oriented languages. While both of these languages were influenced by Simula, and got their object oriented concepts from it, they are also influenced by two different languages, which makes them quite different.

C++ is influenced by C, which is an imperative language. Whereas Smalltalk is influenced by LISP, which is a functional language.

1.4.4 Logic

Prolog is the most important representative of the logic programming paradigm. It was developed for natural language processing problems. It is different and interesting to use Prolog. As we will see later when we study this language, programming in Prolog is actually composed of giving out facts and propositions to the computer, and then asking questions whose answers are supposed to be derived from these facts and propositions.

1.4.5 Parallel

A parallel programming language possesses the concepts and provides the primitives to describe a set of components that are running simultaneously to accomplish a certain common goal/job.

The basic concepts that do not exist in other programming paradigms are communication and synchronization. Since there exist multiple components working to achieve a common goal, they have to communicate with each other.

By communication we mean exchange of data (either by some message passing mechanism or by using shared variables). By synchronization we mean passing the control point information from a component to others. Therefore, it is still a communication (an information is passed), however it is a very restricted one.

We will study Ada as an example language for this paradigm. It has all the typical concepts of a parallel programming language. It is also one of the most famous and widely used parallel programming languages.

1.5 Programming Domains

Computers have been applied to a variety of domains : scientific computing, business applications, safety critical system, etc. In fact the computers are everywhere now. When we talk about a computer, it does not have to be a full-fledged PC or workstation. Microcontrollers in the cars, home appliances (like washing machines, DVD players, vacuum cleaners, etc.), mobile phones, children toys, medicine dispenser, etc. All of these microcontrollers are programmed to serve their purposes.

However, just as there is a need for a high level programming language on full–fledged computers, there is a need for high level languages in these quite diverse application areas of the microcontrollers. It is not easy to program these chips by using their instructions directly, or put it another way, it is easier to program them by using a high level language that is suitable for their application domain. For example, consider a toy, say a robotic Winnie the Pooh, having a microcontroller in it to move the arms/legs of Winnie. The microcontroller has to execute instructions that send signals through its ports to derive the motors moving the arms and legs. However, rather than writing a program at this level of detail by sending explicit signals through the ports of the microcontroller, the toy designer would be much happier if it was possible to write a program in a high level language where s/he could say simply e.g. "wave_hand('left',3)" (for waving the left hand for three seconds), or "say('hello')" (for saying stuff outloud). A compiler could then easily translate this high level statements into the corresponding sequence of low level instructions that would actually send the required signals to the required ports.

However a high level language that is used to program such a toy would be much different than a high level language that is used to program the microcontroller in a washing machine. There, instead of having a high level statement for waving a hand, one would want to have a high level statement or measuring the water level in the machine, or heating up the water upto a certain degree.

So, different domains for the applications of computers require different programming languages that are more suitable for their purposes.

1.5.1 Scientific applications

- In fact, first computers (1940s) were solely developed for the purposes of scientific computation
- Mainly floating point operations were needed

- Most common data structures needed were arrays and matrices
- Most common control structures are for loops and conditionals
- The high level languages invented for scientific computing answer these needs
- Naturally, the first high level language came from this field

FORTRAN $\longrightarrow$ FORmula TRANslation

1.5.2 Business applications

- The computers began to be used for business in 1950s
- Special computers and languages were developed for business applications
- The first successful language for business application was COBOL (COmmon Business Oriented Language).
- These languages are characterized by their facilities to produce reports, precise ways of decimal arithmetic, ability to store data.
- After PCs were introduced, small businesses also got computerized.
- Spreadsheet applications and databases developed.
- COBOL did not have a successful follower, and remained to be the most successful business application language.
- Due to this fact and the resistance of large organizations to change languages, there are a considerable amount of legacy systems written in COBOL that are still alive and being managed.

1.5.3 Artificial Intelligence

- This area of applications is characterized by the use of symbolic computations (rather than numeric values)
- Numeric computations are better performed using arrays
- However, symbolic computations are better performed using linked lists
- The first widely used language for AI was the functional language LISP (1959)
- Many dialects of LISP appeared later (Scheme, ML)
- Another paradigm, logic programming, is also used for AI
- Prolog appears as the most successful logic programming language

1.5.4 Systems Programming

- The operating system and all of the programming support tools of a computer system
- Used heavily, hence they must be very efficient
- It must also have low level features for controlling the interfaces to external devices
- IBM, Digital, UNISYS (back then it was Burroughs) used special machine oriented languages for systems programming
- IBM used PL/S (a PL/1 dialect), Digital used BLISS, Burroughs used Extended ALGOL
- C became almost a standard for systems programming
- UNIX is almost completely written in C

1.5.5 Scripting languages

- A scripting language is used by putting a list of commands (a script) in a file to be executed
- `sh` (standing for shell), the first scripting language, started as a small collection of commands calling system functions
- Then several control structures and variables added
- `awk` began as a language for generating reports, and then expanded into a more general language later
- `Perl` started as a combination of `sh` and `awk` and then became a powerful but a simple language
- `JavaScript` by Netscape is used for scripting purposes in both web servers and browsers

1.5.6 Special purpose languages

- There are languages designed and used for quite specific purposes
- These languages provides necessary features required by their specific application domain
- They excel in their field, but it is hard to compare them with other more general purpose languages

1.6 Evaluation criteria

In order to be able to judge the values of the features of programming languages, we need a list of criteria.

However, it not easy to come up with such a list which has a common acceptance.

Nevertheless, we will use **readability–writability**, **reliability**, and **cost**.

1.6.1 Readability–Writability

- One of the most important factor for judging a programming language is the ease of with which programs can be read and understood.
- Before 1970s, software development was largely thought of in terms of writing code.
- However, nowadays, coding is only a small part of software development
- Major part of the development goes to maintenance, which implies reviewing a code, finding bugs, etc. which all requires the understanding of a previously written code.
- Furthermore, even while writing the code, a programmer has to reread and understand the code written by herself couple of hours ago.

There are several factors affecting the readability of a language

- A language that has a **large number of basic constructs** is harder to read
- The programmers tend to learn and use a subset of the language
- If the person reading the program is different from the person that had written the code, there will be difficulty when they are used to different subsets of the language.
- **Feature multiplicity** (multiple ways of doing the same thing) is another feature that reduces the readability of a program.

```
i = i + 1
i += 1
i++
```

- **Operator overloading** (single symbol, multiple meaning) is a potential danger for readability
- It is actually a useful feature for increasing readability (+ for integer and real addition)
- When programmers are allowed to perform overloading, but not using it sensibly, it may harm readability (e.g. + for subtracting)
- **Orthogonality** : small number of control and data structures, and a small number of ways to combine them
- **Control structures**: Structured programming yields much more readable programs than ones with pure `goto` jump statements. In fact structured programming was born as a reaction to the poor readability of the programming languages in 50s and 60s that allowed liberal use of goto statements. The main observation and claim of structured programming is that, a program is more readable if the control flow of the program during its execution more or less follows the sequence of blocks that we visit during reading the source (text) of the program.

  When goto statements are used, a top–down glance through the source code (this is how we typically read the program) will be very different from how the control will flow in the program when it is actually executed. Hence it will be more difficult to understand what the program will do (read the program) by looking at the source code.

- **Data types**: Sufficient facilities for defining data types and structures increases readability. For example, for a boolean flag `finished`,
  `finished = true`
  is much more readable than
  `finished = 1`
  Similarly `structs` help to increase readability by collecting data closely related to each other in a single object. For example, if we were to keep a list of people with their names and last names, we can keep two arrays of strings (one for names and another for the lastnames). So, `names[i]` and `lastnames[i]` would be the name and the lastname of the i^{th} person. However, it would be more readable if we define a record type with two fields `name` and `lastname`, and use an array where each element is of this record type. Hence `person[i]` would denote the i^{th} person, and we would use `person[i].name` and `person[i].lastname` to access the name and the lastname of the i^{th} person.

  By having our data as organized appropriately, we would increase readability of the program.

- **Restrictions on identifiers** can have a bad influence on the readability. For example, FORTRAN 77 constrained identifiers to at most six characters. Such a restrictive constraint on identifiers will not allow the programmer to use a descriptive name for an identifier. One extreme example is ANSI BASIC, which allows only a single letter or a single letter followed by a single digit to be used as an identifier.

- Keywords can have various effects on readability.
  For example, C closes every block with } whereas FORTRAN 90 uses `end if` to close an `if`, `end loop` to close a loop. This increases the readability as it is easier to see the blocks.

- It also simplifies readability to disallow the use of keywords as identifiers

1.6.2 Reliability

A program is said to be reliable if it performs its intended functionality under all conditions.

It is mostly the programmers responsibility to produce a correct code. However, some features of programming languages may help to detect errors.

Type Checking

- A type error occurs when the actual and the intended type of an expression are different
- For example there is a type error in the assignment statement given below

```
char x;
int y;
...
  x = y;
```

- Different type of expressions use different memory layouts (e.g. char=1 byte, int=8 bytes, etc.)

- When an integer expression is used at a place where a character is needed, only one of the bytes are used, which results in a value different than the current value of the integer expression
- Type checking is simply testing for type errors
- Checking type errors in assignment statements, function applications, operator applications, conditions of if statements, etc...
- Type checking (checking for the type errors) can be performed either at compile time or at run time
- Type checking at run time is expensive, so compile time checking is more desirable. However, full type checking may not be possible at compile time (e.g. arrays' out of bound check for indices)

Exception handling

Almost all programs are buggy. It is hard to consider every condition that the program will probably face. The unforeseen conditions will cause a malfunction in the program.

It is important to be able catch such errors at run time (even if the reason for it is not known), and recover from the error.

Some languages provide **exception handling** facility to catch run time errors and to take corrective measures, which increases the reliability of the programs written in these languages (examples : C++, Java, Ada)

Aliasing

- Aliasing is to have two or more references to the same memory
- For example: two pointers pointing to the same memory location

```
char x;
char *p;
  ...
  p = &x;
  ...
  x = 'a';
  *p = 'b';
  ...
```

- Aliasing is a typical source for common errors unless it is used with extreme care (assign one, the other also changes)
- Some programming languages prohibit aliasing completely to increase the reliability of the programs

Readability and writability

Readability and writability are also important factors for the reliability of the programs. If it is easy to write and understand the programs, then it is more likely that it will be easier to correct them, hence they will become more reliable.

1.6.3 Cost

Cost of a language is also an important factor for its evaluation.

However, it is hard to assess the cost of a programming language.

- Cost of training: Training programmers for the language (simplicity, less number of constructs)
- Cost of development: Closeness to the application domain affects the cost of development. If a language is designed for systems programming, then it will be quite expensive to do web programming with it (it is cheaper to use a language that is designed for web programming)
- Cost of available tools: Availability and cost of development tools, e.g. compilers and IDE) will affect the cost of projects developed in a language. (Java's popularity – free development environments and compilers) (Ada is a bad example)
- Cost of executing programs: run time type checking slows down the programs. If fast code is required, a compiler with a good optimization is needed. However, for educational purposes (compile frequently, run a few time applications), a fast compiler is better.
- Cost of poor reliability: If the application to be developed is critical, then a language with good reliability is better (X-ray machine control)
- Cost of maintaining programs: readability

1.7 Influences on the language design

There are mainly two factors that affect the design of a language: computer architecture and programming methodologies

1.7.1 Computer Architecture

The basic architecture of the computers have a basic role on the design of the language.

The dominant architecture for computers is von Neumann [John von Neumann]

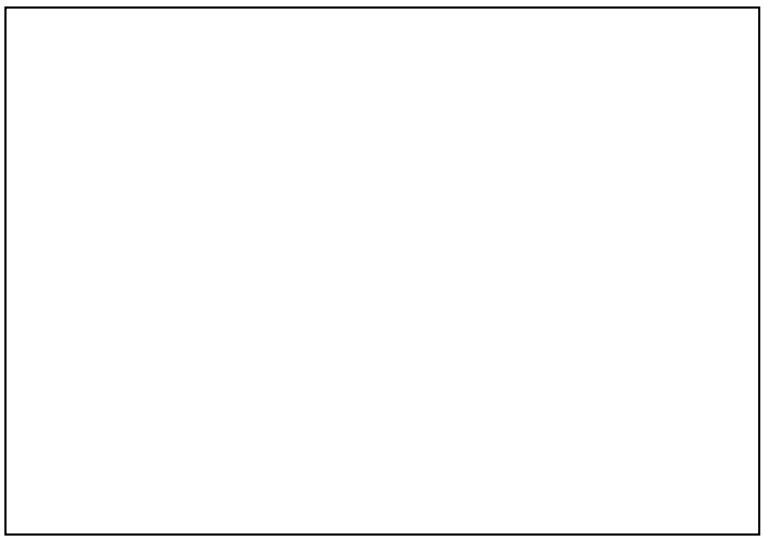
[von Neumann arch]

The program and data are stored in the memory (a random access memory).

CPU and the control unit is separate form the memory.

The instructions in the program fetched from the memory and executed by the CPU.

Data is received from the memory, operations are performed on it, and then the result is stored back at the memory.



A language yields fast running programs if it's constructs are close to the machine architecture on which the programs will be executed.

Imperative languages use constructs that mimic the operations and components of von Neumann architecture.

The basic property of imperative languages:
– variables (memory cells)
– assignments (x=y+z, read y and z form the memory, add them up in the CPU, store the result to the memory cell for x)
– iteration : most efficient way of implementing repetition on von Neumann (instruction are stored in adjacent memory cells, jump back is easy)

Although recursion is often more natural way for repetition, it is not suitable for von Neumann.

Functional languages provide constructs to express repetition as recursion easily. It is possible to write programs without the variables in the sense we are used to, and it is usually more natural (how silly is x = x + 1) (functions and function applications).

1.7.2 Programming methodologies

Until early 1970s, scientific computing dominated the software industry. The hardware was quite expensive, and big research companies, defense and government bodies were the only customers that could afford these expensive investments.

When the hardware prices started go down, computers became affordable for many other businesses. This increased the spectrum of application areas. Bigger and bigger software programs needed to be written. However, the programmers abilities to produce such big programs could not follow the demand.

Hence people started to look for better ways of writing programs.

The first research results showed that the limited number of control structures (only if and goto) lead to huge number of goto statements, which decreases the readability and hence maintainability of programs. To remedy the situation *structured programming* methodology was proposed which introduced the loop constructs (while, for, until, etc) that removed the need for goto statements.

Late 70s brought a shift from control oriented design to data oriented design which emphasizes data design. Several data definition and data abstraction constructs were introduced into the languages.

Data oriented design evolved into object oriented design methodology in early 80s. Object oriented methods requires the processing to be encapsulated within data.

To simplify SW reuse, inheritance was introduced as well.

1.8 Implementation methods

As explained earlier, two of the main components of a computer are memory and the CPU.

CPU is a collection of digital circuits that provides very primitive instructions such as arithmetic and logical, which are called **macroinstructions**.

In fact, CPU divides these macroinstructions into smaller pieces called **microinstructions** to execute them. However the microinstructions are invisible (and irrelevant) from the software point of view, hence are not studied and even not known by the programmers.

Regardless of the programming language we use, the program at the end runs on the computer by the execution of an equivalent sequence of macroinstructions.

It is theoretically possible to build a computer that directly executes the statements in a programming language. For example **C machine**, **Pascal machine**, etc.

There are some disadvantages for such an approach. First of all, currently computers are quite general purpose, they can serve to a variety of purposes. They can be a quite efficient calculator, a web server, a word processor, a controller. When a machine understands only a single high level language, then it will be (efficiently) usable only for the application area for which the language is designed for.

Furthermore, with the language implementations, the effect of having machines for individual languages are created.

[**the machine hierarchy : machine $\subseteq$ machine + OS : then a PASCAL machine, a C machine, etc.**]

Consider a computer with no software on it. As is, it provides very low level instructions to the user. The only way to program it to use the macroinstructions provided by the machine.

It is quite hard to write programs using such primitive instructions. Most of the programming time goes into developing routine jobs that are common to most software, e.g. i/o with external devices, resource management, etc.

Operating systems encapsulates the bare computer. They lift the programming primitives to a higher level by providing instructions for I/O operations, resource management, etc.

Therefore (a computer+an operating system) is a virtual machine which can be programmed by using the primitives of the OS. A programmer see (the computer+OS) as if it were a hardware built as capable of executing the instructions from OS.

Programs can still be written using machine primitives.

Similar to the OS virtual machine, a high level language implementation provides a virtual machine for that language. E.g. a **C++ virtual machine**.

1.8.1 Compilation

Produces machine language programs. Before the execution of a program produced by a compiler, the other programs required to run the generated program (e.g. library modules) must be found and **linked** together.

Sometimes a generated program must be linked with other user programs as well.

Linker (or loader) performs this job and produces the **load module** or **executable image**.

Linking (or loading) is performed by a systems program called **linker**.

1.8.2 Interpretation

Compilation is partially visible to the user, hence shades away the view that the computer is a virtual machine.

Another implementation technique, called **interpretation** gives the affect of executing the statements in the high level language directly.

In fact the statements in the program, inevitably, are converted into machine language and OS function calls. However, this conversion is performed at run time directly and completely invisible to the user.

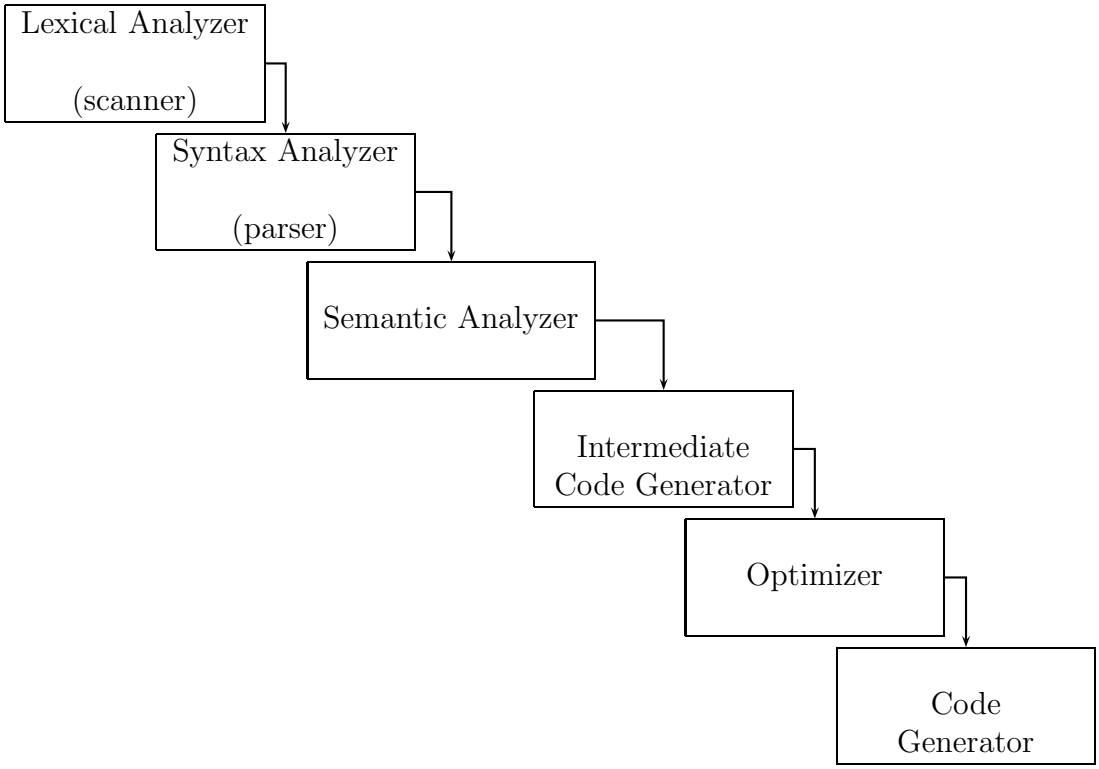


Figure 1.1: Typical phases of a compiler

Interpretation technique results in slower running programs, as it is not possible to optimize the program using techniques that require an overall analysis.

However, it reduces the development time as it does not require compilation of the entire program in advance.

Some languages are more suitable for interpretation, e.g. LISP.

Shell scripts and JavaScripts are examples of languages that are interpreted.

1.8.3 Hybrid implementation

First compile to a certain intermediate language, then interpret the intermediate language.

The most famous example is Java. The **byte code** language is the intermediate language which is generated by a Java compiler. Then the JVM – Java Virtual Machine interprets the byte code.

The initial compilation phase allows some analysis and optimizations increasing the speed of the program.

Sometimes both a compiler and an interpreter is provided for a language. Interpreter is used during the development of the program. Once the programmer decides that the program is relatively bug free, then the compiler is used to generate an executable that is optimized for fast execution.

Chapter 2

Describing Syntax & Semantics

2.1 Introduction

- The description of a language is essential for its success

- It is important for a language to have a clear and simple description (for its increased readability, writability, implementability).

- The users of the language must be able to understand and use the language correctly

- Programming language implementors must be able to understand the language correctly to be able to generate correct compiler and interpreters

- A language is described by providing its

  1. Syntax (the rules for writing or speaking)
  2. Semantics (the meaning)

- This applies to both natural languages and programming languages

- The syntax of a programming language is the form of its expressions, statements and program units

Example: For example the syntax of a C **if** statement is:

```
if ( <expr> ) <statements>
```

- The semantics of this C statement is: if the `<expr>` is true, then execute `<statements>`

- Syntax and semantics are two different aspects of a language description

- However, the syntax must be suggestive for the semantics. E.g. consider

"`if ( <expr> ) <statements>`" vs. "`<expr> : <statements>`"

- Syntax description is easier than semantics description
- There is a universal formal notation for syntax description
- Such a universal description for the semantics of a language does not exist

2.2 Syntax Description

- A language, whether natural or programming, is defined over an alphabet.
- A sentence in a language is a sequence of elements from the alphabet. An equivalent and easier view of a sentence is that a sentence is a sequence of words, where words are sequence of alphabet symbols and they are separated by using space and some punctuation symbols.
- The syntax rules of a language is described for a complete sentence in the language.
- A sentence in a natural language is well known. We do not give a grammar rule for a paragraph or for a full document.
- What is a sentence in a programming a language? : **a full program**
- The syntax rules of a language give the rules for producing the correct sentences in the language

Example: English

```
<sentence> ::= <subject> <verb> <object>
               I ran home
```

- Natural languages have much more complex syntax rules than programming languages
- However, the same syntax description techniques are used for natural languages.
- The basic unit used in the syntax description of a programming language is called a **token**. You can consider a token corresponding to a word in a natural language. However, for programming languages, punctuation symbols are also tokens (words).
- Just like we have different type of words (like nouns, verbs, adjectives, etc.) in natural languages, we have different kind of tokens (words) in programming languages.

Example: Typical tokens for programming languages are `tADD, tMULTIPLY, tASSIGN, tIF, tIDENT,` etc., corresponding to addition, multiplication, assignment, if, and identifier respectively. Note that

the names of the tokens here are made up names. They might as well be `add, mult, asgn, if, identifier`. The token names to be used are up to the compiler programmer.

- The spelling of a token is the actual sequence of the characters (= elements of the alphabet) that form the token. This is also called the **lexeme** of the token.

- For example, suppose that we use a token named `tADD` to denote the addition operator in C, which we know represented by the character "`+`". In this case, the lexeme of `tADD` is `+` in C.

- The lexemes for some tokens are fixed by the language description as it is the case for `tADD` example given above. Other examples for fixed lexeme tokens are given below:

Example: The lexeme for `tASSIGN` in C is `=`

Example: The lexeme for `tASSIGN` in Pascal is `:=`

Example: The lexeme for `tIF` in C is `if`

- However, for some tokens, the lexemes are only defined in the actual program. The tokens for numbers and identifiers are these type of tokens. The actual lexeme of a number token or an identifier token will only be known after scanning the input.

Example: Consider the following code fragment in C

```
int width = 13;
int length = 55;
```

This corresponds to five tokens

- `tINT` (the token for integer type)
- `tIDENT` (lexeme: width)
- `tASSIGN`
- `tINT_NUMBER` (lexeme: 13)
- `tSEMICOLON`
- `tINT`
- `tIDENT` (lexeme: length)
- `tASSIGN`
- `tINT_NUMBER` (lexeme: 55)
- `tSEMICOLON`

• For such tokens (identifiers, numbers), there may be different spellings even in the same program, as can be seen above. For example there, we have two instances of `tIDENT`. The one on the first line has the lexeme `width`, and the other one on the second line has the lexeme `length`.

• Even though such tokens may correspond to multiple lexemes, the syntactic description of a language introduces rules for the possible lexemes of such tokens. Therefore for example, not every sequence of characters can be a lexeme for `tIDENT`.

Example: For example in C, we cannot use `-` in identifiers.

• Typically, an identifier must be a sequence of letters, digits and the underscore character.

• Sometimes, the same character or character sequence may correspond to different tokens in different contexts.

Example: Consider the two statements given below

```
x = 5;
if ( y == 4) ...
```

In "`x = 5;`", the symbol "`=`" corresponds to `tASSIGN` token. However the two characters `==` in the condition of the `if` statement do not correspond to two consecutive `tASSIGN` tokens, but rather they together correspond to a single `tIS_EQUAL` token.

• If a token corresponds to a single lexeme (like `tIF`, `tASSIGN`, etc.), then it is a **keyword** of the language. For example, "`class`" is a keyword in C++.

• If a keyword cannot be used as an identifier, then it is also a **reserved word** of the language. For example "`if`" is a reserved word in C, i.e. you cannot name a function or a variable as "`if`". The lexeme "`if`" is reserved for the branching control structure (i.e. for the "if statement") of C.

2.2.1 Recognizing tokens

• Recall the structure of a compiler given in Figure 1.1 on Page 16.

• The first phase of a compiler is **lexical analyzer**.

• Note that, a program, being a sentence in a programming language is simply a sequence of characters in the alphabet of the language

• The purpose of a lexical analyzer is to *scan* an input program, and convert the input program

(sequence of characters) into a sequence of tokens.

- A lexical analyzer is also called as a **scanner**, as it scans the input program.
- The number of keywords is finite for a programming language.

Example: For example, for C, an incomplete set of keywords is

$$\{ \texttt{if, else, while, for, =, <=, +, *} \}$$

- Hence, we may write a program that will scan the input, and will try to match these keywords.
- What about the numbers?

$$1,\ 2,\ 3.14,\ 123455778999765$$

We cannot give a complete list of possible numbers that the programmer may use.

- Similarly, what about identifiers?

```
my_beatiful_and_useful_function_with_a_very_very_long_name
```

- How do we describe legal numbers and legal identifiers?
- The solution is to use "regular expressions".
- Regular expressions are patterns (of characters) using which we can describe sequences of characters obeying a certain combination rule.
- Using these patterns, we can describe which sequence of characters should correspond to identifiers, numbers, etc.
- Regular expressions are themselves described by using a language which has a syntax and a semantics.
- They are widely used in tools like `awk, sed, perl, grep`.
- Each tool may impose a different syntax for describing regular expressions.
- We will see an example usage: that of `flex`

2.2.2 Regular expressions

- A regular expression corresponds to a set of strings (character sequences).
- If a string is in conformance with the pattern described by a regular expression, than that regular

expression is said to **match** that string.

- Some example regular expressions

Regular expression	Matches
a	a
x	x
abc	abc
123	123
\n	a newline character

That is, a string matches itself

- A group of characters

Regular expression	Matches
[abc]	a or b or c
[123]	1 or 2 or 3
.	any character other than newline

- A range of characters

Regular expression	Matches
[ak-nx]	a, k, l, m, n or x
[A-Z]	any capital letter
[0-9]	any digit

- Negation

Regular expression	Matches
[ˆA-Z]	any character that is not a capital letter
[ˆ\n]	any character that is not a newline

Note that the combined usage of constructors are possible, like in [ˆA-Z]

- These basic regular expressions can be combined together to form more complex regular expressions

- + can be used to match 1 or more repetitions of a regular expression

Regular expression	Matches
[a-z]+	any word written in small letters
[0-9]+	any positive integer number (possible with preceding 0s)

$*$ is similar to + but it denotes 0 or more repetitions

- Concatenation

Regular expression	Matches
-[1-9][0-9]∗	any negative integer

Note that the meaning of − is different within and outside of [ ]. Inside, it means a range, whereas outside, it means the - character.

- Other repetitions

Regular expression	Matches
[a-z]?	0 or 1 repetition (used for optional parts)
-?[1-9][0-9]∗	any nonzero integer
[A-Z] { 4 }	any word of length 4 written in capital letters
[A-Z] { 2, 4 }	any word of length 2, 3 or 4 written in capital letters
[1-9][0-9] { 2, }	any positive integer of 3 or more digits

- Let us assume that a programming language requires identifiers to start with a letter which can be followed by a sequence of letters, digits and underscore

[a-zA-Z] [a-zA-Z0-9_]∗

2.2.3 Flex introduction

- Given a regular expression we can write a program that will try to match the regular expression on given texts (program sources).

- Regular expressions define a regular language.

- A set of regular expressions will therefore define a set of (union of) regular languages, which is again a regular language.

- Hence, a program that will match a given set of regular expressions will be an implementation of a finite state automaton.

- However, trying to match a set of regular expressions is a common programming need, and for that reason people have developed tools that will create such programs automatically.

- I.e., there are tools that generate programs which match regular expressions.

- Examples are lex, flex, rex

- These specific examples are tools that are widely used for generating lexical analyzers (scanners)

- We will see one of them as an example: **flex**

- Input to flex is a set of regular expressions and commands that tells what to do when each

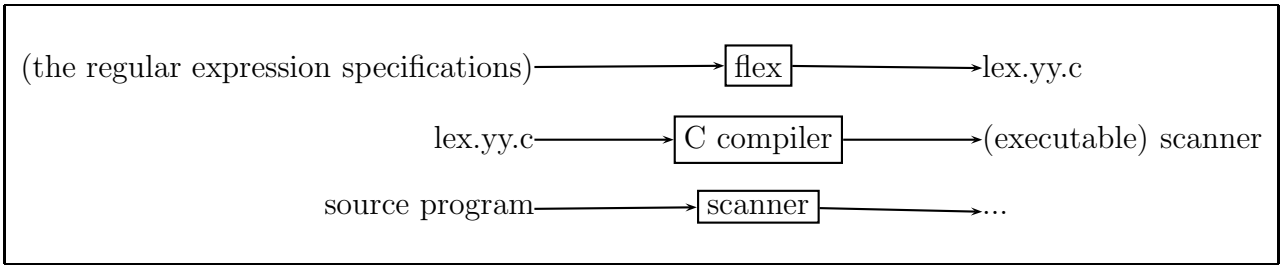


Figure 2.1: flex flow

regular expression is matched

- Output of flex is a C program that tries to match the regular expressions given in the flex file in a given input string and performs the actions associated with the regular expression.

- The general flow is given in Figure 2.1:

- More concretely, if the flex file is named as `myscanner.flx`

```
flex myscanner.flx
gcc -o myscanner lex.yy.c -lfl
```

- [Note: `-o` option of `gcc` is used to give a name to the output that will be produced. In the example above, we want our executable scanner to be named as `myscanner`. If you do not use this option but simply use the command "`gcc lex.yy.c -lfl`", then the output will still be produced, but its name would be "`a.out`".]

- [Note: `-l` option of `gcc` is used for linking libraries. An option of the form `-l`x is a request to link the library `lib`x`.a` (`.a` is the standard extension for static libraries) with the executable that will be produced. Therefore, in the case of the gcc compilation given above, it requests a link with the library `libfl.a`, which is the library of flex.]

- `lex.yy.c` produced by the lex compiler includes a C function named `yylex` which is the function that matches the regular expressions

- A flex file consists of three parts as shown in Figure 2.2.

- In the "regular expressions and actions" part, we provide the regular expressions. Associated with each regular expression is a code fragment that will be executed whenever the corresponding

```
general definitions
%%
regular expressions and actions
%%
user code
```

Figure 2.2: General structure of a flex specification

```
%%
[A-Z] printf("saw a capital letter");
[0-9] printf("saw a digit");
%%
main() {
  yylex();
  }
```

Figure 2.3: A simple flex specification

regular expression is matched in the input

- "user code" part must be written in C and it is directly copied into `lex.yy.c`
- A very simple flex file is given in Figure 2.3
- In this example, the general definitions part is empty.
- Note that a `main` function is given in user code part. Recall that everything in the user code part is directly copied into `lex.yy.c`. Hence, when we produce the executable scanner (by passing `lex.yy.c` through gcc), this `main` function will be executed, which will in turn directly call the `yylex` to start matching the regular expressions given in the "regular expressions and actions" part.
- The "regular expression and actions" part is a list of "regular expression – action" pairs.
- When we run the executable scanner, it starts waiting for a text input, and tries to match the regular expressions ([A-Z] and [0-9] in the example given above).
- When it matches one of the regular expressions given in the flex specification, it executes the code fragment associated with the matched regular expression.
- Normally the code fragment to be executed when a regular expression is matched is given on the same line as the regular expression. However, if the code fragment is too long to be given on a

```
%%
[A-Z] {
      printf("saw a capital letter");
      printf("saw a letter");
      }

[a-z] {
      printf("saw a lower case letter");
      printf("saw a letter");
      }

[0-9] printf("saw a digit");
%%
main() {
  yylex();
  }
```

Figure 2.4: A simple flex specification with long actions

single line, and when it spans more than one line, it has to be enclosed in two braces as shown in Figure 2.4.

• Although it is possible to make the generated scanner read the input text from a file whose name is provided as a command line option, the default scanner reads its input from the standard input stream.

• Therefore, if we produce our scanner in the way explained above, and with the name `myscanner`, we can provide the input text (the source in which the scanner will look for parts that can be matched by the regular expressions) in the following ways:

- Either start the scanner by simply typing its name on the command line:

  ```
  $> myscanner
  ```

  In this case, the scanner will start waiting for you to type something in as the input text. As you type and send new line characters (hit return key) on the console (standard input stream is connected to the console input), the scanner will try to match the regular expression in what you type. In order to send "end of file" character, you need to type `^D` (i.e. `ctrl-D`).

- Another way is to write the input text and store it in a file, say `input.txt`. Then, we can pipe the contents of this file to the standard input stream by using input redirection in the following way:

```
DIGIT [0-9]
NONZERODIGIT [1-9]
CAPLETTER [A-Z]
%%
{CAPLETTER} printf(" saw a capital letter ");
{NONZERODIGIT}{DIGIT}* printf("saw a positive integer");
%%
main() {
  yylex();
  }
```

Figure 2.5: A flex specification with nonempty definitions part

```
$> myscanner < input.txt
```

In this case, the scanner will read the contents of the file `input.txt` as its input, and try to match the regular expressions in the contents of this file.

- As the regular expression in the flex specification, we can use the regular expressions we have seen before.

- In addition to these, there are several other features that can be used.

- The first part, "general definitions" part, of a flex specification can be used to give a name to a regular expression to simplify the specification. We can use these named regular expression in the second part of the flex specification, to increase the readability of our regular expressions and actions.

Example: The flex specification given in Figure 2.5 is an example showing how the general definitions part can be used.

- The name `DIGIT` is given to the regular expression `[0-9]` in the general definitions part. Then, this regular expression is used within the second part. Note that, when it is used in the second part, it is enclosed within braces, as `{DIGIT}`. The reason is that, without such an escape notation, it would not be possible distinguish what is really wanted: `[0-9]` or `DIGIT` itself.

- The part of the input that is not matched by any regular expression is directly copied to the output.

Example: If we provide the text

```
a4Bc
```

to the flex specification given above, then the output will be

```
asaw a positive integersaw a capital letterc
```

- Matching at the beginning of a line

```
...
%%
...
^[0-9] printf("saw a digit at the beginning of a line");
...
%%
...
```

Note that the semantics (meaning) of ^ is different in and out of `[ ]`

- Matching at the end of a line

```
...
%%
...
[0-9]$ printf("saw a digit at the end of a line");
...
%%
...
```

Again, note that the semantics of $ is different in and out of `[ ]`

- Matching only if it is followed by a certain pattern

```
...
%%
...
[0-9]/[0-9] printf("saw a digit followed by a digit");
...
%%
...
```

Suppose we provide the following as the input

```
123
```

Then the output of the scanner will be:
1 → saw a digit followed by a digit

```
    int noOfLines = 0;
%%
\n noOfLines++;
<<EOF>> printf("There are %d lines\n", noOfLines); return;
%%
main() {
  yylex();
  }
```

Figure 2.6: A flex specification to count the number of lines

$2 \rightarrow$ saw a digit followed by a digit
$3 \rightarrow 3$

• Let us write a flex specification to count the number of lines in the input. We can use the flex file given in Figure 2.6.

Example given in Figure 2.6 shows another usage of the general definitions part. We can also declare global variables in this part. In fact, any line in the general definitions part that starts with a whitespace directly copied into `lex.yy.c` by flex.

Therefore one has to make sure that the definitions for regular expression in this part start at the beginning of the line to be processed correctly by flex. For example, in Figure 2.5, the lines for `DIGIT`, `NONZERODIGIT`, and `CAPLETTER` in the general definitions section should not have any leading white spaces. Otherwise, these lines would be copied into `lex.yy.c` as is.

```
%{
int noOfLines = 0;
%}
%%
\n noOfLines++;
<<EOF>> printf("There are %d lines\n", noOfLines); return;
%%
main() {
  yylex();
  }
```

Figure 2.7: C code in general definitions part to be copied to `lex.yy.c`

• We can also give a C code fragment in the general definitions part to be directly copied into

`lex.yy.c` by enclosing the C code fragment between `%{` and `%}` as shown in Figure 2.7.

• Another feature of flex introduced by the example given in Figure 2.6 is the predefined regular expression `<<EOF>>`, which matches the end of file symbol.

• A scanner built by flex matches the regular expressions according to the following rules: The input is scanned until a match is found.

- If there are more than one regular expression matching (e.g. `=` and `==`) then the regular expression that yields the longest lexeme is considered.
- If there are more than one way of matching the same lexeme (e.g. `if` as keyword and as identifier) than the rule that is given earlier in the flex specification is considered.

Example: Consider the following flex specification

```
%%
= printf("saw an assignment\n");
== printf("saw an equality check\n");
[^ \t\n]+ printf("saw a word\n");
if printf("saw an if\n");
%%
main () {
  yylex();
  }
```

• When we supply the following input text to this flex specification: "`= == = == if`"
The output will be :

```
saw an assignment
 saw an equality check
 saw an assignment
 saw an equality check
 saw a word
```

• Note that, the first "`saw an assignment`" and the remaining outputs are not aligned. This is because we have spaces between the items in the input. These spaces, since they are not matched by any of the regular expressions, are directly copied to the output.

• After finding a match and executing its corresponding action, the scanner continues from the point it left to find a match in the remaining part of the input.

• There are some built in variables (automatically inserted in `lex.yy.c`) which are available to

be used in actions
`yytext` : pointer to the actual string matched
`yyleng` : the length of the matched string

- Similarly, there are some built in functions available

`ECHO;` : print out the matched string (`printf("%s", yytext)`)
`REJECT;` : reject the current matching rule and take the next best match

- Let us try to count the number of words and the number of "`if`"s in the input text.

```
        int word_count = 0;
        int if_count = 0;
%%
if          if_count++; word_count++;
[^ \t\n]+   word_count++;
%%
main () {
  yylex();
  printf("There are %d words.\n", word_count);
  printf("There are %d if's.\n", if_count);
  }
```

or the same thing could have been done

```
%{
int word_count = 0;
int if_count = 0;
%}
%%
if          if_count++; REJECT;
[^ \t\n]+   word_count++;
%%
main () {
  yylex();
  printf("There are %d words.\n", word_count);
  printf("There are %d if's.\n", if_count);
  }
```

- `"SomeTextHere"` can be used to match a string. Note that with the apostrophes, all the special characters lose their meaning (except \")

```
...
%%
...
"abc\"def\"ghi" ...
...
%%
...
```

This will match `abc"def"ghi`

- The action of a pattern can be empty, in which case, the matched string will simply be discarded. For example, the following rules can be used to remove any space and tab in a file (but it would pass the newline characters to the output)

```
%%
" "
\t
%%
main () {
  yylex();
  }
```

- How can we write a flex program that will reduce consecutive space and tabs into a single space?

- Remove the white space at the end of the lines as well?

- The special action of | for a pattern means that the action is the same as that of the following rule

```
%%
[0-9]+      |
[0-9A-F]+   printf("saw a hex number");
%%
...
```

- Assume that we want to write the digits and the letters in the second rule above separately.

```
%%
[0-9]+           |
([0-9]|[A-F])+   printf("saw a hex number");
%%
...
```

Note that, since + has a higher priority than |, we should not use

```
%%
[0-9]+           |
[0-9]|[A-F]+     printf("saw a hex number");
%%
...
```

The second regular expression above will not match a sequence of digits and letters A–F. Instead it will match a digit, or a sequence of letters A–F.

• **Start conditions:** Assume that we are to write a scanner for C++. So, we will write a rule for each keyword like

```
...
%%
...
if       return tIF;
while    return tWHILE;
else     return tELSE;
int      return tINT;
...
%%
...
```

However, when these keywords are seen within a comment, we should actually ignore them. Therefore we could do something like the following.

```
...
    char inComment=0;
%%
"/*"    inComment=1;
"*/"    inComment=0;
...
if      { if (inComment==0)
           return tIF; }
while   { if (inComment==0)
           return tWHILE; }
else    { if (inComment==0)
           return tELSE; }
int     { if (inComment==0)
           return tINT; }
...
```

However, a construct of flex called **start conditions** simplifies this job.

```
%x comment
%%
"/*"              BEGIN(comment);
<comment>"*/"     BEGIN(INITIAL);
...
if      return tIF;
while   return tWHILE;
else    return tELSE;
int     return tINT;
...
```

In fact, there is a predefined start condition called `INITIAL` and every rule which does not have a start condition implicitly defined for `INITIAL`. So, the above flex program is equivalent to the following.

```
%x comment
%%
<INITIAL>"/*"     BEGIN(comment);
<comment>"*/"     BEGIN(INITIAL);
...
<INITIAL>if       return tIF;
<INITIAL>while    return tWHILE;
<INITIAL>else     return tELSE;
<INITIAL>int      return tINT;
...
```

`BEGIN` is used to switch from one start state to another.

- A useful feature of flex for debugging purposes is the `-d` option

```
%%
[a-zA-Z][a-zA-Z0-9_]+   { /* catch identifiers*/
                          /* return tIDENT; */
                          printf("saw identifier");
                        }

while                   { /*return tWHILE;*/
                          printf ("saw while");
...
```

When we use `flex` with `-d` option, it will produce a scanner that prints out some debugging information

```
flex -d myfile.flx
```

Suppose we provide the following input to the scanner:

```
while notFinished ...
```

The output will be

```
--accepting rule at line 2 ("while")
--accepting default rule (" ")
--accepting rule at line 2 ("notFinished")
```

2.3 Context Free Grammars

- In the previous section, we have seen that the basic building blocks of a language syntax are tokens.

- In this section, it will become clear why this is so...

- The following

```
go who that orange
```

is not a valid English sentence, although all the words (`tokens`) in it are valid English words.

- Natural languages have a set of rules (called the grammar rules) to describe the set valid of sentences.

- Just like we have grammatical rules in a natural language to put the words one after another to form legal sentences, there are rules that specify how tokens must be put together to form a legal program.

- Two different researchers, namely John Backus and Noam Chomsky, invented almost the same notation to describe the grammar rules of languages

- Noam Chomsky, a great mind and a famous linguist, was mainly interested in formalizing the rules for natural languages.

- In mid-50s, he introduced two classes of grammars, namely regular and context free grammars, which turned out to be suitable for describing the syntax of programming languages as well.

- John Backus gave a fully formal description of Algol 58 (in 1959) using a notation.

- In 1960, Peter Naur[1] modified the notation of Backus a little bit, and gave the formal syntax of Algol 60.

- Hence the latter notation is named as Backus-Naur Form, i.e. BNF.

- In fact, BNF is similar to what was used by Panini a several centuries B.C. while describing the syntax of Sanskrit.

2.3.1 BNF & Context Free Grammars

- It is remarkable that BNF (by Backus and Naur) is almost identical to Context Free Grammars of Chomsky.

- BNF and CFG are used almost interchangeably.

- BNF and CFG use abstractions (a.k.a. nonterminal symbols or nonterminals) for syntactic structures.

- For example, a simple C assignment might be represented by the abstraction `<assign>`

- We will use the angular brackets `<` and `>` to delimit the names of the abstractions.

- The actual definition of `<assign>` might be given by

[1] 2005 ACM Turing Award Winner – please see Appendix A

```
<assign> -> <var> tASGN <expression>
```

- In such a definition for an abstraction, the text to the left of `->` is called the left hand side (LHS) and it denotes the abstraction whose definition is being given.

- The text to the right of `->` is called the right hand side (RHS) and it is the actual definition

- RHS is a sequence of abstractions (nonterminals) and tokens (terminals).

- The definition is also called a **rule** or **production**

- Sometimes, when it is clear, the actual lexemes of tokens can also be used

```
<assign> -> <var> = <expression>
```

- Intuitively, the rule above says that:

  An instance of the (abstraction) `<assignment>` is formed by an instance of the (abstraction) `<var>`, followed by =, followed by an instance of the (abstraction) `<expression>`.

- As seen from the example above, the definition of an abstraction may refer to the other abstractions.

- One example sentence whose syntactic structure is given by the example rule above is

$$c = a + b$$

Here, `c` is an instance of the abstraction `<var>` and `a + b` is an instance of the abstraction `<expression>`.

- A nonterminal can have more than one definition

Example: Consider "if statement" in Pascal

```
<if_stmt> -> if <boolean_expr> then <stmt>
<if_stmt> -> if <boolean_expr> then <stmt> else <stmt>
```

- Such multiple productions for the same nonterminal can be given in as a single production combining alternative RHSs with the "OR" operator `|`

```
<if_stmt> ->  if <boolean_expr> then <stmt> |
              if <boolean_expr> then <stmt> else <stmt>
```

- Productions can be (directly or indirectly) recursive.

Example: Consider the following grammar for describing the syntax of list of identifiers separated by comma

```
<identifier_list> ->  tIDENT
                    |  tIDENT , <identifier_list>
```

- A CFG description of a syntax of a language is a simply a collection of such rules.

Example: CFG for a very simple programming language

```
    <program> ->  begin <stmt_list> end
  <stmt_list> ->  <stmt>
                |  <stmt> ; <stmt_list>
       <stmt> ->  <var> = <expression>
        <var> ->  tIDENT
 <expression> ->  <var> + <var>
                |  <var> - <var>
                |  <var>
```

2.3.2 Derivations

- Each CFG description has a distinguished nonterminal called *the start symbol*

- Unless specifically defined, the start symbol is the nonterminal on the LHS of the first production given

- Application of a production to a nonterminal is performed by replacing the nonterminal by the RHS of one of the productions of the nonterminal

Example: Application of the rule

<program> -> begin <stmt_list> end

to the nonterminal <program>, results in

begin <stmt_list> end

- The start symbol of a CFG is a sentential form of the grammar

- A **sentential form** is obtained by applying productions to nonterminals in a sentential form. (recursive definition).

Example: So, <program> is a sentential form.

begin <stmt_list> end

is also a sentential form as it is obtained by application of the rule

<program> -> begin <stmt_list> end

to the nonterminal <program>.
Similarly,

begin <stmt> ; <stmt_list> **end**

is also a sentential form.

- Obtaining a sentential form α_2 from another sentential form α_1 by applying a production to a nonterminal in α_1 is called a **derivation** (of α_2 from α_1).

- A derivation is denoted by: α_1 ==> α_2.

For example:

begin <stmt_list> **end** ==> **begin** <stmt> ; <stmt_list> **end**

Example: A complete derivation using the CFG of the very simple programming language

<program> ==> **begin** <stmt_list> **end**
==> **begin** <stmt> **end**
==> **begin** <var> = <expression> **end**
==> **begin** tIDENT = <expression> **end**
==> **begin** tIDENT = <var> + <var> **end**
==> **begin** tIDENT = <var> + tIDENT **end**
==> **begin** tIDENT = tIDENT + tIDENT **end**

- Note that we can continue derivations on a sentential form as long as there are nonterminals in the sentential form.

- In a sentential form, there can be more than one nonterminals.

- If always the left most nonterminal is used for derivation, then such a derivation is called a **leftmost derivation**.

- A sentential form derived by a leftmost derivation is called a **leftmost sentential form**.

Example: A complete leftmost derivation using the CFG of a very simple programming language

<program> ==> **begin** <stmt_list> **end**
==> **begin** <stmt> **end**
==> **begin** <var> = <expression> **end**
==> **begin** tIDENT = <expression> **end**
==> **begin** tIDENT = <var> + <var> **end**
==> **begin** tIDENT = tIDENT + <var> **end**
==> **begin** tIDENT = tIDENT + tIDENT **end**

- Similar definitions are made for **rightmost derivation** and **rightmost sentential form**.

- When a sentential form consists of terminals only, then it is called a **sentence**.

- The language generated by a grammar is the set of all sentences that can be derived by the grammar.

- CFG can be seen as a **generative device** for generating sentences in the language (using derivations). We will also see that CFG can be used to recognize a language.

- A CFG describes the **concrete syntax** of its language, as it fully specifies the syntactic details of the sentences.

- The grammar generating a language is not necessarily unique. In other words, there can be more than one grammar for the same language.

Example: Both

```
<identifier_list> ->  tIDENT
                    |  tIDENT , <identifier_list>
```

and

```
<identifier_list> ->  tIDENT
                    |  <identifier_list> , tIDENT
```

describe the same language (sequence of `tIDENT`s separated by comma)

2.3.3 Parse Trees

- Another way of presenting how a sentence is derived from a CFG is to use a **parse tree** (a.k.a. **derivation tree**).

- In a parse tree, the order of derivation is not visible.

- A parse tree also describes the hierarchy of syntactic structure of the sentence

- A parse tree is a tree (obviously, as the name implies) where
  - The root is labeled by the start symbol of CFG
  - Non-leaf nodes are labeled by nonterminals
  - The leaves are labeled by terminals (tokens and lexemes directly given in the grammar)
  - For a non-leaf node n , the labels of the children of n read from left to right corresponds to the RHS of a production of the nonterminal labeling n

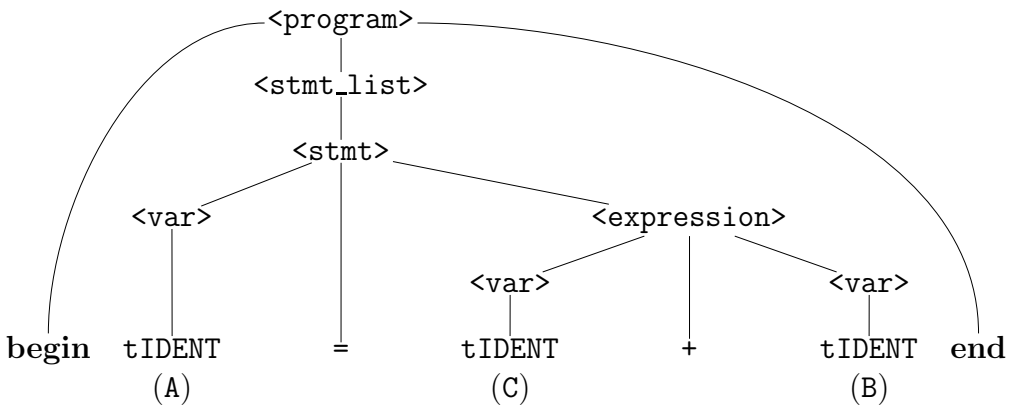
Example: Consider our previous simple language grammar

```
   <program> ->  begin <stmt_list> end
<stmt_list> ->  <stmt>
             |  <stmt> ; <stmt_list>
      <stmt> ->  <var> = <expression>
       <var> ->  tIDENT
<expression> ->  <var> + <var>
             |  <var> - <var>
             |  <var>
```

The(???) parse tree of the sentence

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

with respect to this grammar is



- Note that, the variable names (i.e. the identifiers) `A`, `B` and `C` that appear in the source program are converted into `tIDENT` tokens by a scanner.

- The labels of the leaves read from left to right is called the **yield** of the parse tree

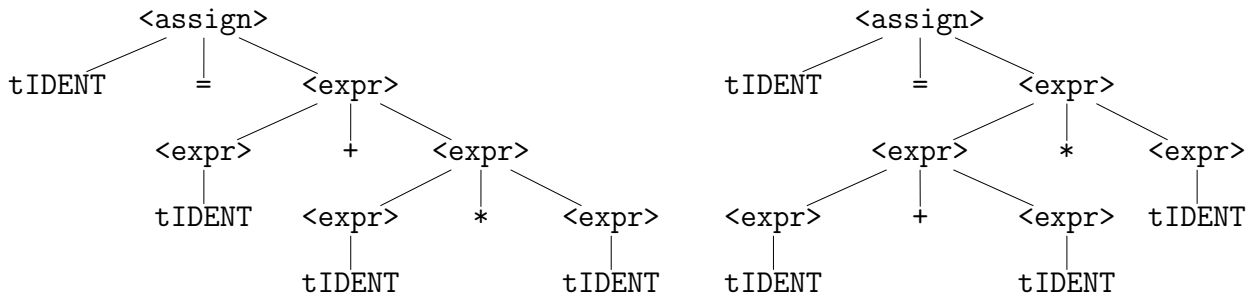
2.3.4 Ambiguity

- A grammar that generates a sentence for which there are two or more distinct parse trees is said to be an **ambiguous grammar**.

Example: Consider the grammar given in Figure 2.8 for a simple assignment statement and consider the sentence `A = B + C * A` that can be generated by this grammar. There are two possible parse trees for this sentence using this grammar

```
<assign> ->  tIDENT = <expr>
  <expr> ->  <expr> + <expr>
           |  <expr> * <expr>
           |  tIDENT
```

Figure 2.8: An ambiguous grammar

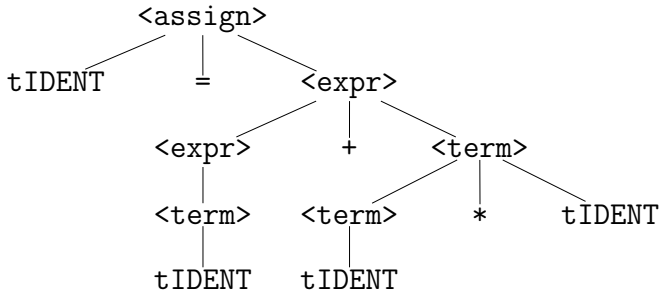


- Recall that there can be more than one grammar for the same language.
- Ambiguity of a grammar does not always imply that the language itself is ambiguous.
- That is, we may be able to find another grammar for the same language which is not ambiguous.

Example: Consider the same language above (for assignment statement) which can be generated by the following grammar which is not ambiguous

```
<assign> ->  tIDENT = <expr>
  <expr> ->  <expr> + <term>
           |  <term>
  <term> ->  <term> * tIDENT
           |  tIDENT
```

The problematic sentence "`A = B + C * A`" in the example above can now be parsed in only one way, and the unique parse tree is:



- A language is said to be **ambiguous** if there does not exist an unambiguous grammar for it.

- (Syntactic) Ambiguity is important for programming languages and it is not a desired property.

- As we have seen in the example above, if there are multiple ways of parsing a program, each of these parse trees give a different meaning to the program. (e.g. `B+C*A`: Is it `(B+C)*A` or `B+(C*A)`? These two ways of interpreting this expression would have different values for most of the values of these variables.)

- Translation performed by a compiler is based on the parse tree of a program. Hence, if there are multiple parse trees for a program, the compiler cannot understand the actual meaning (i.e. what is actually meant by the programmer).

Operator precedence

- The ambiguous grammar given in Figure 2.8 did not have enough information about the priority of the operators. This was the reason why we had an ambiguity in the grammar.

- If we pay attention to the part of the grammar defining `<expr>`, it can produce a sequence of `<expr>`'s where each element is separated by a `+` or `*`.

- If a grammar can generate a sequence of such symbols with more than one different separator, and does not provide a hierarchy (priority) between these separators, then it is an ambiguous grammar.

- When we see an expression like `B+C*A`, we understand what is actually meant. It means `B+(C*A)`.

- Why and how do we understand this?

- Because we have the convention that multiplication has a higher priority than addition, which is actually nothing but well established convention. It would perfectly be consistent if we assume addition has a higher priority over multiplication.

- That is the ambiguity caused by the precedence of the operators, or more concretely, because of the fact that the grammar did not reflect the mutual priorities of these operators.

- Note that, in the two possible parse tree for "`A = B + C * A`", either `+` and `*` operators appeared higher in the trees.

- An operator that appears higher in the tree reflects the fact that it is to be performed later (than the other operators that appear lower in the tree), and hence it has a lower priority.

- It is possible to reflect the correct priority ordering between operators by forcing lower–priority

operators to appear higher in the tree.

- In our unambiguous grammar, we achieved this by introducing new nonterminals and new rules (note how `<expr> -> <expr> + <term>` forces a `+` to be placed higher in the tree).

- This is a general way to get rid of an ambiguity in a grammar that is caused by priority of operators.

- In fact, it is a general way for getting rid of an ambiguity in a grammar due to priorities of separators in a list.

Operator Associativity

- Associativity of operators is another usual reason for ambiguity.

- Or if we think in terms of separators in a list, associativity of the separators at the same priority level, is another reason for ambiguity.

- Note that, `/` and `-` are associative neither in math nor in computer arithmetic.

$$(2/2)/2 \neq 2/(2/2)$$
$$(2-2)-2 \neq 2-(2-2)$$

- Although `+` is associative in math (and usually in computer arithmetic), there are some situations where `+` can be not associative

$$1+(1+(1+.....(1+(10^{107}))..) = 10^{107}$$

due to inaccurate representation of numbers

$$(...((1+1)+1)+.....1)+10^{107} = 1.000001 * 10^{107}$$

since adding 1s first can accumulate a big enough number that is not negligible when 10^{107} is considered.

- Consider a grammar for expressions that includes addition

```
<expr> ->  <expr> + <expr>
        |  tIDENT
```

- Consider an expression `A + B + C` (which is converted in to the token sequence `tIDENT + tIDENT + tIDENT` by a scanner)

- There are two possible parse trees

- One reflects the association `(A + B) + C` while the other reflects the association `A + (B + C)`

- Note that the problematic rule is

```
<expr> ->  <expr> + <expr>
```

which allows the grammar to grow in either sides in a totally symmetric way

- We can again force an order on operators by introducing new nonterminals and rules
- For example consider the grammar

```
<expr> ->  <expr> + <term>
         |  <term>
<term> ->  tIDENT
```

- [draw the parse tree of `A + B + C`]

Note how this grammar pushes the first `+` in the parse tree of `A + B + C` deeper in the tree, hence reflecting left associativity

- A production (rule) is called **left recursive** if the nonterminal on the LHS of the rule appears as the first symbol on the RHS of the production.
- Definition of a **right recursive** production is made similarly.
- The rule

```
<expr> -> <expr> + <term>
```

is left recursive.

- Left recursive rules are used to reflect the left associativity of operators

Dangling else

- Recall the rules we gave for "if" statements before

```
<if_stmt> -> if <logic_expr> then <stmt>
<if_stmt> -> if <logic_expr> then <stmt> else <stmt>
```

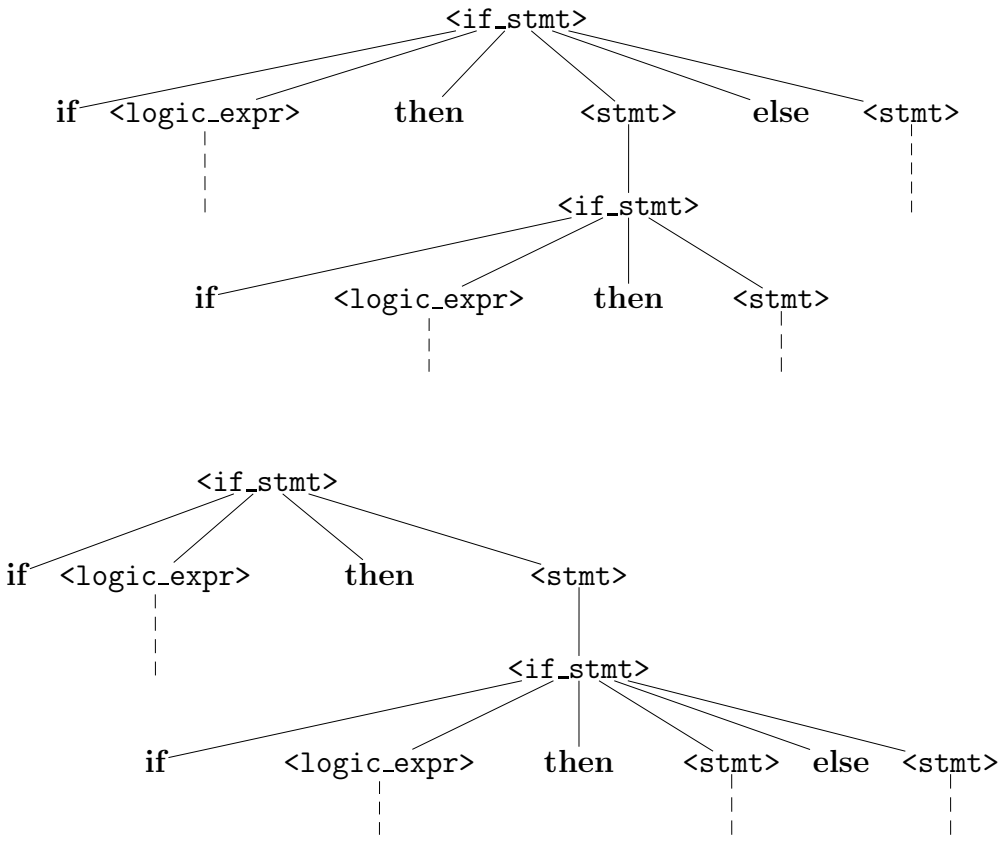
- Assume that we also have the following rules defined in the grammar:

```
<stmt> ->  <if_stmt>
         |  <asgn_stmt>
         |  <while_stmt>
         |  ...
```

- Then it is easy to see that this grammar is ambiguous.
- Consider the following sentential form

if <logic_expr> **then if** <logic_expr> **then** <stmt> **else** <stmt>

- There are multiple possible parse trees



- The problem is that there is no enough syntactic rules to understand to which "if" statement an "else" belongs to

- In most languages, the convention is the following:

  An "else" belongs to the nearest previous "if" which does not have an "else".

- Or equivalently, between a "then" and its matching "else", there cannot be an "if" without an "else"

- Therefore, the statements must be distinguished as those that are unmatched ("else"less "if" statements)', and those that are matched (any statement that is not unmatched)

- The problem with the grammar above is that, it treated every statement as if they had equal significance

- This ambiguity, again, is resolved by adding new nonterminals and rules

- Different categories of (matched – unmatched) of statements must be represented by different

abstractions

- The grammar below removes the ambiguity

```
    <stmt> ->  <matched> | <unmathced>
 <matched> ->  if <logic_expr> then <matched> else <stmt>
             |  <asgn_stmt>
             |  <while_stmt>
             |  ...
<unmatched> ->  if <logic_expr> then <stmt>
```

2.3.5 EBNF – Extended BNF

- BNF is not the only notation used for describing syntax of programming languages
- Due to few inconveniences, BNF is sometimes extended
- Actually there are several extensions that are based on similar ideas, however differing in the actual notation
- Most widely used extensions are for
  1. optional constructs
  2. repeating constructs
  3. alternative constructs

- **Optional Constructs:** Note that "else" part of a "if" statement is optional. Similar optional constructs appear frequently in the syntax description of programming languages (e.g. semicolon at the end, fpar list of a procedure declaration)

```
<if_stmt> -> if <logical_expr> then <stmt> [ else <stmt> ]
```

which allows using a single rule (as opposed to two rules we've been using)

```
<func_decl> -> [ <type> ] tIDENT tLPAR [ <fpar_list> ] tRPAR
               tLBRACE [ <stmt_list> ] tRBRACE
```

- **Repeating constructs:** Lists of some syntactic constructs are typically used while describing the syntax of programming languages. The part that can be repeated any number of times (including 0) is embraced in curly braces.

Example: List of identifiers

```
<ident_list> -> identifier { , identifier }
```

which saves us one rule

• **Alternative constructs:** Another typical feature required is to be able to pick from a group of constructs

Example: In Pascal, "for" statements have a syntax like the following

```
for x := 1 to limit do ...
for y := 10 downto 1 do ...
```

which normally needs two rules

```
  <for_stmt> ->  for tIDENT := <expr> <toORdownto> <expr> do <stmt>
<toORdownto> ->  to | downto
```

Using EBNF it can be given in a single rule

```
<for_stmt> ->  for tIDENT := <expr> ( to | downto ) <expr> do <stmt>
```

Example: Consider the following BNF grammar for expressions

```
<expr> ->  <expr> + <term>
        |  <expr> - <term>
        |  <term>
<term> ->  <term> * <factor>
        |  <term> / <factor>
        |  <factor>
```

• Note that, the rules for `<expr>` generate a list of `<term>`'s that are separated by either a `+` or a `-`. Similarly, the rules for `<term>` generate a list of `<factor>`'s that are separated by either a `*` or a `-`. After this observation, it is easy to write the same grammar in EBNF using only two rules

```
<expr> ->  <term> { ( + | - ) <term> }
<term> ->  <factor> { ( * | / ) <factor> }
```

• Some EBNF notations use `+` superscript to denote 1 or more repetitions

Example: Assume that a compound statement in a programming language is given by giving a non empty list of statements between keywords "begin" and "end"

```
                    either :  <compound> ->  begin <stmt> { <stmt> } end
or (when + is available):  <compound> ->  begin { <stmt> }+ end
```

2.3.6 Syntax graphs

- Using a graphical notation to describe grammars is also compelling for easy readability

- Syntax graphs, or syntax diagrams, or syntax charts (all equivalent) are also used as a graphical notation for grammars

- For each nonterminal a separate graph is given

- Nonterminals are given by circular or elliptic nodes

- Terminals are given by rectangular nodes

Example: Consider the following EBNF description for the "if" statements in Ada

```
<if_stmt> ->  if <logical_expr> then <stmts> { <else_if> } [ else <stmts> ] endif
<else_if> ->  elsif <logical_expr> then <stmts>
```

2.4 Parsing

- Recall that first two phases of a compiler are "lexical analysis" and "syntax analysis".

- We have seen that the purpose of a lexical analyzer is to convert the actual character sequence in a program text into a token sequence.

- The purpose of a syntax analyzer (also called as parser) is to check if the program is correct with respect to the grammar.

- Note that, in order to decide if a program is correct or not with respect to a grammar, it is not always necessary to know the exact lexemes.

- For both of the declarations below

```
int x;
int mean;
```

grammatical correctness can be checked by knowing that they both correspond to the sequence of tokens

```
tINT tIDENT tSEMI
```

That is, the lexeme of the `tIDENT` is not important to check grammatical correctness.

This is why, the actual program text is converted into a token sequence.

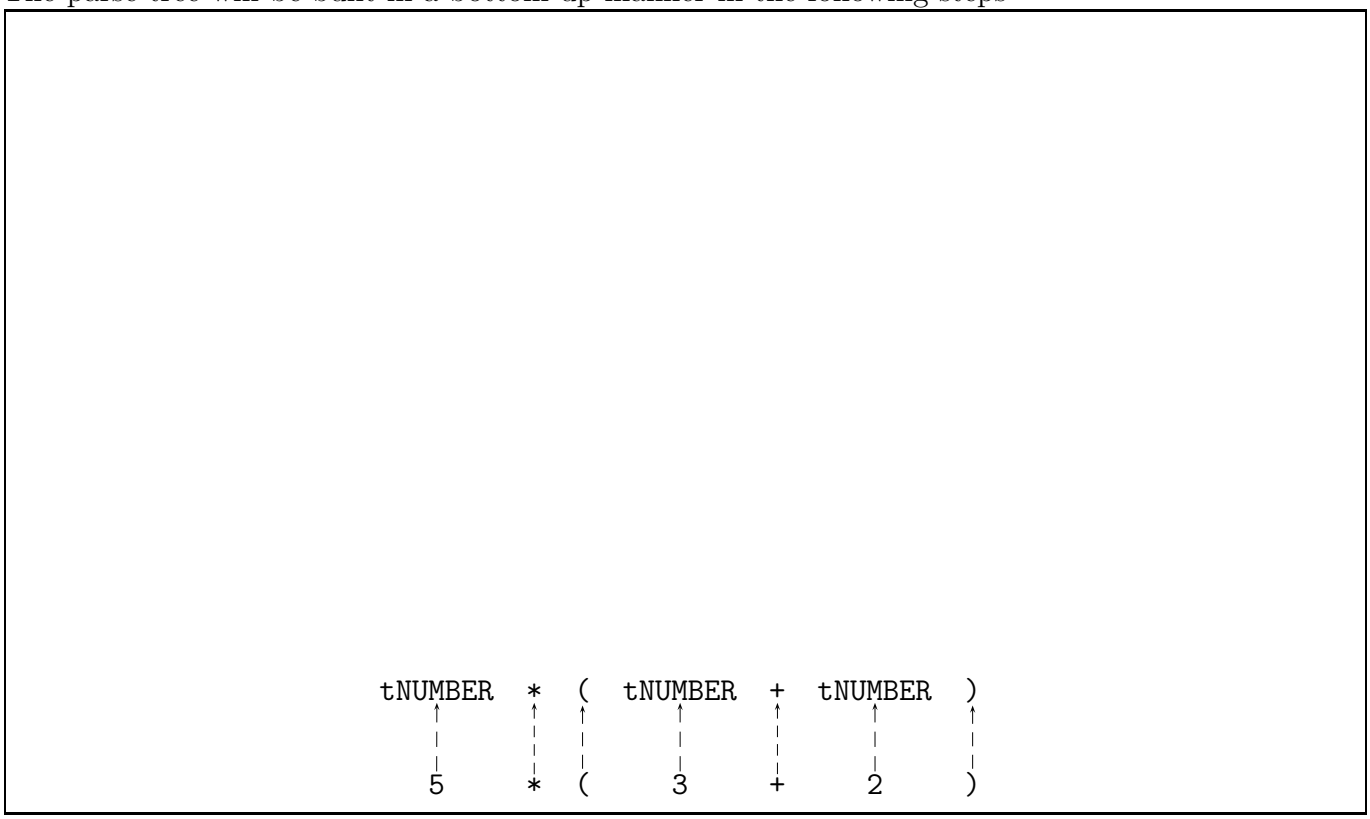
- A parser receives the token sequence, and checks if the program is grammatically correct or not.
- This check is based on the construction of a parse tree for the input using the grammar given
- A parse tree can be build using one of two approaches: bottom–up and top–down
- In bottom–up parsers, the tree is built starting from the token sequence and working upwards. The parsing is successful if the tree can be completed and the root is labelled by the start symbol of the grammar.
- For example consider the following grammar

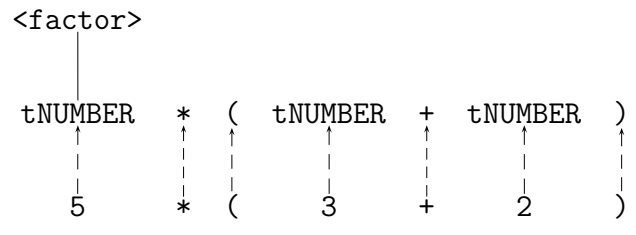
```
  <expr> ->  <expr> + <term>
           |  <term>
  <term> ->  <term> * <factor>
           |  <factor>
<factor> ->  ( <expr> )
           |  tNUMBER
```

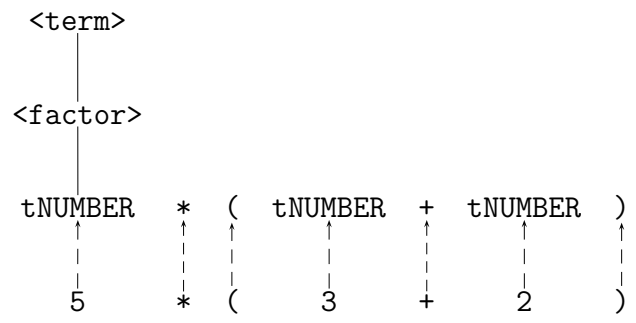
and the input

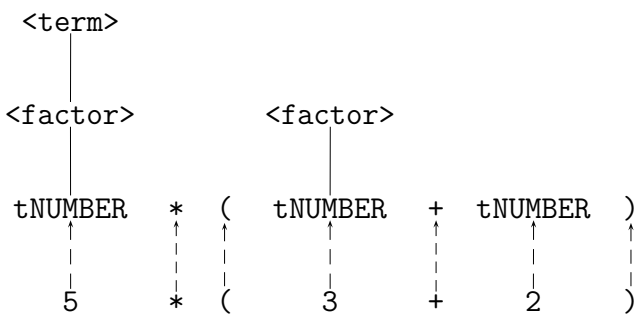
```
5 * ( 3 + 2 )
```

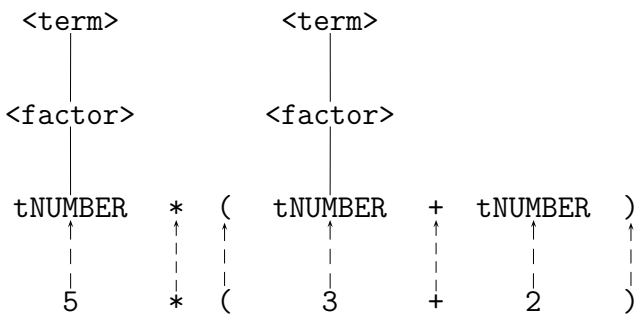
- The parse tree will be built in a bottom up manner in the following steps

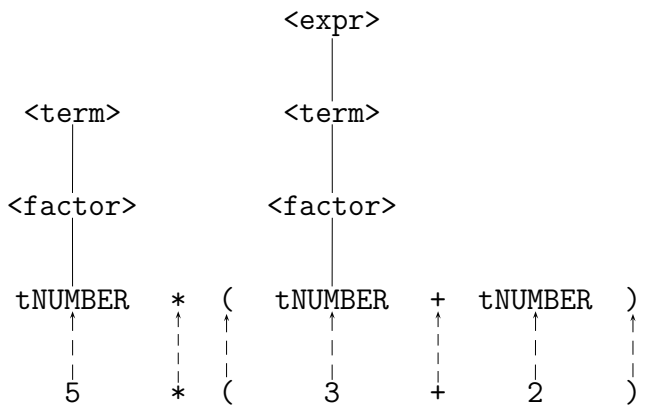


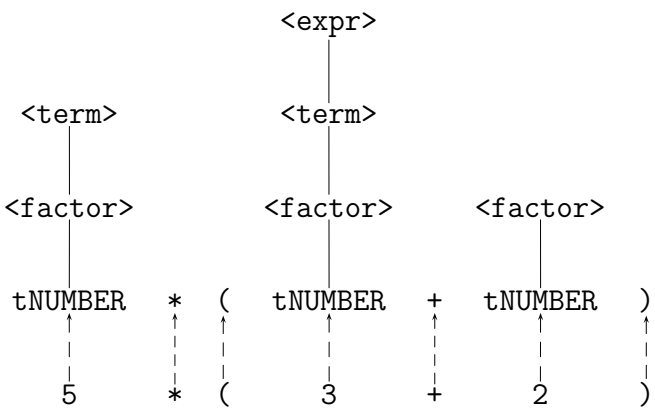

<factor>
tNUMBER * (tNUMBER + tNUMBER)
5 * (3 + 2)

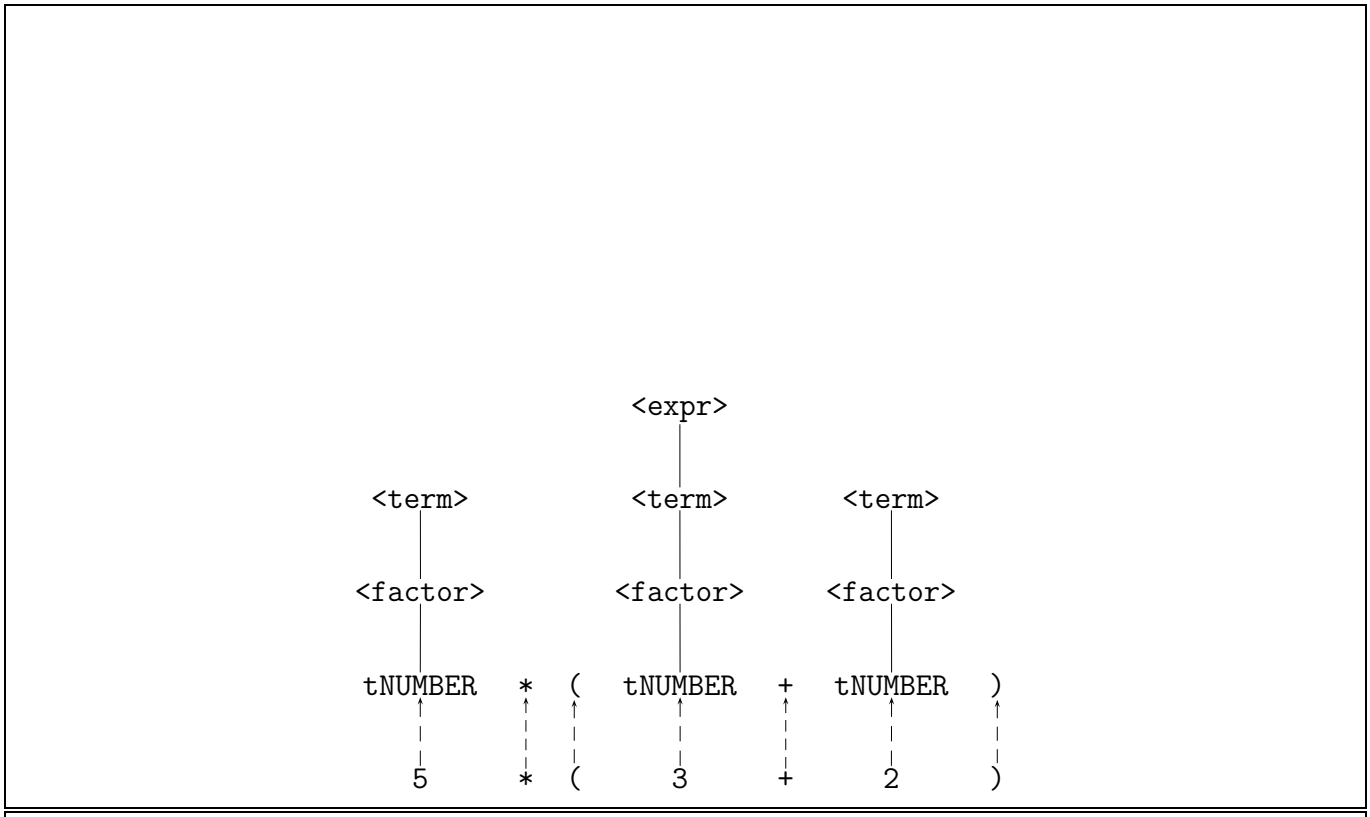

<term>
<factor>
tNUMBER * (tNUMBER + tNUMBER)
5 * (3 + 2)

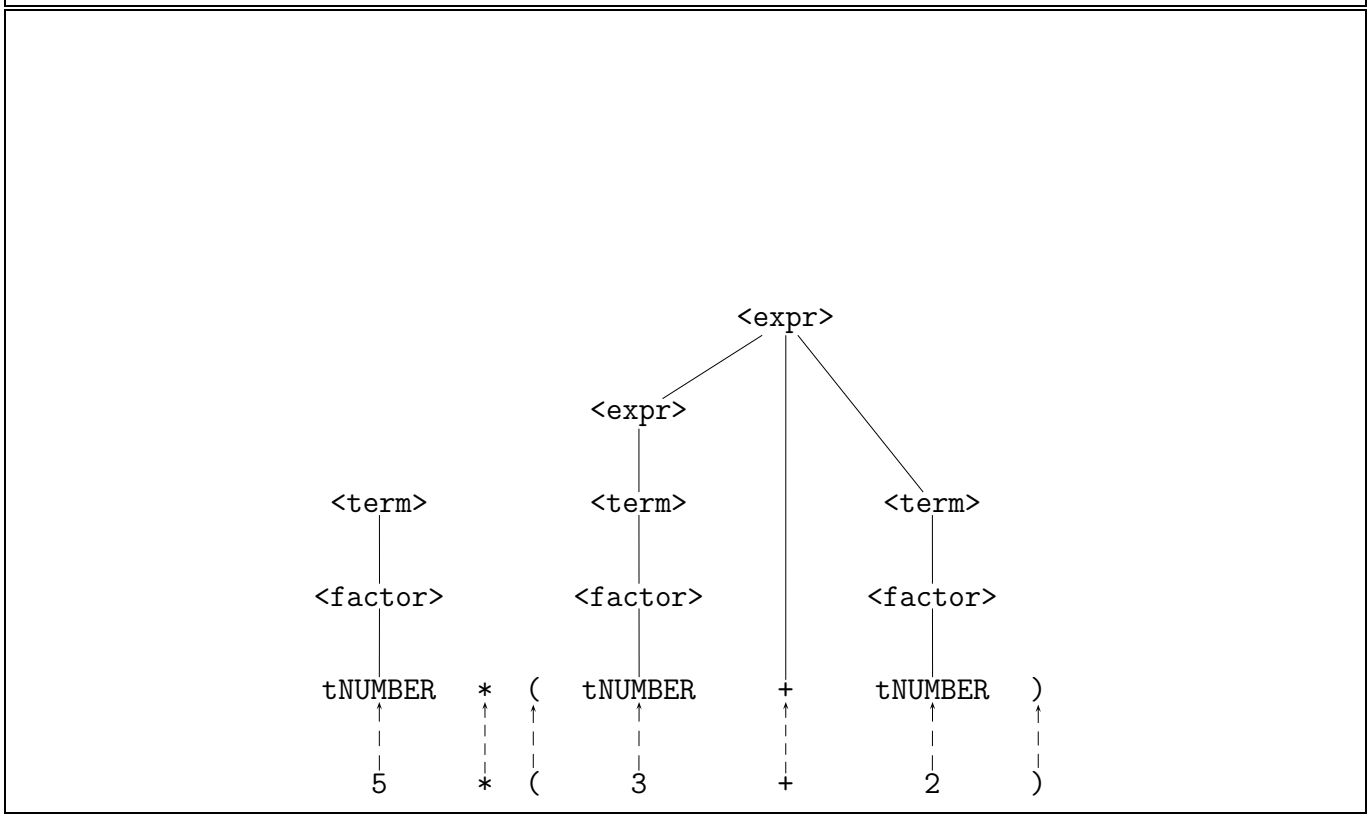

<term>
<factor>
<factor>
tNUMBER
*
(
tNUMBER
+
tNUMBER
)
5
*
(
3
+
2
)

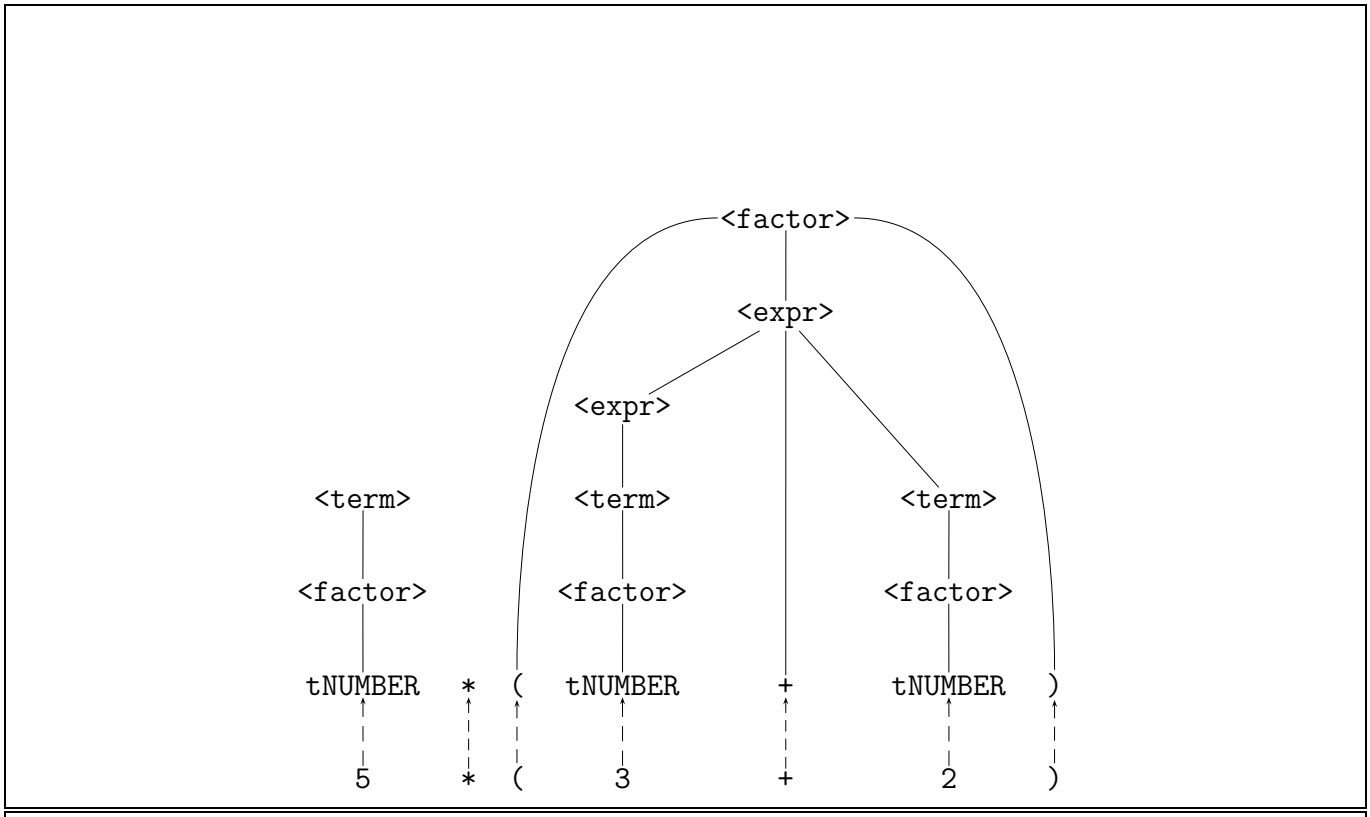

<term>
<term>
<factor>
<factor>
tNUMBER
*
(
tNUMBER
+
tNUMBER
)
5
*
(
3
+
2
)

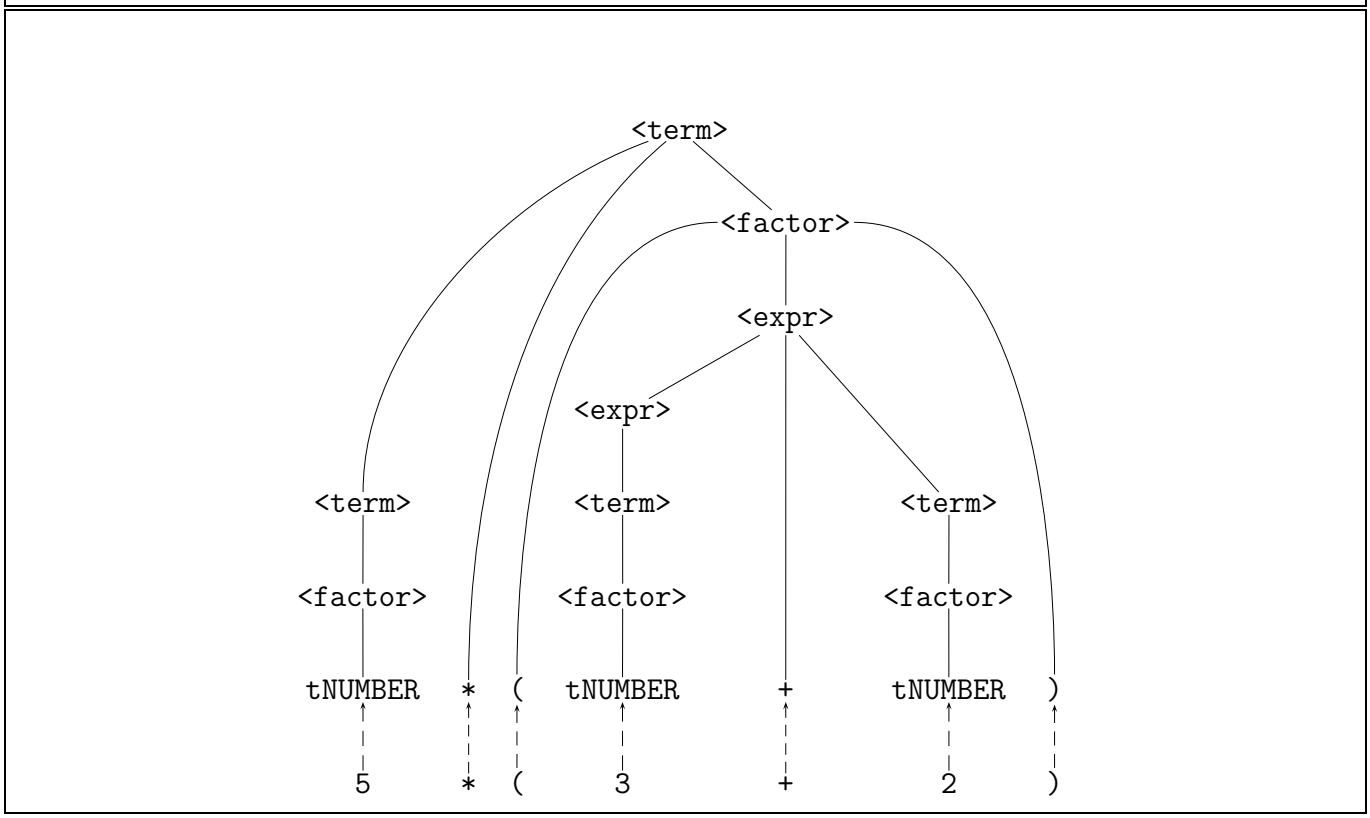

<expr>
<term>
<term>
<factor>
<factor>
tNUMBER
*
(
tNUMBER
+
tNUMBER
)
5
*
(
3
+
2
)

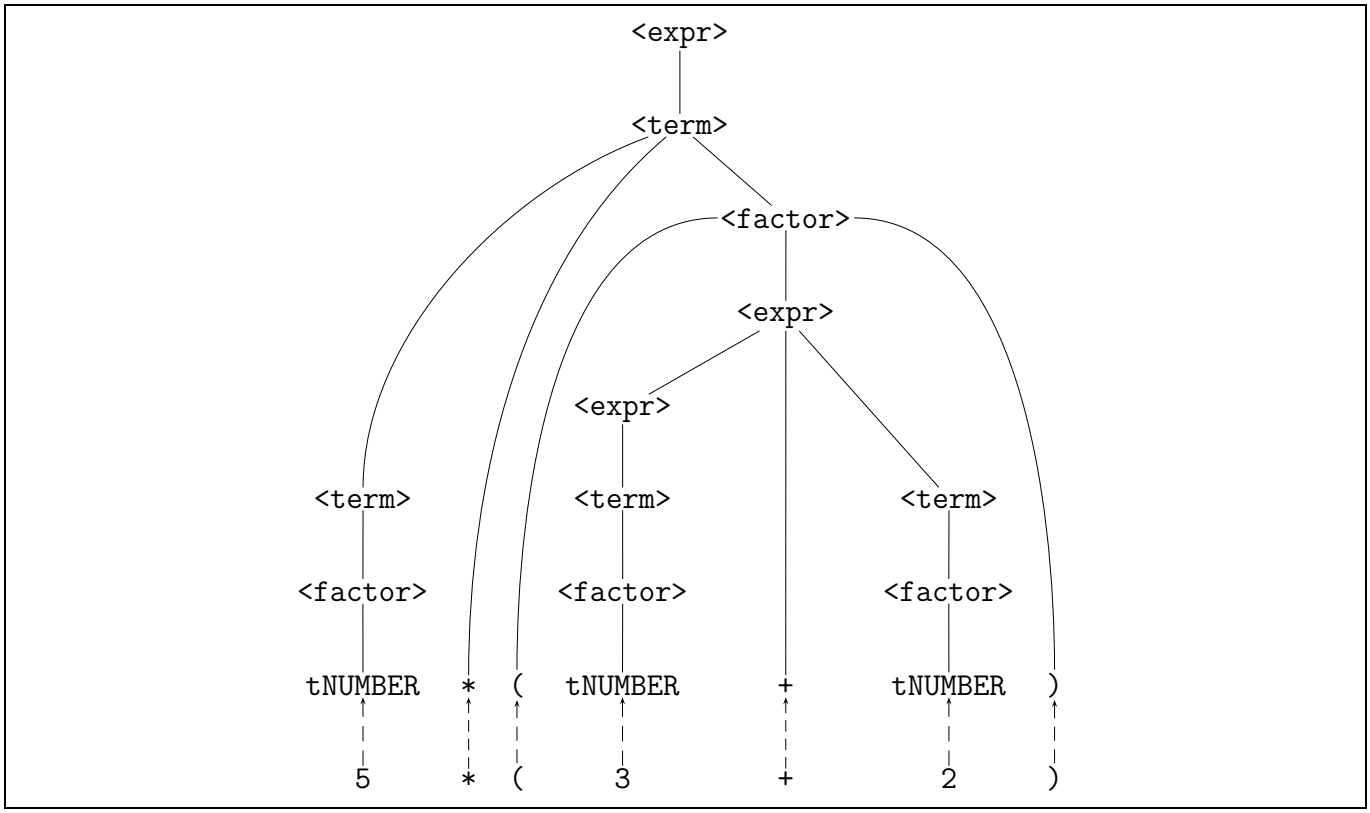

<expr>
<term>
<term>
<factor>
<factor>
<factor>
tNUMBER
*
(
tNUMBER
+
tNUMBER
)
5
*
(
3
+
2
)


<expr>
<term>
<term>
<term>
<factor>
<factor>
<factor>
tNUMBER
*
(
tNUMBER
+
tNUMBER
)
5
*
(
3
+
2
)

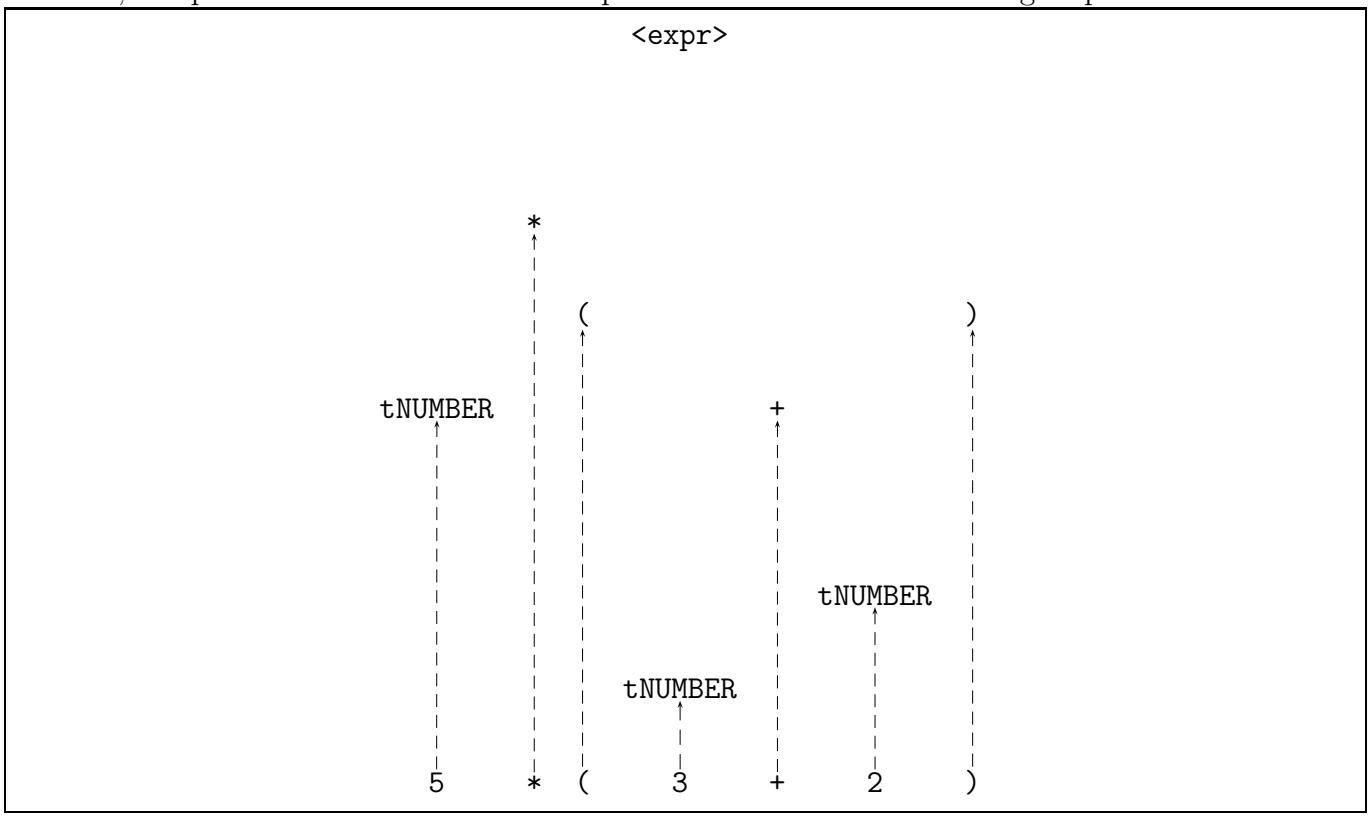

<expr>
<expr>
<term>
<term>
<term>
<factor>
<factor>
<factor>
tNUMBER
*
(
tNUMBER
+
tNUMBER
)
5
*
(
3
+
2
)

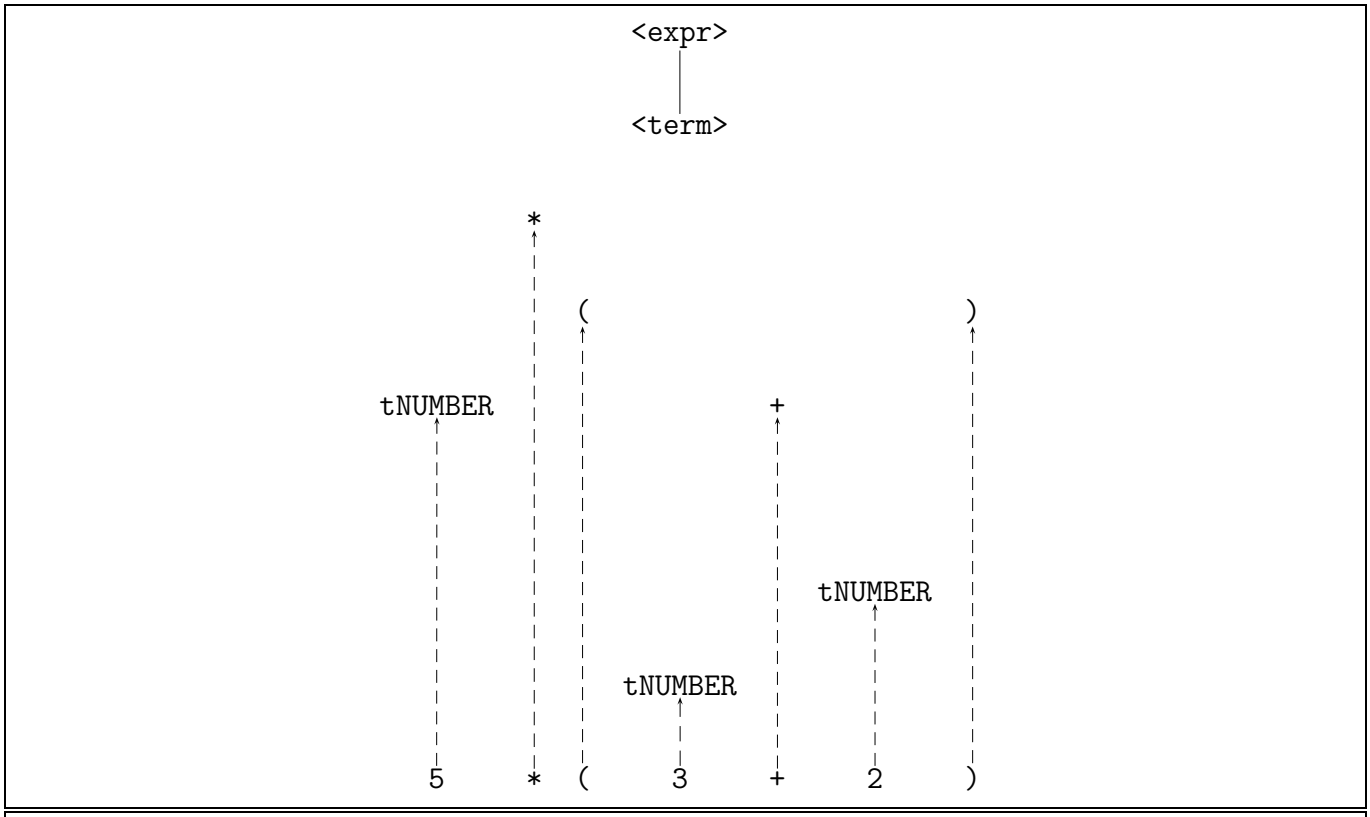

<factor>
<expr>
<expr>
<term>
<term>
<term>
<factor>
<factor>
<factor>
tNUMBER
*
(
tNUMBER
+
tNUMBER
)
5
*
(
3
+
2
)

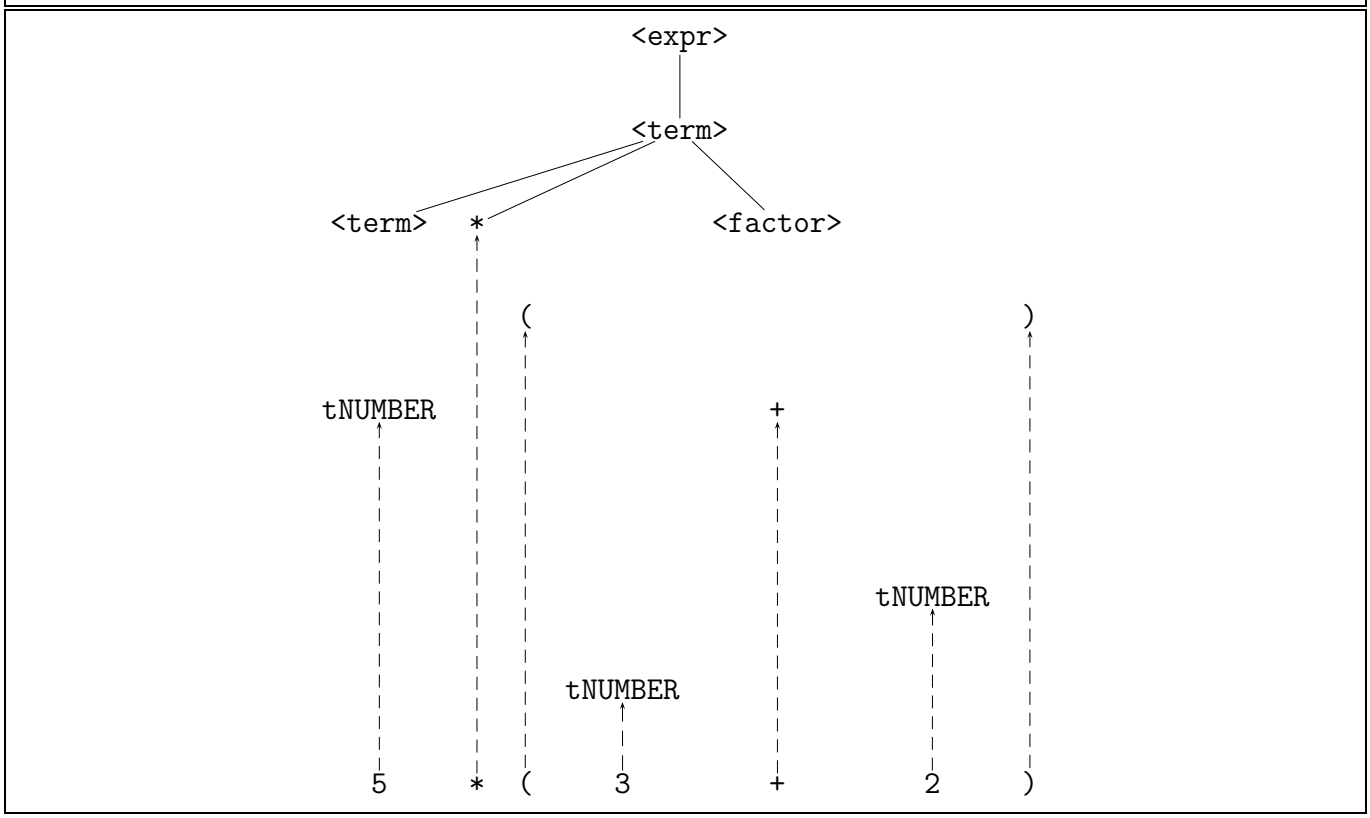

<term>
<factor>
<expr>
<expr>
<term>
<term>
<term>
<factor>
<factor>
<factor>
tNUMBER
*
(
tNUMBER
+
tNUMBER
)
5
*
(
3
+
2
)

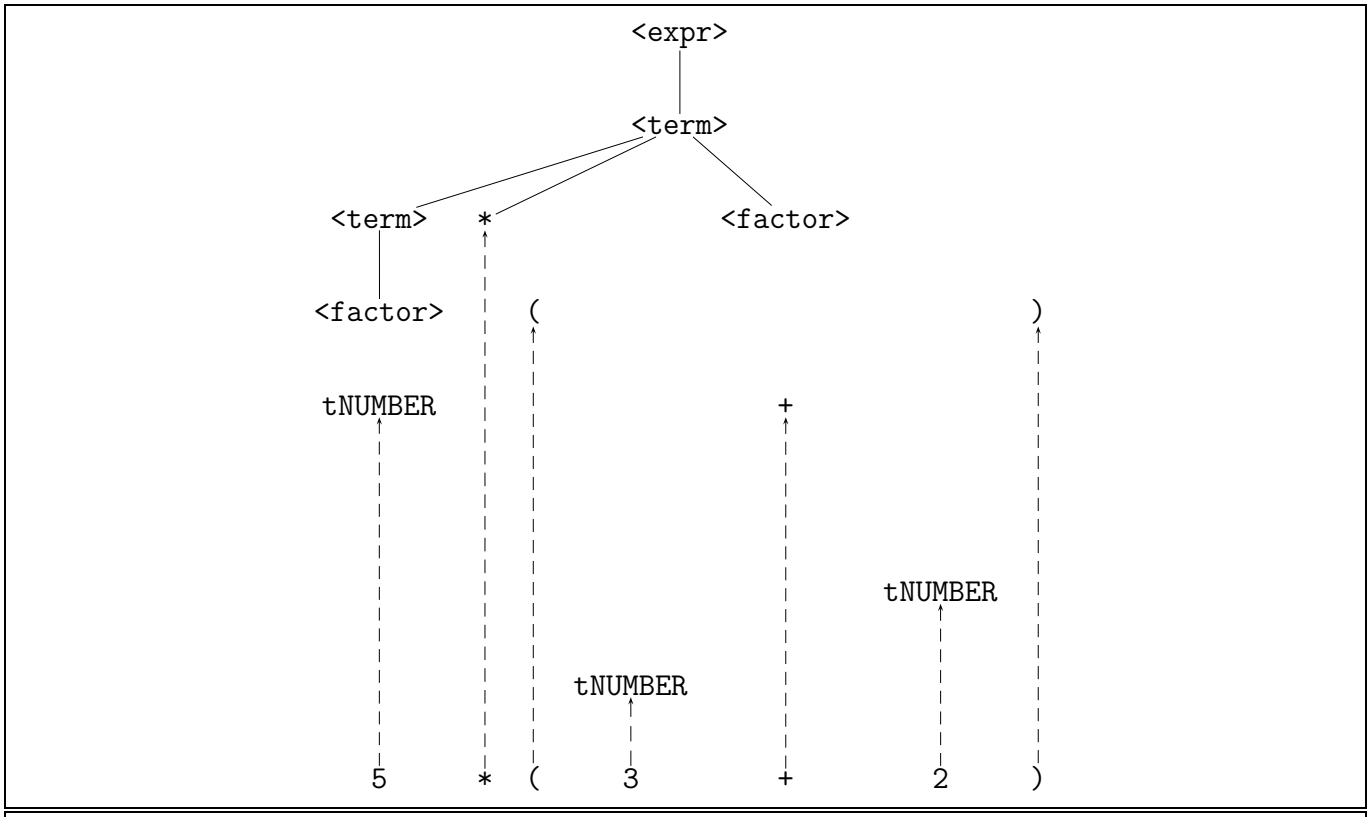


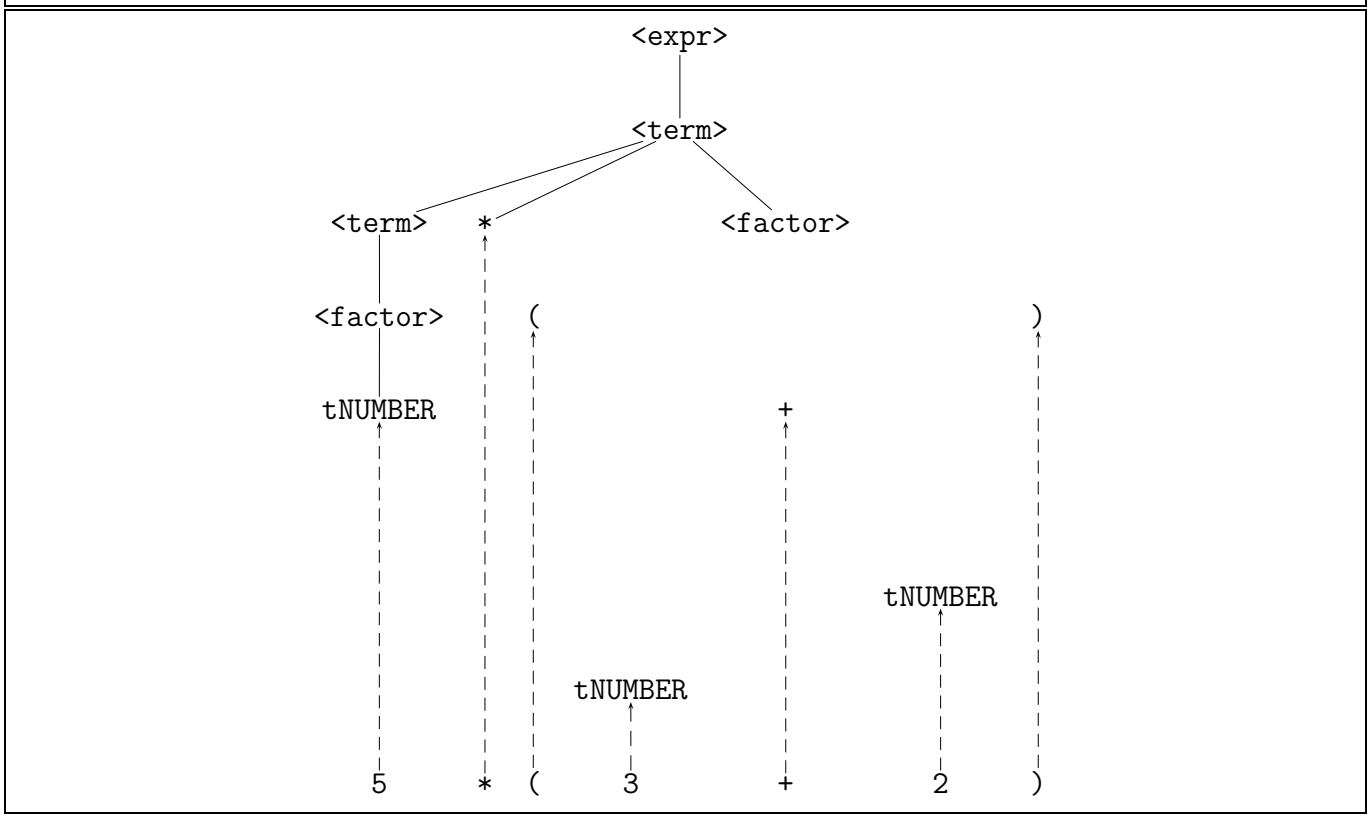
- However, the parse tree will be built in a top down manner in the following steps

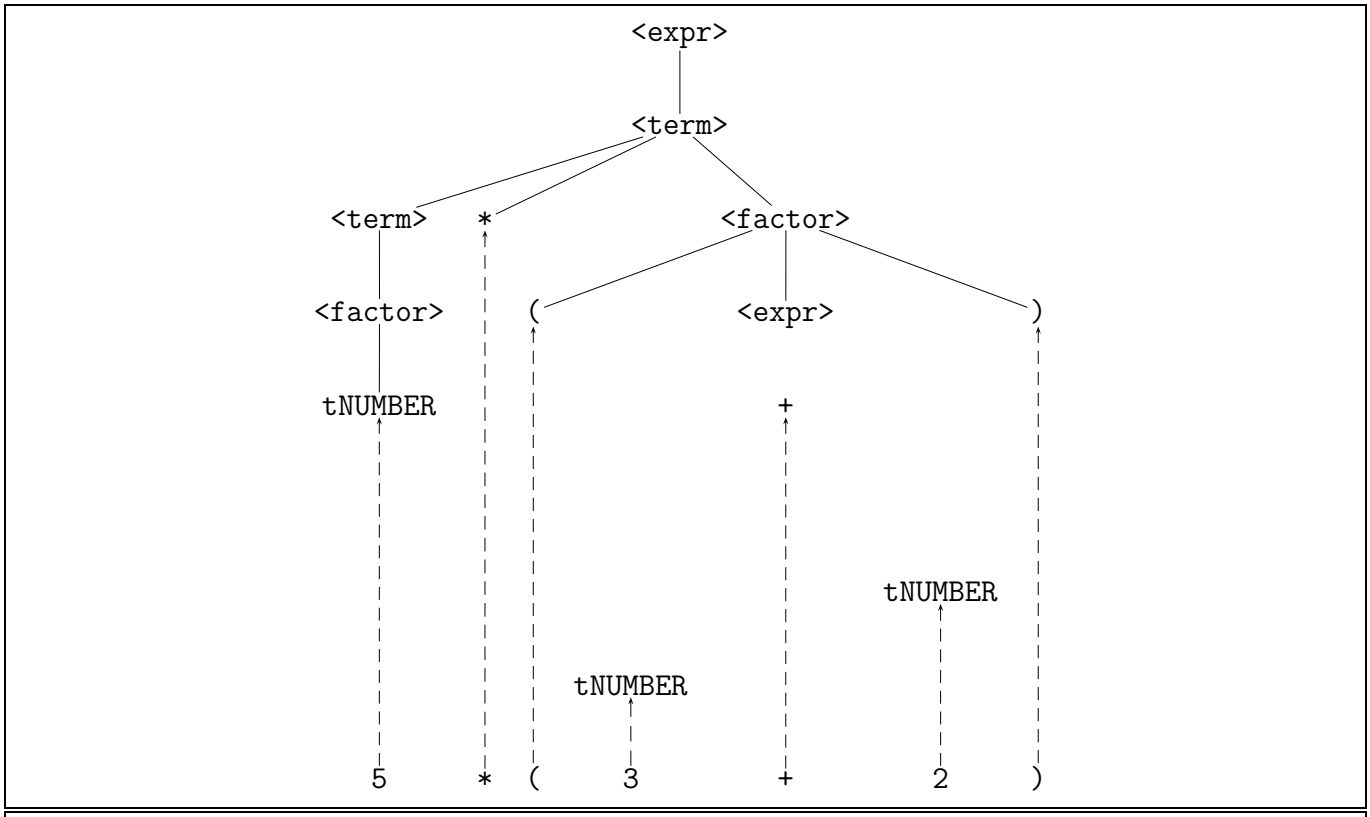


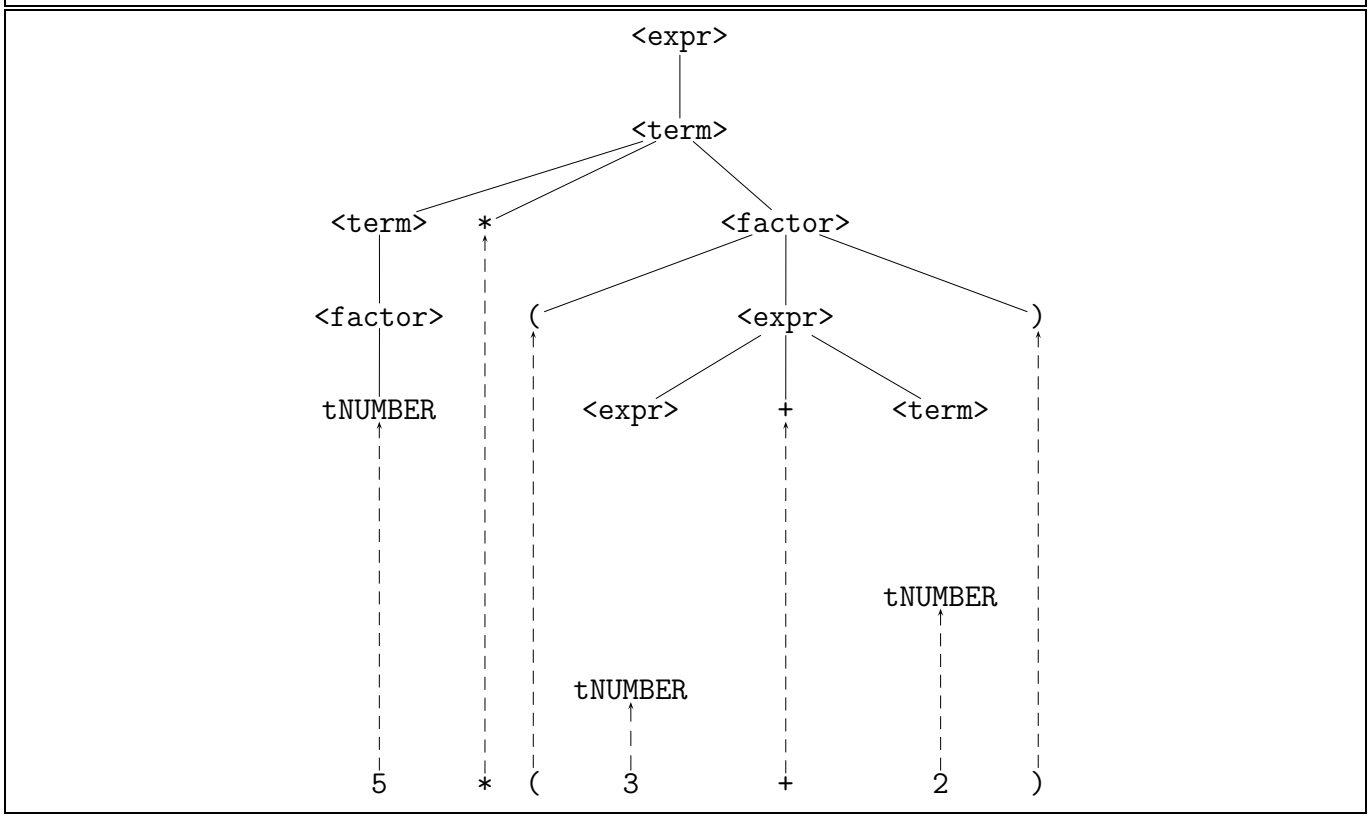

<expr>
<term>
*
(
)
tNUMBER
+
tNUMBER
tNUMBER
5
*
(
3
+
2
)

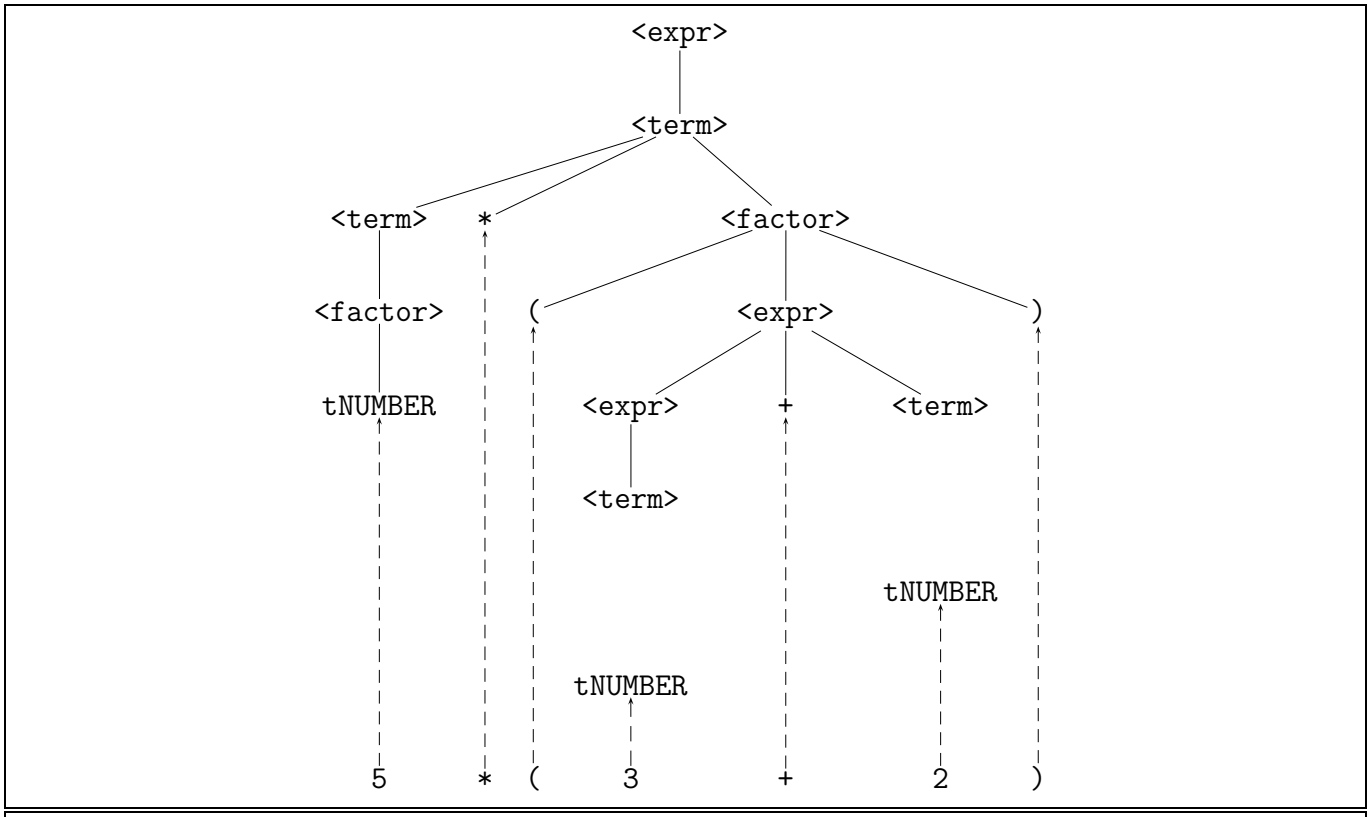

<expr>
<term>
<term>
*
<factor>
(
)
tNUMBER
+
tNUMBER
tNUMBER
5
*
(
3
+
2
)

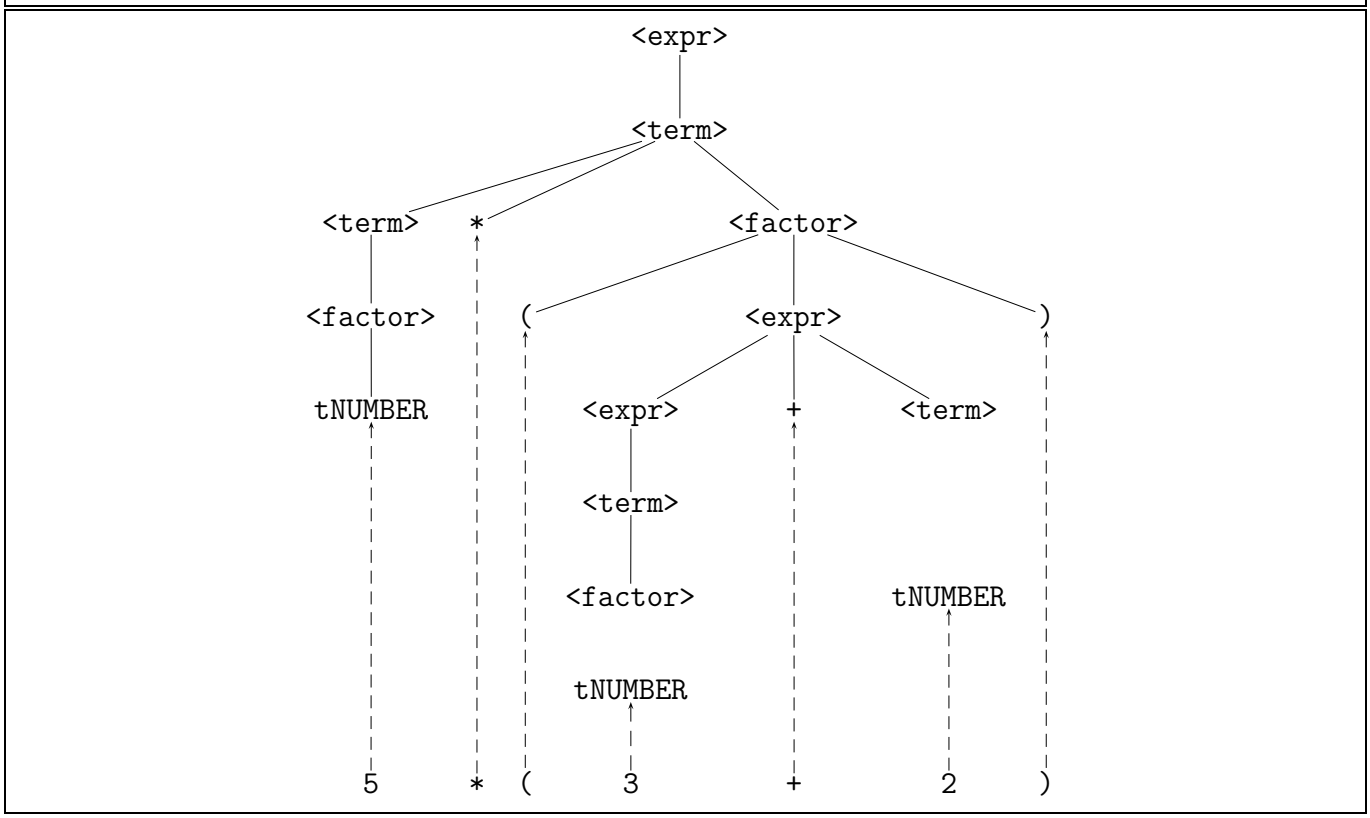

<expr>
<term>
<term>
*
<factor>
<factor>
(
)
tNUMBER
+
tNUMBER
tNUMBER
5
*
(
3
+
2
)

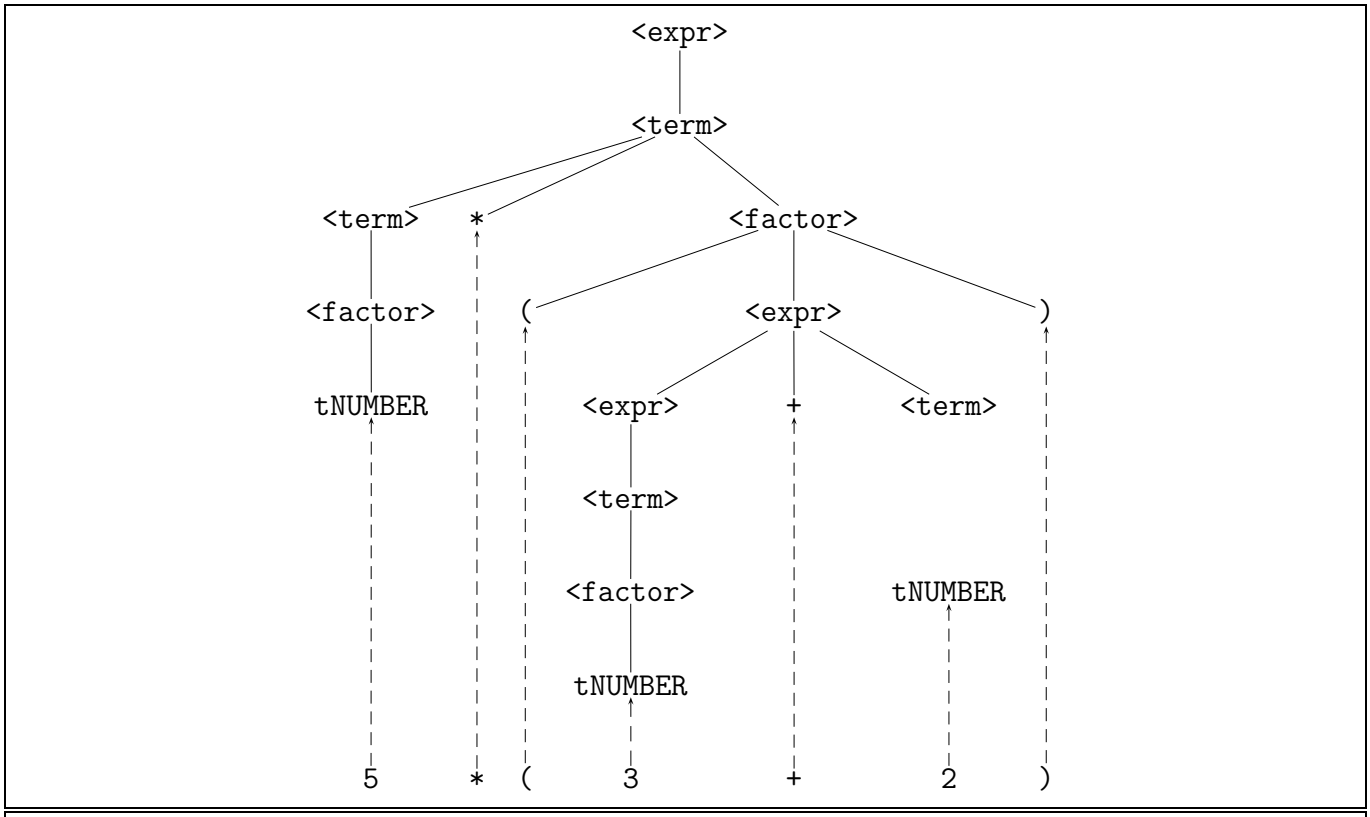

<expr>
<term>
<term>
*
<factor>
<factor>
(
)
tNUMBER
+
tNUMBER
tNUMBER
5
*
(
3
+
2
)

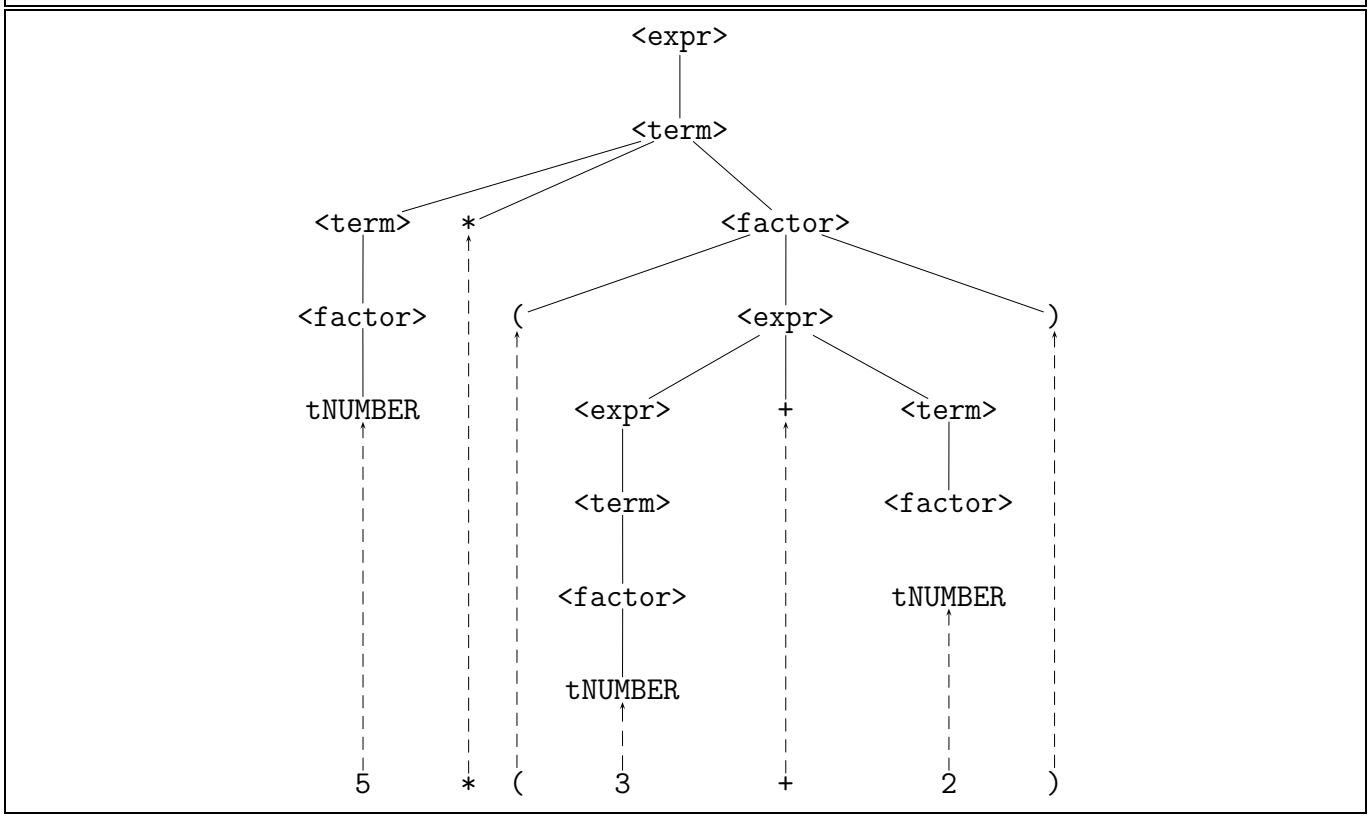

<expr>
<term>
<term>
*
<factor>
<factor>
(
<expr>
)
tNUMBER
+
tNUMBER
tNUMBER
5
*
(
3
+
2
)

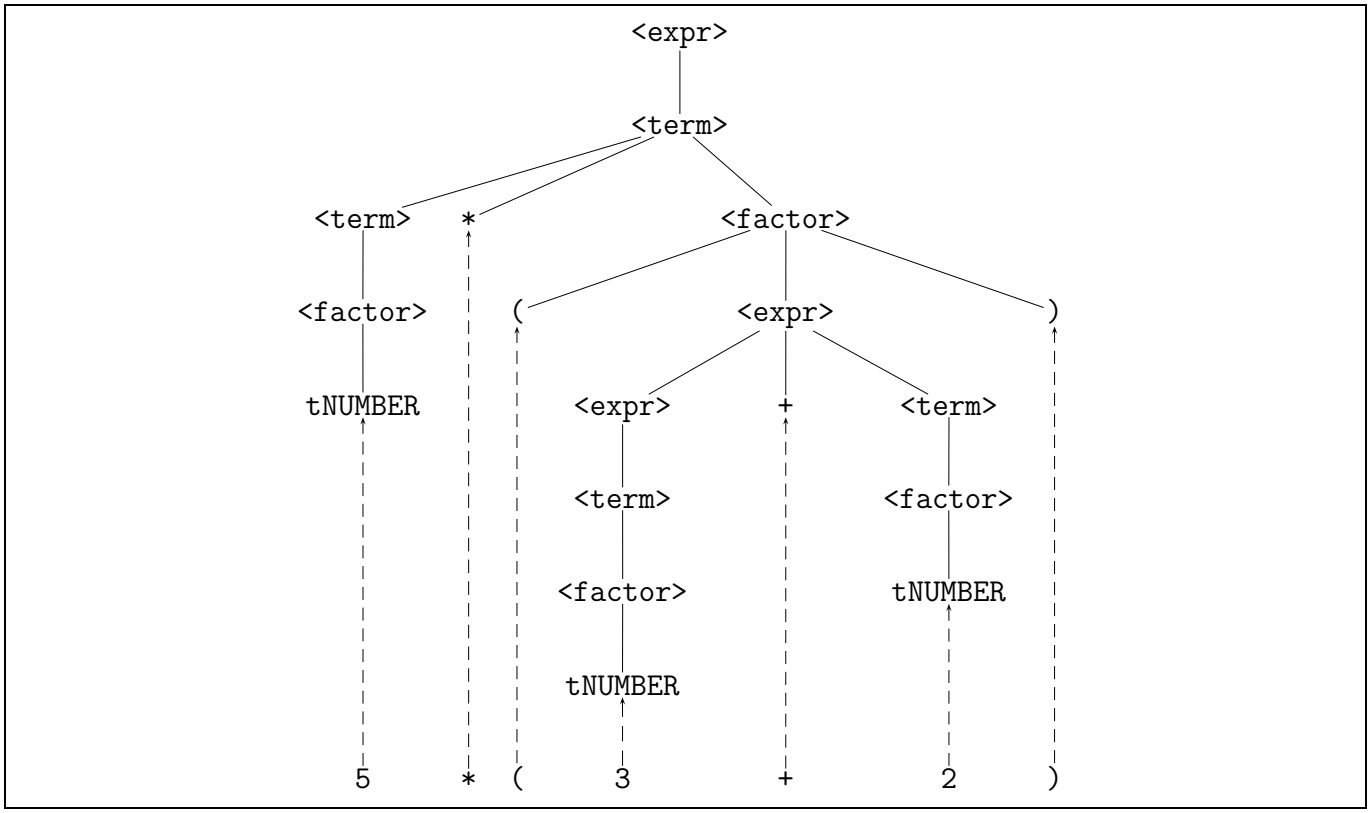

<expr>
<term>
<term>
*
<factor>
<factor>
(
<expr>
)
tNUMBER
<expr>
+
<term>
tNUMBER
tNUMBER
5
*
(
3
+
2
)


<expr>
<term>
<term>
*
<factor>
<factor>
(
<expr>
)
tNUMBER
<expr>
+
<term>
<term>
tNUMBER
tNUMBER
5
*
(
3
+
2
)


<expr>
<term>
<term>
*
<factor>
<factor>
(
<expr>
)
tNUMBER
<expr>
+
<term>
<term>
<factor>
tNUMBER
tNUMBER
5
*
(
3
+
2
)


<expr>
<term>
<term>
*
<factor>
<factor>
(
<expr>
)
tNUMBER
<expr>
+
<term>
<term>
<factor>
tNUMBER
tNUMBER
5
*
(
3
+
2
)


<expr>
<term>
<term>
*
<factor>
<factor>
(
<expr>
)
tNUMBER
<expr>
+
<term>
<term>
<factor>
<factor>
tNUMBER
tNUMBER
5
*
(
3
+
2
)

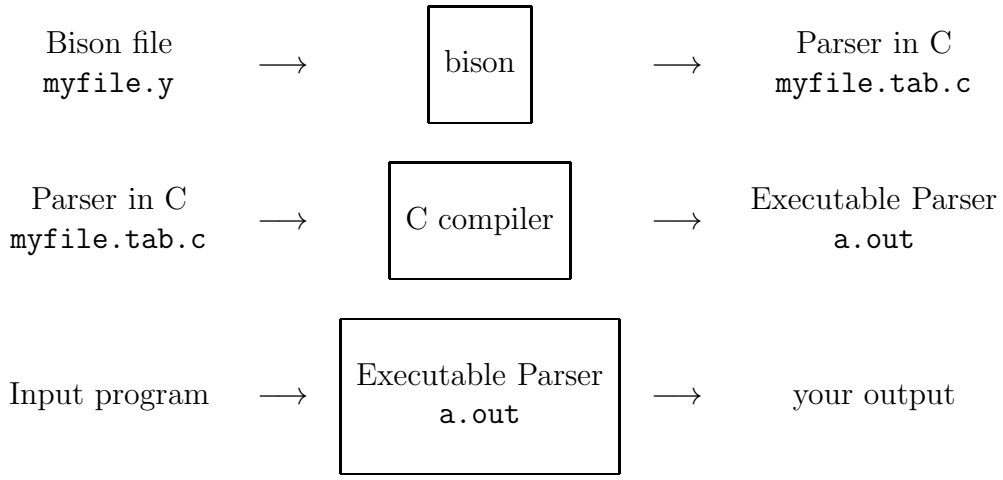


2.5 Bison Introduction

- It is possible to write a parser for a given grammar from scratch in C or in some other programming language.
- However, there are parser generator tools : yacc, bison, etc...
- Given a grammar, these tools generate the parser automatically.
- We will see bison as an example.
- However, using other similar tools are easy and quite similar to the bison.
- Similar to flex, bison generates an output file in C.
- This C file generated is the parser for the given grammar.
- The general form of a bison file is as follows:

```
%{
C declarations
%}
Bison declarations
%%
Grammar rules
%%
Additional C code
```

- "`C declarations`" part is for general C definitions (types, variables, that are required)
- "`Bison declarations`" part is for declaring the grammar symbols
- "`Grammar rules`" part is for the grammar for which the parser will be generated
- "`Additional C code`" part is for any other C functions that are required
- Bison supports attribute grammars: it is possible associate attributes with symbols and semantic functions with grammar rules
- Typical flow to compile bison files



- Note that, a parser is supposed to deal with a token sequence. The token sequence is provided by scanner.
- Although we have mentioned before that a scanner is the first phase of a compiler and it converts an input character stream (the input text, the source program) into a token stream, which is in turn received by the second phase of the compiler, i.e. the parser, this is not how things exactly work in practice.
- In practice, a scanner and a parser work interactively in the following way: First the parser is

started. The parser requests a token from the scanner when it requires one. The scanner starts working and identifies and returns a token to the parser, and the control is passed to the parser. Then the parser continues its work until it needs another token. To receive the next token, parser calls the scanner again. Therefore, the scanner actually works as a function called by the parser.

- Bison generates a C file `filename.tab.c`, in which there is a function called `yyparse` which is the actual parser function, and it must be called to start the parsing.

```
int main() {
   if (yyparse()) {
      // yyparse returns 1 if there is an error
      // parse error
      }
   else {
     // yyparse returns 0 if the parsing is completed successfully
     // successful parsing
     }
   }
```

- The parsers generated by bison will automatically call the scanner when a token is needed to continue parsing.

- In order to get the next token, `yyparse` calls a function named `yylex`. That is, the parsers generated by bison uses a scanner named `yylex` by default.

- You can write a scanner function by hand, and name it as `yylex`.

- However, flex generated scanner functions can also be used easily (recall that flex generated scanners are already named `yylex`).

- Let us see now our first example on the usage of bison.

- We will write a parser for expressions given in postfix notation

```
3 5 +
3 5 + 2 -
3 5 + 4 6 - +
```

- For simplicity, let's restrict ourselves to the positive integer numbers and to addition and subtraction operators.

- An input file in our example is supposed to consist of a single expression in postfix form

- Here is our first bison file, let's assume that it is saved as the file `ex1.y`:

ex1.y

```
%{
#include <stdio.h>
void yyerror (const char *s)  /* Called by yyparse on error */
{
  printf ("%s\n", s);
}
%}
%token tNUMBER tADD tSUB
%% /* Grammar rules and actions follow */
expr:     tNUMBER
        | expr expr tADD
        | expr expr tSUB
;
%%
int main ()
{
   if (yyparse()) {
      // yyparse returns 1 if there is an error
      // parse error
      printf("Could not parse the input.\n");
      return 1;
      }
   else {
     // yyparse returns 0 if the parsing is completed successfully
     // successful parsing
     printf("Parsing finished successfully.\n");
     return 0;
     }
}
```

- `yyerror` must be supplied by the user and must accept a `char *` parameter. It is called by `yyparse` when a parsing error occurs. The parameter value is normally set to `"parse error"` by `yyparse`.

- In the "Bison Declarations" part we see
`%token tNUMBER tADD tSUB`
which is used to declare tokens used in the grammar. This line states that, within the grammar

part, there will be tokens named `tNUMBER`, `tADD` and `tSUB`.

- By convention, the token names are written in capital letters in bison.

- As can be seen from the example, multiple tokens are given by separating them with a space.

- The grammar is pretty clear and simple: there are three productions given by using alternative operator `|` as the LHS of all the productions have the same nonterminal `expr`.

- The productions for a nonterminal terminated by a semicolon.

- In the "Additional C Code" part, we have the declaration of the `main` function, which simply starts the parsing.

- Now, we come to the scanner part. Let us assume that, our input can be given with spaces between numbers and operators, and newline characters are also allowed.

- So, the scanner has to filter out such white space characters.

- It has to catch numbers and when it is sees one, it has to return the token `tNUMBER` to the parser.

- Similarly, for the other tokens, when our scanner sees a "+" or a "-" character in the input, it has to return the tokens `tADD` and `tSUB`, respectively.

- In addition to these, the scanner has to pass any other character to the parser, and let the parser decide if they are addition or subtraction operators in expected places (which will denote a successful parsing) or something unexpected (which will end up in an error in parsing)

- Here is a flex specification that can be used as a scanner (let's assume that it is saved as file `ex1.flx`)

ex1.flx

```
%{
/* get the token definitions from bison output file */
/* tokens defined in ex1.tab.h are: tNUMBER, tADD, tSUB */
#include "ex1.tab.h"
%}
%%
[0-9]+    return tNUMBER; /* caught a number */
"+"       return tADD;    /* caught an addition */
"-"       return tSUB;    /* caught a subtraction */
[ \t\n]+  /* eat up white space */
.         {/* pass any other character to the parser */
          return yytext[0];
          }
%%
```

- It returns the token `tNUMBER` when it sees a decimal number. When the characters "`+`" or "`-`" are seen, the scanner will return the corresponding tokens (i.e. `tADD` and `tSUB` respectively).

- However, what is `tNUMBER`, i.e. how the flex file (or the C file `lex.yy.c` generated from this flex file) knows about it?

- Tokens are represented as numbers in practice. That is each token is given a unique number. When a scanner returns a number to the parser as the token, the parser understands the token from this number.

- Hence the token `tNUMBER` will be given a unique number automatically by a statement of the form

```
#define tNUMBER 258
```

or of the form

```
enum tokentype { tNUMBER = 258; }
```

(or some other number)

- These definitions for tokens are placed in the file `ex1.tab.h` that will be generated by bison.

- Hence, by `#include`ing this file `ex1.tab.h` all the token numbers become visible to the scanner as well.

- We have a rule for hiding the white space from the parser.

- The last rule is for passing all the other characters to the parser directly. Recall that, `yytext` is an automatic variable in flex, and it keeps the lexeme of the pattern matched. By passing `yytext[0]`,

we will be passing the actual character seen in the input, since this rule matches a single character.

- Note that, we didn't need to give a `main` function in flex file as we used to do in stand alone scanner examples we had before.

- The reason is that, `yylex` function, which we used to call in these `main` functions of stand alone scanner implementations, will now be called by the parser.

- Now, how do we produce the executable parser from `ex1.y` and `ex1.flx`?

- The following steps are used:

```
bison -d ex1.y
flex ex1.flx
gcc -o ex1 lex.yy.c ex1.tab.c -lfl
```

The option `-d` of bison produces the `ex1.tab.h` file which includes the unique number assignment definitions to tokens (without this option, it does not generate this file).

Flex usage is still the same.

Then we produce the executable (which is assumed to be named as `ex1` in this case) by compiling the parser and scanner C files generated by bison and flex together.

Tokens with single character fixed lexemes

- If the grammar is using a fixed lexeme token with a single character lexeme, then we can use the following trick to simplify the specification of the grammar and the scanner.

- Note that in our previous example, we used three tokens: `tNUMBER`, `tADD` and `tSUB`.

- Among these three, `tADD` and `tSUB` are fixed lexeme tokens and the lexeme for them are single characters. Using this fact, we can simplify the bison specification as follows:

SimpleTokens.y

```
%{
#include <stdio.h>
void yyerror (const char *s)  /* Called by yyparse on error */
{
  printf ("%s\n", s);
}
%}
%token tNUMBER
%% /* Grammar rules and actions follow */
expr:     tNUMBER
        | expr expr '+'
        | expr expr '-'
;
%%
int main ()
{
   if (yyparse()) {
      // parse error
      printf("Could not parse the input.\n");
      return 1;
      }
   else {
     // successful parsing
     printf("Parsing finished successfully.\n");
     return 0;
     }
}
```

- Note that, we removed the tokens `tADD` and `tSUB`, as we will not be using these tokens any more. Instead of using these tokens, we will be using the lexemes of these tokens directly in our grammar.

- With this modification, the grammar becomes more readable.

- We have to change the scanner specification in flex accordingly. Here is the new flex specification that has to be used with the modified parser:

SimpleTokens.flx

```
%{
/* get the token definitions from bison output file */
/* tokens defined in ex1.tab.h are: tNUMBER, tADD, tSUB */
#include "ex1.tab.h"
%}
%%
[0-9]+    return tNUMBER; /* caught a number */
[ \t\n]+  /* eat up white space */
.         {/* pass any other character to the parser */
          return yytext[0];
          }
%%
```

• This scanner misses the two rules for the recognition and return of the tokens `tADD` and `tSUB`. Since these tokens are now represented by their actual lexemes in the parser, the scanner now has to pass the lexemes of these tokens to the parser. However, this is already handled by the last rule in the scanner. When a "`+`" or "`-`" is seen in the input, it will be matched by the last rule. The character will directly be passed to the parser by the execution of the statement "`return yytext[0];`".

• However, for those tokens without a constant lexeme or with constant lexemes of multiple characters (e.g. the token `tNUMBER`), we should always declare a token name, and use the scanner to recognize and return these tokens.

2.6 Resolving ambiguities

• An important property of the postfix notation for expressions is that, it is quite easy to write an unambiguous grammar for it.

• However, as we have seen earlier, such an easy grammar for infix notation is ambiguous

```
<expr> -> tNUM
       |  <expr> + <expr>
       |  <expr> - <expr>
       |  <expr> * <expr>
```

• Let us try to write a bison parser with this simple grammar for expressions using infix notation

• First, the scanner `ex2.flx` (which is exactly the same as the scanner `ex1.flx` used for postfix notation)

ex2.flx

```
%{
/* get the token definitions from bison output file */
/* (actually there is only one which is tNUM in this example) */
#include "ex2.tab.h"

%}
%%
[0-9]+    return tNUM; /* caught a number */
[ \t\n]+  /* eat up white space */
.         { /* pass any other character to the parser */
            return yytext[0];
          }
%%
```

- Now we give the bison file `ex2.y`

ex2Ambiguous.y

```
%{
#include <stdio.h>

void yyerror (const char *s)  /* Called by yyparse on error */
{
  printf ("%s\n", s);
}
%}

%token tNUM
%% /* Grammar rules and actions follow */
expr:     NUM
        | expr '+' expr
        | expr '-' expr
        | expr '*' expr
;
%%
int main ()
{
  return yyparse ();
}
```

- Note that we added the multiplication operator as well in order to emphasize the priorities.

- When we use `ex2.flx` and `ex2.y` as given above, bison produces a warning "`shift/reduce conflict`" which is a reminder for a possible ambiguity in the grammar.

- We can rewrite the grammar by using `factor` and `term` nonterminals as we did before. However, this complicates the grammar and there is an easier way of resolving such ambiguities in bison.

- It is performed by declaring relative priority and associativity of the operators as follows:

ex2Unambiguous.y

```
%{
#include <stdio.h>

void yyerror (const char *s)  /* Called by yyparse on error */
{
  printf ("%s\n", s);
}
%}

%token NUM
%left  '+' '-'
%left  '*'

%% /* Grammar rules and actions follow */

expr:     NUM
        | expr '+' expr
        | expr '-' expr
        | expr '*' expr
;
%%

int main ()
{
  return yyparse ();
}
```

- In the bison declarations part, the keyword `%left` is used to give a left associative operator.

Similarly, `%right` is used to declare a right associative operator.

- Operators given on the same line have the same priority

- The priority of the operators given on the next line are higher than priority of the operators given on the previous line

2.7 Attribute Grammars

- An **Attribute grammar** is a device used to describe more structural features of a language than it is possible with a context free grammar.

- An attribute grammar is actually an extension of a context free grammar

2.7.1 Static Semantics

- The additional features that are described by attribute grammars are called as the **static semantics** of languages

- Static semantic features of a programming language are still related to the syntactic correctness (as opposed to dynamic semantics which is related to the run time behavior)

- Static semantic features are difficult to be described by context free grammars

Example: A floating point value cannot be assigned to an integer value

```
int i;
double d;
...
i = d;
```

This can be actually described using CFG, however it makes the description quite long.

Example: In Ada, if the **end** of a subprogram is followed by a name, then it must be the same as the name of the subprogram.
This rule cannot be described using CFG.

Example: A variable must be declared before it is used.
This is another typical rule for programming languages, and again it cannot be described by CFG.

- Because of the problems of describing static semantics using context free grammars, a variety of more powerful mechanisms have been devised.

- Attribute grammars is one such mechanism proposed by Knuth.

2.7.2 Basic Concepts

- Attribute grammars are CFG to which the following concepts have been added
  1. Attributes: variables associated with grammar symbols (used to pass information up and down in the parse tree)
  2. Attribute computation functions (semantic functions) : associated with productions and specify how the attributes are computed
  3. Predicate functions : associated with productions and specify the semantic rules

- They will become more clear as we explain them in more detail and give examples

- If you consider a parse tree whose nodes are labeled by grammar symbols, the attributes can be considered as additional labels for these nodes, or variables attached to these nodes. The attributes are used to store additional information in the parse tree. Since the values of the attributes are calculated based on each other, we pass information up and down in the parse tree using the attributes and attribute computations.

2.7.3 Attribute Grammars Defined

- Associated with each symbol X in the grammar is a set of attributes $A(X)$.

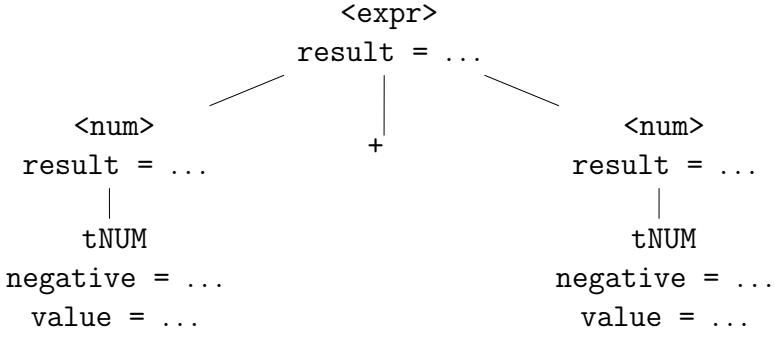
Example: Consider the following grammar

```
<expr> ->  <num> + <num>
 <num> ->  tNUM
```

There are three symbols in this grammar: `<expr>`, `<num>`, and `tNUM`. For each one of these symbols, there will be a set of attributes:

Attributes of `<expr>`	:	$A($`<expr>`$)$	=	{`result`}
Attributes of `<num>`	:	$A($`<num>`$)$	=	{`result`}
Attributes of `tNUM`	:	$A($`tNUM`$)$	=	{`value`, `negative`}

- $A(X)$ is the set of variables that would be attached to a parse tree node which is labeled by X.



- This set is partitioned into two parts: Inherited attributes $I(X)$, and synthesized attributes $S(X)$, as $A(X) = I(X) \cup S(X)$ where $I(X) \cap S(X) = \emptyset$.

- In other words, an attribute in $A(X)$ is either an inherited attribute or a synthesized attribute.

- Synthesized attributes are used to pass information upward in a parse tree.

- Inherited attributes, on the other hand, pass information down in a parse tree.

- Synthesized attributes of terminals (leaves in the parse tree) are also called **intrinsic attributes**. Intrinsic attributes are usually filled in by lexical analyzers.

- As can be seen from the example, a symbol may have more than one attribute. For example $A(\texttt{tNUM}) = \{\texttt{value}, \texttt{negative}\}$.

- Also the same attribute (name) may be associated with more than one symbol. For example we have `result` as an attribute for both `<expr>` and `<num>`.

- Associated with each production is a set of semantic functions over the set of attributes of the symbols that appear in the production. The semantic functions can be considered as code fragments that will be used to calculate the values of the attributes.

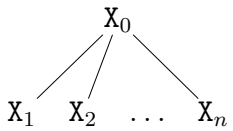
- That is, for a rule of the form

$$\texttt{X}_0 \texttt{ -> } \texttt{X}_1 \ \texttt{X}_2 \ \ldots \ \texttt{X}_n$$

synthesized attributes of $\texttt{X}_0$ will be computed by functions on the set of attributes

$$A(\texttt{X}_1) \cup A(\texttt{X}_2) \cup \cdots \cup A(\texttt{X}_n)$$

On a parse tree,



Synthesized attributes of $\texttt{X}_0$ will depend (calculated based on) the values of attributes (both synthesized and inherited) of its children $\texttt{X}_1 \ \texttt{X}_2 \ \ldots \ \texttt{X}_n$.

- **Synthesized attributes of a symbol depend on the attributes of its children**

- Let us go back to our example grammar. Semantic functions associated with the grammar rules can be given as follows:

Production	Semantic function
`<expr> -> <num> + <num>`	`<expr>.result = <num`$_1$`>.result + <num`$_2$`>.result`
`<num> -> tNUM`	`<num>.result =` `(tNUM.negative)? -1 * tNUM.value : tNUM.value`

• Note that all the attributes in this example are synthesized attributes. The value of the `result` attribute of an `<expr>` node, will be calculated based on the values of `result` attributes of `<num>` nodes as can be seen from the semantic action of the production on the first line above. Since such `<num>` nodes will be children of the `<expr>` node in a parse tree, `result` is a synthesized attribute of `<expr>` nodes.

• Similarly, the value of the `result` attribute of a `<num>` node, will be calculated based on the values of the `result` and `negative` attributes of the `tNUM` node. Such a `tNUM` node will be a child of the `<num>` node in a parse tree, and hence `result` is also a synthesized attribute of a `<num>` node.

• Note that, within the production

`<expr> -> <num> + <num>`

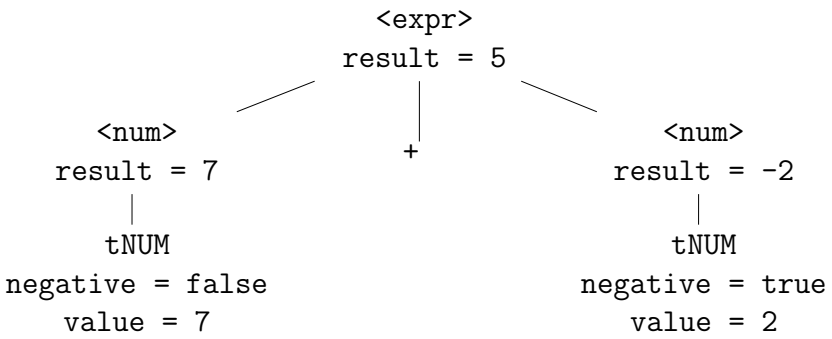
there are two occurrences of the symbol `<num>`. They will be corresponding to different nodes in a parse tree, hence the values of the attributes for them may be different. In order to be able to refer to the attribute of the required occurrence and to distinguish between them, we use subscripts within the semantic functions.

In the semantic function

$$\texttt{<expr>.result = <num}_1\texttt{>.result + <num}_2\texttt{>.result}$$

$\texttt{<num}_1\texttt{>.result}$ is the `result` attribute of the first `<num>` symbol on the right hand side of the corresponding production, and $\texttt{<num}_2\texttt{>.result}$ is the `result` attribute of the second `<num>` symbol on the right hand side of the corresponding production.

• An *attributed tree* is a parse tree where the attributes of the symbols are also labeling the nodes, and their values are set.



• An inherited attribute of a node, on the other hand, depends on the attributes of its parent and its siblings

• That is again for a rule of the form

$$\texttt{X}_0\ \texttt{->}\ \texttt{X}_1\ \texttt{X}_2\ \ldots\ \texttt{X}_{j-1}\ \texttt{X}_j\ \texttt{X}_{j+1}\ \ldots\ \texttt{X}_n$$

synthesized attributes of $\texttt{X}_j$, where $1 \leq j \leq n$, will be computed by functions over the set of attributes

$$A(\texttt{X}_0) \cup A(\texttt{X}_1) \cup A(\texttt{X}_2) \cup \cdots \cup A(\texttt{X}_n)$$

- Consider the grammar

```
<ident_list> ->  tIDENT, <rest>
      <rest> ->  tIDENT, <rest>
               |  tIDENT
```

This grammar produces (recognizes) a list of `tIDENT` tokens.

Assume that for each `tIDENT` we want to know its index number. For this purpose we can use an inherited attribute with name say, `order`, associated with each `tIDENT` symbol.

`order` of `tIDENT` on the RHS of the first production is easy to set, it's got to be 1. But, what about the `order` of `tIDENT`s on RHSs of the other productions. In order assign correct index numbers to them, we need to know how many `tIDENT`s seen before. This information can be passed by using another attribute, say `last`, associated with `<rest>` which will be assigned to the number of `tIDENT`s seen so far.

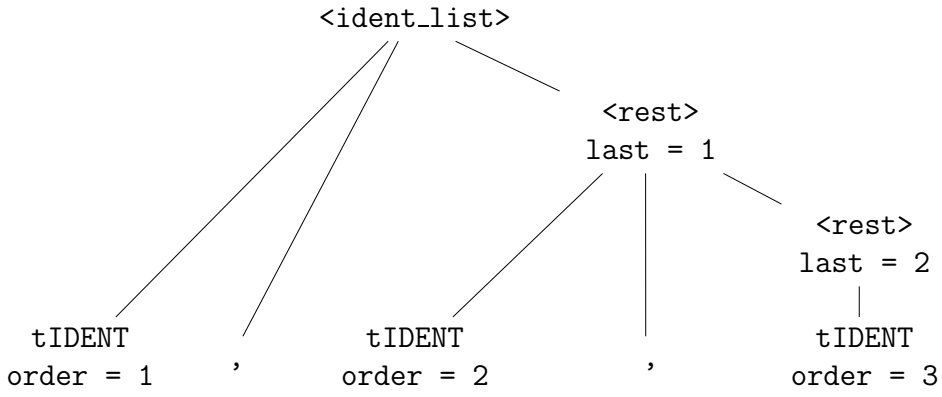
The semantic actions for such a purpose will be:

Production	Semantic rule
`<ident_list> -> tIDENT, <rest>`	`tIDENT.order = 1` `<rest>.last = 1`
`<rest> -> tIDENT, <rest`$_1$`>`	`tIDENT.order= <rest>.last+1` `<rest`$_1$`>.last = tIDENT.order`
`<rest> -> tIDENT`	`tIDENT.order= <rest>.last+1`

- Note that, both `order` and `last` are inherited attributes.

- Also note that, `<ident_list>` has no attribute. So we do not need to have attribute associated with all the symbols.

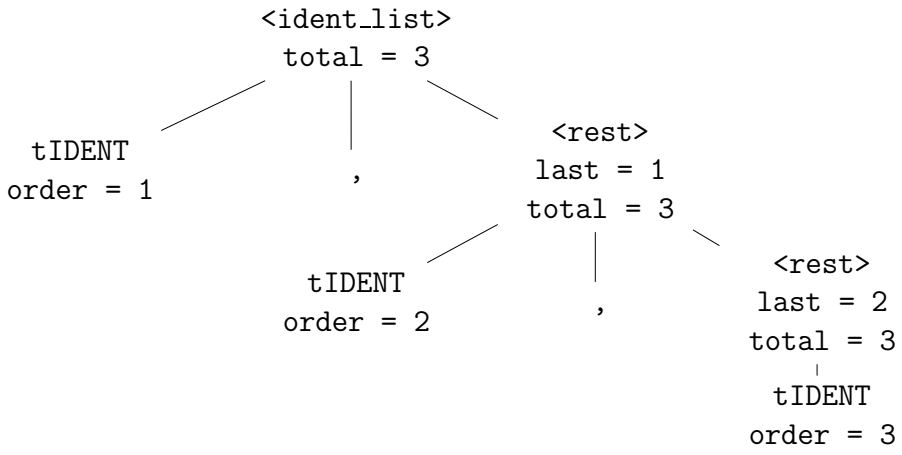
- An example attributed tree with this attribute grammar would be



- Assume that we want to associate the total number of `tIDENT`s with `<ident_list>` at the end. This information is already available as the value of the `order` attribute of the deepest `tIDENT`. All we have to do is to pass it to the root of the parse tree, which can be achieved by using a synthesized attribute, say `total`, associated with `<rest>` and `ident_list`.

Production	Semantic rule
<ident_list> -> tIDENT, <rest>	tIDENT.order = 1 <rest>.last = 1 <ident_list>.total = <rest>.total
<rest> -> tIDENT, <rest>	tIDENT.order= <rest>.last+1 <rest$_1$>.last = tIDENT.order <rest>.total = <rest$_1$>.total
<rest> -> tIDENT	tIDENT.order= <rest>.last+1 <rest>.total = tIDENT.order

- Which would produce the following attributed tree



- Note that `total` is a synthesized attribute of `<rest>` and `<ident_list>` nodes. Hence inherited and synthesized attributes can be used together.

- Inherited attributes are used to identify context sensitive information.

- A predicate function has the form of a boolean expression on the attribute set

$$A(\mathtt{X}_0) \cup A(\mathtt{X}_1) \cup A(\mathtt{X}_2) \cup \cdots \cup A(\mathtt{X}_n)$$

- As an example, consider the following grammar and rule for Ada procedures

Production	Predicate
<proc_def> -> **procedure** tNAME <body> **end** tNAME	{ tNAME$_1$.str == tNAME$_2$.str }

assuming that lexical analyzer fills in the intrinsic attribute `str` of `tNAME` tokens with the actual lexeme of `tNAME` tokens.

- For a derivation with attribute grammar to be successful, all the predicate functions must evaluate to true

- Although it is not the case in practice, it is useful to think that, first the parse tree is produced with empty attribute values, and then the attribute values are computed using the semantic actions

- If all the attributes are inherited, then attributes can be computed by visiting the nodes in the parse tree in a top–down manner

- If all the attributes are synthesized, then attributes can be computed by visiting the nodes in the parse tree in a bottom–up manner

- However, if there are both inherited and synthesized attributes, then a care must be taken for the order of the evaluation of the attributes in order not to get into a circular dependency.

2.8 Attribute Grammars in Bison

- Let us consider the following example

```
<expr> ->  tNUM
         |  <expr> <expr> +
         |  <expr> <expr> -
```

- Suppose that we want to calculate the value of the input expression.

- We know we can do this by using attributes. We can attach a single attribute named, say, `val` to all the symbols. Then the following attribute grammar will calculate the value of the input expression, and store the final result in the `val` attribute of the `<expr>` node at the root of the parse tree.

Production	Semantic function
<expr> -> tNUM	<expr>.val = tNUM.val
<expr> -> <expr> <expr> +	<expr>.val = $<expr_1>$.val + $<expr_2>$.val
<expr> -> <expr> <expr> -	<expr>.val = $<expr_1>$.val - $<expr_2>$.val

- If all the symbols has a single attribute of the same type, then it is quite easy to implement such an attribute grammar using bison.

- In the example above, all the symbols has a single attribute `val` of type integer.

- Let us see the bison file and explain the required modifications on it (`ex3.y`)

ex3.y

```
%{
#define YYSTYPE int
#include <stdio.h>

void yyerror (const char *s) { /* Called by yyparse on error */
  printf ("%s\n", s);
}
%}

%token NUM EOE
%start full_expr

%% /* Grammar rules and actions follow */
expr:     NUM           { $$ = $1; }
        | expr expr '+' { $$ = $1 + $2; }
        | expr expr '-' { $$ = $1 - $2; }
;

full_expr : expr EOE /* End Of Expression */ { printf("Value=%d\n", $1); };

%%
int main () {
  return yyparse ();
}
```

- If we `#define` the built-in macro `YYSTYPE` to a certain type (`int` in this example), then it means there will be only one attribute associated with all the symbols, and the type of this attribute will be type to which `YYSTYPE` is defined to.

- Let us start from the grammar part first.

- We have the same productions for `expr` as before, however we added the semantic actions to these productions.

- The symbols `$$` and `$n` (where `n` is a number) are used to refer to the attributes of the symbols that appear in the production.

- `$$` is the attribute of the symbol that appears on the LHS of the production

- `$n` is the attribute of the $\texttt{n}^{th}$ symbol (counted from left to write including all the tokens) that appear in the production.

- For example, when we consider

```
expr:  expr expr + { $$ = $1 + $2; }
```

`$$` is the attribute of the `expr` on the LHS, `$1` is the attribute of the first `expr` appear on the RHS, and `$2` is the attribute of the second `expr` that appear on the RHS.

- It is clear that, with the given semantic actions, attribute of the root node in the parse tree will have the value of entire expression.

- Consider the parsing of `3 5 + 2 -`, and give the attributed parse tree

- However, assume that we want to print out this calculated value. In order to do that, we need to write in a semantic function something like `printf("Value=%d\n",$$)`

- Assume that we add this print statement into the semantic actions of the productions of `expr`. However, in this case, the value will be printed out for each subexpression parsed.

- We do not want that, instead we want to print out the value of the final expression.

- In order to do that, we need to understand that the parsing of an expression is finished.

- Note that seeing `3 5 + 2 -` in the input (a valid expression) does not necessarily mean that the expression is completed. The user may continue to write an expand the expression as `3 5 + 2 - 4 5 - +`, or in some similar way.

- Therefore we need to have an explicit mark in our syntax that the user can use to mark the "End Of an Expression (EOE)" (note EOE is just a token name we made up).

- For this reason, we introduce a new token `EOE` and a new nonterminal `full_expr` in our grammar.

```
%token NUM EOE
```

and

```
full_expr:  expr EOE
```

- When a derivation is performed by using the production of `full_expr`, we understand the expression has ended. At this point, we can print out the value of the expression.

```
full_expr:  expr EOE { printf("Value=%d",$1); }
```

Note that, the value of the entire expression is kept in the attribute of the `expr` that appear as the first symbol on the RHS of this production, hence we are using `$1` to access this value.

- Note that, we did not state or described what actual text in the input will correspond to EOE yet. We will give this in the description of the scanner.

- Another new feature we see in this bison file is the use of the `%start` keyword. Normally, the start symbol of the grammar is taken to be the symbol on the LHS of the first production in the

grammar part. If we want to specify the start symbol explicitly, then this can be done by using the `%start` statement, as we did in this example.

• The only thing that remains unexplained is the following: How does the token `NUM` gets its attribute value assigned?

• Recall that, the attribute of `NUM` is an intrinsic attribute, and as we have mentioned before, the intrinsic attributes of the tokens are usually assigned by the scanner. Hence this question will be answered when we examine the scanner that will be used in this example.

• Here is the scanner, which looks like a lot to the previous scanner with slight modifications

ex3.flx

```
%{
/* get the token definitions from bison output file */
#include "ex3.tab.h"
%}
%%
[0-9]+    { /* caught a number */
            yylval = atoi (yytext);
            return NUM;
          }
\n\n      return EOE; /* saw end of expression pattern */
[ \t\n]+  /* eat up white space */
.         { /* pass any other character to the parser */
            return yytext[0];
          }
%%
```

• The first modification noticed is the assignment for the intrinsic attribute of `NUM` tokens. Whenever a number is seen in the input text, its value is assigned to the predefined variable `yylval` which corresponds to the attribute of the tokens.

• The other modification is the rule for `EOE`. So we assume here that the user will enter an empty line (without any spaces or tabs on the line) to mark the end of the expression. Hence seeing two consecutive newline characters will produce an `EOE` token.

• Note that, `EOE` does not have an attribute, hence we do not have an assignment to `yylval` in the action of the rule given for `EOE`.

2.9 Multiple Attributes

- We have seen how to make use of attributes if all the symbols are assumed to have a single attribute of the same type
- However, this is not realistic for most of the language translators
- Different symbols may need to have different typed attributes, and a symbol may need to have more than one attribute
- Let us consider the following exercise to give an example
- Postfix expressions consisting of positive integer numbers and addition and subtraction
- As the output, we will print out something similar to the parse tree
- First a common header file named `ex4.h` that will contain the definitions that will be used by both the scanner and the parser

ex4.h

```
#ifndef __EX4_H
#define __EX4_H

typedef enum { EXPR3, EXPR1 } NodeType;

typedef enum { ADD, SUB } OpType;

typedef struct Expr3Node {
  OpType operator;
  struct TreeNode *leftChild;
  struct TreeNode *rightChild;
  } Expr3Node;

typedef struct Expr1Node {
  int Value;
  } Expr1Node;

typedef union {
  Expr1Node expr1;
  Expr3Node expr3;
  } ExprNode;

typedef struct TreeNode  {
  NodeType thisNodeType;
  ExprNode *exprNodePtr;
  } TreeNode;

#endif
```

- A node will be created for each symbol in the parse tree
- However, there are two different type of `expr` nodes in the parse tree (those with 3 children, and

those with a single child)

- In order to differentiate between these two, let us declare an enum type named `NodeType`
- For `expr` nodes with 3 children, it may correspond either to an addition or to a subtraction
- Let us declare another enum type `OpType`
- Now the general tree node can be declared as in the definition of the `struct TreeNode`.
- It is reasonable to use an instance of this struct for `expr` nodes of 3 children
- However, we will also use it for `expr` nodes with a single child
- In this case, we will store the value of the integer seen in the input in the `leftChild` field by interpreting this pointer as an integer.
- Now the bison file `ex4.y`

```
%{
#include <stdio.h>
#include "ex4.h"

void yyerror (const char *s)  /* Called by yyparse on error */
{
  printf ("%s\n", s);
}

TreeNode * mkEXPR3 ( TreeNode *, TreeNode *, OpType);
TreeNode * mkEXPR1 ( int );
void printTree ( TreeNode *, int);

TreeNode * rootPtr;
%}

%union {
  int value;
  TreeNode *treePtr;
}

%token <value> NUM
%token EOE
%type  <treePtr> expr

%start full_expr
```

```
%% /* Grammar rules and actions follow */


expr:     NUM           { $$ = mkEXPR1 ( $1 ); }
        | expr expr '+' { $$ = mkEXPR3 ( $1, $2, ADD ); }
        | expr expr '-' { $$ = mkEXPR3 ( $1, $2, SUB ); }
;

full_expr : expr EOE { rootPtr = $1; };
%%

void printTree ( TreeNode * t, int indent) {
   int i;
   for (i=1; i <= indent; i++) printf(" ");

   if (t->thisNodeType==EXPR1) {
      printf ("EXPR(%d)\n", t->exprNodePtr->expr1.Value);
      return;
      }

   printf("EXPR\n");
   printTree(t->exprNodePtr->expr3.leftChild, indent+1);
   for (i=1; i <= indent+1; i++) printf(" ");
   if (t->exprNodePtr->expr3.operator==ADD)
      printf("+\n");
   else
      printf("-\n");
   printTree(t->exprNodePtr->expr3.rightChild, indent+1);
}

TreeNode * mkEXPR3 ( TreeNode *left, TreeNode *right, OpType operator) {
    TreeNode * ret    = (TreeNode *)malloc (sizeof(TreeNode));
    ret->thisNodeType = EXPR3;
    ret->exprNodePtr = (ExprNode *)malloc (sizeof(ExprNode));
    ret->exprNodePtr->expr3.operator = operator;
    ret->exprNodePtr->expr3.leftChild = left;
    ret->exprNodePtr->expr3.rightChild = right;
    return (ret);
    }

TreeNode * mkEXPR1 ( int value ) {
    TreeNode * ret    = (TreeNode *)malloc (sizeof(TreeNode));
    ret->thisNodeType = EXPR1;
    ret->exprNodePtr = (ExprNode *)malloc (sizeof(ExprNode));
    ret->exprNodePtr->expr1.Value  = value;
    return (ret);
```

```
    }

int main ()
{
  if (yyparse())
     printf ("Could not parse!!!\n");
  else
     printTree ( rootPtr, 0 );
}
```

- Now the flex file `ex4.flx`

```
%{
/* get the token definitions from bison output file */
#include "ex4.h"
#include "ex4.tab.h"

%}
%%
[0-9]+    { /* caught a number */
            yylval.value = atoi (yytext);
            return NUM;
          }
\n\n      return EOE; /* saw end of expression pattern */
[ \t\n]+  /* eat up white space */
.         { /* pass any other character to the parser */
            return yytext[0];
          }
%%
```

Chapter 3

Names, binding, type checking and scopes

- Names are used for all kinds of entities in programming languages : variables, functions, types
- Design issues related to names are therefore important
- A name is a string of characters used to identify an entity in a program
- Early languages developed for purely scientific computation purposes allowed single character names only (this was reasonable since usually single symbol names are used for unknowns in a mathematical formula)
- FORTRAN I removed this limit and allowed up to 6 characters in a name
- FORTRAN 90 and C originally allowed 31 chars
- Some languages like Ada and Java do not put a limit
- In some languages, the names are case sensitive (e.g. `myvar` and `myVar` denote different variables)
- Case sensitivity is not good for readability: similar looking constructs must have the same meaning
- For this reason, in case sensitive languages, people developed coding standards (like variables start with a small letter, first letter of every word is capital)

3.1 Variables

- Imperative languages are abstractions of the underlying von Neumann computer architecture.

- Variables are abstractions for memory cells, and the operators on them are abstractions of the operators provided by the processor.

- Sometimes this abstraction is very close to the actual representation.

- E.g. an integer is directly represented by a single machine word, addition is directly provided by the CPU.

- However, sometimes the abstraction is quite distinct from the actual representation.

- E.g. a matrix (2D array) is represented in a complex way in the memory, and software support is required to maintain its structure. Furthermore, an operation like matrix multiplication is not provided by a usual CPU, and software again is used to provide such an operation.

- A variable can be characterized by a collection of properties like : name, address, value, type, lifetime, scope

3.1.1 Address

- **Address:** The address of a variable is the memory address with which the variable is associated.

- This association is not as simple as it first appears.

- It is possible for the same name to be associated with different addresses at different places, at different times in the same program.

- For example, two subprograms `sub1` and `sub2` each declaring a variable named `sum`.

- Since `sum` in `sub1` and `sum` in `sub2` are actually different variables, they must be associated with different memory addresses.

- Also a variable (like `result` below) in a recursive function

```
int fact (int x) {
  int result;
  if (x = 2)
     result = 2;
  else
     result = x * fact(x-1);
  return result;
  }
```

will need to be associated with different addresses at each recursion of the function.

• The address of a variable is sometimes called the **l–value** since the address of the variable is required when the variable appears on the LHS of an assignment.

3.1.2 Aliases

• It is possible to have multiple identifiers referencing to the same memory address.

• For example, through pointers, reference variables, certain type of parameters in procedure calls can be used to create aliasing identifiers.

• Aliasing, since it allows to use different names to refer to the same address, may cause hard to find errors.

3.1.3 Type

• Type of a variable determines the set of values that the variable can be assigned.

• It also determines the set of operations to which the variable can be subject to.

3.1.4 Value

• The value of a variable is the content of the memory cell (or cells) associated with the variable.

• Note that, a single physical cell may not be enough to store the value of a variable. In this case, the value of the variable will actually be stored in consecutive physical cells.

• However, it is convenient to think that the variable is associated with a single logical memory cell, which corresponds to multiple physical cells large enough to store the value.

• The value of a variable is sometimes called the **r–value** since the value of the variable is required

when the variable appears on the RHS of an assignment.

• To access the r–value (the value of a variable), l–value must be determined first. For example, considering an assignment statement of the form "`x = y;`", in order to access to the value of the variable `y` on the RHS of this statement, we should first know the l–value (the address) for `y`. Then this address in the memory has to be accessed to read the current value for `y`.

3.2 Binding

- In general sense, binding is an association.
- The time at which binding takes place is called the **binding time**.
- Consider the following code fragment which displays possible different binding times.

```
int myVar;
...
myVar = abs(myVar * -2);
```

- Bindings can take place at
  - language design time: the character `*` is bound to multiplication operator
  - compiler design time: set of possible integer values are bound to -32.768 to 32.767 by deciding that 2 bytes will be used for an integer
  - compile time: type of `myVar` is bound to the type `int`
  - link time: `abs` is bound to the library function `abs` at link time
  - load time: `myVar` is bound to an address at load time if it is a static variable
  - run time: `myVar` is bound to an address at load time if it is not a static variable
- A binding is **static** if it occurs before the run time.
- If a binding occur at run time, or can be changed at run time, then it is called **dynamic binding**.

3.2.1 Type binding

- A variable must be bound to a type.

- Variables can be bound to a type by explicit declarations (as in C), or by implicit declarations.

- As an example for implicit type binding: FORTRAN implicitly declares variables with names starting with `I, J, K, L, M, N` to have integer types, and all the others to have real types.

- Type binding through declarations are static.

- Some languages provide dynamic type binding (e.g. APL, JavaScript, PHP).

- The type of a variable can be changed during run time and the type of a variable is decided based on the type of the expression being assigned to that variable.

- E.g. in JavaScript

```
list = [ 1.2, 3.5 ]
...
list = 33
```

  `list` is first an array of numbers, then it becomes a scalar variable.

- Advantage of dynamic type binding: It is easy to write a program that will be applied to a variety of types (e.g. count the number of elements in a list of ...)

- Disadvantage: It becomes harder to detect the errors.

- The same disadvantage appears in languages that allow coersion (implicit type conversion).

- Another disadvantage of dynamic type binding is the runtime cost of changing (logical) memory cell to which the variable is bound. Since the type changes, the variable may need a different number of (possibly more) physical memory cells to represent the values of its new type. Therefore, the old address binding of the variable may need to be removed, and a new address binding may be needed.

- Therefore, dynamically typed languages are usually used in interpreted languages (buries the time into the interpretation overhead, hence the additional cost of dynamic type binding is not apparent).

3.2.2 Storage binding

- The memory cell to which a variable is bound is somehow taken from a memory pool, which is known as the **allocation**.

- **Deallocation** is placing the memory cell allocated for a variable back in the pool of available memory.

- The **lifetime** of a variable is the time during which the variable is bound to a specific memory location.

- There are four types of storage binding for variables.

- **1) Static variables:** They are bound to a memory location before the program execution begins, and remain bound to that location until the program terminates.

- E.g. variables requiring a global access (i.e. the global variables).

- Variables declared with the keyword `static` in C.

- Since their address is constant during the execution of the program, they can be accessed directly (whereas the variables whose address change requires an indirect method of access which is slower).

- A disadvantage is the following: If a PL allows only static variables, then it cannot support recursive procedures (e.g. FORTRAN I, II, IV).

- **2) Stack–dynamic variables:** These are the variables that are bound to a storage when the program reaches to the part where they are declared.

- For example the variables declared in a subprogram (except those declared to be `static`) are stack dynamic variables.

- A stack is managed by the runtime environment, and these variables are bound to memory locations on this stack.

- Every time a new subprogram is called, the local variables declared within the subprogram are bound to memory locations on the stack.

- The bindings for these local variables are removed when the subprogram returns.

- In Java, C, C++, the variables of functions are by default stack–dynamic.

- **3) Explicit heap dynamic variables:** Memory cells that are allocated by explicit instructions

specified by the user.

- These type of variables can only be accessed through pointer or reference variables, and do not have names.

- Some languages provide operators (`new` in C++), some languages provide library functions (`malloc` in C) for creating such variables.

- These variables are bound to a memory location in the part of the memory which is called the **heap**.

- Due to unpredictable allocation and deallocation, the heap can become highly disorganized.

- Although the storage binding for these variables are dynamic, the type binding can be static.

- For example:

```
int * intptr;
...
intptr = new int;
...
delete intptr;
```

The variable created by `new int` has type `int` and this binding is done at compile time. Only the binding to a memory cell is performed at run time.

- Some languages provide statements to deallocate explicit heap dynamic variables (`delete` in C++, `free` in C).

- However some languages, like Java, do not have such statements, and rely on a **garbage collector** to deallocate the explicit heap dynamic variables that are not used any more.

- These variables are usually used to implement dynamic structures like linked lists, trees, etc.

- **4) Implicit heap dynamic variables:** Similar to explicit but without an explicit allocation statement.

- If an assignment requires allocation of more memory, this is understood at run time, and the required binding is performed.

3.3 Type Checking

• Note that, the subprograms can be seen as operators whose operands are the parameters of the subprogram.

• An assignment can be considered as a binary operator with target variable and expression being the operands.

• **Type checking** is the activity of ensuring that the formal and the actual operands of an operator are of compatible types.

• A **compatible** type is one that is either legal for the operator, or is allowed under language rules to be implicitly converted to a legal type (this implicit and automatic conversion is known as **coersion**).

• A **type error** is the application of an operator to an operand of an inappropriate type.

• If the type binding is static, then type checking can be performed almost statically as well. However, there may still be some checks that need to be performed dynamically (e.g. array out of bound checks).

• However, dynamic type binding requires dynamic type checking.

3.4 Strong typing

• A language is said to be **strongly typed** if all the type errors can be detected (type checking can be dynamic and /or static).

• For example, C and C++ are not strongly typed (union's are not type checked).

• On the other hand both Ada and Java are nearly strongly typed: explicit casting of types are required.

• Coersion rules decreases the possibility of detecting types errors, although they provide an easier way to write the programs.

• Java has half as many assignment type coercions as C++, Ada has much less (so the user must explicitly cast the types denoting that the type incompatibility is intentional).

3.5 Type compatibility

- Type compatibility is important for type checking.

- A variable can be assigned to an expression without a type error if the type of the variable and the type of the expression are compatible.

- There are two different type compatibility methods: **name type compatibility** and **structural type compatibility**

- Structural type compatibility is flexible but difficult to implement.

- Rule 1: A type name is compatible with itself.

- Rule 2: A type can be declared to be compatible with another type (e.g. by using a type declaration of the form "`type` T_1 `=` T_2" or "`typedef T1 T2;`").

- Rule 3: Two types are structurally compatible if they are formed by applying the same type constructors to compatible types.

- For example the types `S` and `T` given below are compatible according to structural compatibility:

```
type mychar = char;
type S : array [0..9] of char;
type T : array [0..9] of mychar;
```

- Name compatibility is easy to implement but quite restrictive.

- There are certain flavors of name equivalence.

- **Pure name equivalence:** A type is only compatible to itself and constructed types are not compatible with any other constructed types.
  For example:

```
type S : array [0..9] of integer;
type T : array [0..9] of integer;
type U : S;
type V : U;
```

  In this example, if pure name equivalence is used, then `S`, `T`, `U` and `V` are **not** compatible with each other.

- **Transitive name equivalence:** A type can be declared as compatible with another type.

In the example given above, `S`, `U` and `V` are compatible with each other, however `T` is not compatible with any type.

- C and C++ use structural compatibility for all types except `struct`s (this approach helps compilers to avoid circular type comparisons).

If we allow structural compatibility for `struct`s as well, let us consider what will be needed to check the type in the following example:

```
struct MyList {
  int data;
  MyList * next;
  }
////
struct YourList {
  int data;
  YourList * next;
  }
////
MyList list1;
YourList list2;
////
list1 = list2; // problematic
```

The last assignment statement in this example is type safe if the LHS and RHS of the assignment are compatible types. Are `MyList` and `YourList` compatible types? They are both `struct`s, hence the same type constructor is used. However, we need to make sure that the type constructor is applied to compatible types as well. Both `MyList` and `YourList` have two fields, and one of these fields has `int` type. The other field of `MyList` and `YourList` are pointers to `MyList` and `YourList`, respectively. Hence, in order to decide `MyList` and `YourList` are compatible or not, we need to know if `MyList` and `YourList` are compatible.

In order to avoid this cyclic dependency that may easily emerge when structural compatability is used, C and C++ opt to use name equivalence for `struct` types.

3.6 Scope

- A variable is **visible** in a statement if it can be referenced in that statement.

- For example:

```
int x,y;

void foo1 (int r) {
  x = r;
  }

void foo2 (int s) {
  y = s;
  }
```

`x` is visible in both the assignment of `foo1` and `foo2`. Similarly, `y` is visible in both the assignment of `foo1` and `foo2`. However, `r` is not visible in the assignment of `foo2` and `s` is not visible in the assignment of `foo1`.

- The **scope** of a variable is the range of statements in which the variable is visible.
- `x` and `y` has the entire program as their scope.
- The scope of `r` is the body of the function `foo1`.
- A variable is **local** to a unit (e.g. subprogram, function) or block if it is declared within that unit or block.
- A **non–local** variable in a unit or block is a variable that is visible in the unit or block, but not a local variable of the unit or block.
- Scoping rules of a language determine how a particular occurrence of a name is bound to a variable.
- In particular, scoping rules determine how references to non–local variables are associated with their declarations.
- There are two different methods used for scoping: static scoping and dynamic scoping.

3.6.1 Static scoping

- This is the scoping we are used to.
- The scope of a variable can be determined statically at compile time.
- The binding of a non–local variable to its corresponding declaration is performed at compile

time.

- Scopes are associated with program units or blocks.

- The units, and hence scopes can be nested.

- For example, in Pascal the procedures can be nested:

```
procedure big;
  var x : integer;
  procedure sub1;
    begin { of sub1 }
    ... x ...
    ... y ...
    end; { of sub1 }
  procedure sub2;
    var x : integer;
    var y : integer;
    begin { of sub2 }
    ... x ...
    end; {end of sub2 }
  begin { of big }
  ... x ...
  end; { of big }
```

- When a variable reference is seen, first the unit in which the variable reference is seen is searched for a declaration of that variable.

- If a declaration can be found within the same unit, then the variable reference is bound to that declaration. In this case, the variable reference is local.

- For example, both in `procedure big` and in `procedure sub2`, the variable references to `x` within the bodies of these procedures are local references, since there is a declaration for `x` within these procedures.

- When a declaration for the referenced variable cannot be found within the same scope, then the enclosing scope of it is searched for a declaration.

- The enclosing scope is called the **static parent**.

- For example, `procedure big` is the static parent of the `procedure sub1`.

- When the reference to `x` in `sub1` is considered, we see that there is no declaration for `x` within

`sub1`, hence it is not a local reference.

• If we are using static scoping, then we goto the static parent of `sub1` and search for a declaration of `x` there, and find the declaration for `x` in the procedure `big`, and bind the reference to `x` in `sub1` to the declaration of `x` in `big`.

• If a declaration cannot be found in the static parent either, then we goto the static parent of the static parent (which is the next enclosing unit), and look for a declaration in that unit.

• This process continues up to the largest unit. If a declaration could not be found in the largest unit, then it corresponds to an undeclared variable usage.

• For example, the reference to `y` in `sub1` is an undeclared variable usage. Although there is a declaration of `y` in `sub2`, this declaration will not be found in our search, as we will always keep going up in the unit enclosure hierarchy.

• Note that, the same variable can be declared in a unit and in another unit which is a static ancestor. For example consider `x` declarations in `big` and `sub2`.

• In such a case, the declaration in the ancestor `foo1` is **shadowed** by the declaration in the descendant `foo2`. In any descendant unit of `foo2`, the declaration in `foo1` becomes invisible.

• Some programming languages provide scope resolution operators to access such shadowed declarations: `big.x` or `big::x`

• Some languages allows blocks to be declared in the middle of a code fragment.

• For example:

```
...
{
  int x;
  x = y;
  y = z;
  z = x;
}
...
```

• Such blocks are common in **block-structured languages** (procedures are blocks with names, so they can be entered from any part of the code, however these blocks are nameless, hence they can only be entered where they are declared).

• Blocks introduce their own scope as any other named blocks (procedures).

• The variables declared in a block are also stack dynamic (just as the variables declared in

procedures).

- The same scoping rules apply to nameless blocks.

- In some languages, there are some constructs that are considered as blocks (e.g. `for`, `while` in C). These constructs again introduces their own scope.

- One problem with static scoping is that, it does not reflect the program design structure very well.

- Consider the following example:
  - The main program will make use of two subprograms `A` and `B`, but `A` and `B` are not required to call each other.
  - Similarly, `A` will make use of two subprograms `C` and `D`, but `C` and `D` are not required to call each other.
  - `B` will make use of a subprogram `E`.
  - The procedure calling needs given above are the only needs.

- Normally, in a statically scoped language, one would form the following skeleton structure

```
program main;
  procedure A;
    procedure C;
    end C;
    procedure D;
    end D;
  end A;
  procedure B;
    procedure E;
    end E;
  end B;
end main;
```

- This skeleton is good as it hides `C` and `D` from `main`, `B` and `E` which are not required to call `C` or `D`.

- However, we do not need `A` calling `B`, or `D` calling `A`.

- There is no way to avoid these unwanted visibilities, hence a program which includes these unwanted calls will be allowed without any error detection.

3.6.2 Dynamic scoping

- In dynamic scoping, the binding for a nonlocal reference is decided at run time.

- When a procedure `foo1` calls a procedure `foo2`, `foo1` is called **the dynamic parent** of `foo2`.

- The dynamic scoping works in the same way as the static scoping, however instead of static parents, dynamic parents are used.

- Consider the example:

```
procedure big;
  var x : integer;
  procedure sub1;
    ... { suppose there is no declaration for x }
    begin { of sub1 }
    ... x ...
    end; { of sub1 }
  procedure sub2;
    var x : integer;
    begin { of sub2 }
    ...
    end; {end of sub2 }
  procedure sub3;
    ... { suppose there is no declaration for x }
  end; { end of sub3 }
  begin { of big }
  ...
  end; { of big }
```

- `x` in `sub1` is a non-local reference. Hence, the declaration it will be bound to, will be in another procedure.

- Note that, `sub1` can be called by `big` and `sub2`, and both of these procedures include a declaration for `x`.

- The declaration to bind this nonlocal reference of `x` in `sub1` depends on the calling sequence:
`big -> sub1` : bound to declaration in `big`
`big -> sub2 -> sub1` : bound to declaration in `sub2`
`big -> sub2 -> sub3 -> sub1` : bound to declaration in `sub2`

- Dynamic scoping is not without any problems.

- As the actual variable that will be bound to a non-local reference can only be known at run time, the programs become unreadable easily.

- Also, the run time activity required to resolve a non-local reference slows down execution of the programs.

- However, as the variables of the caller procedure are immediately visible to the called procedure, there is usually no need to pass parameters in a procedure call.

- The disadvantages given above are considered to be more important, and the dynamic scoping is not as widely used as static scoping.

3.6.3 Static vs Dynamic Scoping Example

```
var x = 0; // define x and initialize it to 0

function foo1 ()
{
   print x;
   x = 1;
}

function foo2 ()
{
  var x = 2; // define x and initialize it to 2
  foo1();
}

foo2();
print x;
```

If static scoping is used, then this program will output: 0 1
If dynamic scoping is used, then this program will output: 2 0

3.7 Named Constants

- A **named constant** is a variable that is bound to a value only when it is declared.

- Named constants are sometimes called **manifest constants** as well.

- Usually, named constants are used to increase the readability (e.g. by using the name `pi` to denote the value `3.14159`).

- Another typical usage is to increase manageability (e.g. consider you are expected to input 100 integer values, and compute the mean of them. It would be wise to use a named constant set to 100, since the program can be changed to handle any other number of inputs simply by changing the value of this named constant.)

Chapter 4

Data Types

4.1 Introduction

- Since the programs produce their results by manipulating data, the design and implementation of data types are quite important for programming languages.

- In early languages, there were only a few built in data types and programmers were not allowed to define new types.

- Languages introduced features to let programmers specify the accuracy of data types and introduced new data type support.

- But, it was ALGOL68 which first introduced the features that let a programmer to define a new data type.

4.2 Primitive Data Types

- If a data type is not defined in terms of other data types, then it is called a **primitive data type**.

- Almost all programming languages provide a set of primitive data types.

- Some of these types are just reflections of the hardware (e.g. integer types).

- Some require just a little software support.

4.2.1 Numeric types

- Many early languages had only support for numeric primitive types.
- These types still play an important role in contemporary languages.

Integer

- The most common primitive type.
- Languages nowadays provide different sizes of integer types (e.g. short, long).
- Both signed and unsigned (positive only) integers are supported.
- Unsigned integers are represented in the hardware as a sequence of bits. For example, if we assume that 1 byte (=8 bits) is used represent an integer, then the integer number 5 would be stored as "`00000101`". This is also called the binary representation of the numbers.
- We will assume below that 1 byte is used to represent integers in the examples of this section. However, the actual size (the number of bytes) to represent an integer depends on the actual implementation of the programming language.
- For signed integers, there are two different representation techniques: Sign magnitude and complements.
- In sign magnitude, the first bit in the binary representation denotes the sign of the integer number (having 1 as the first bit denotes that the number is negative, and 0 as the first bit denotes that the number is positive), and the rest of the bits give the absolute value of the number. For example, "`00000101`" is 5, but "`10000101`" is -5.
- The complements approach is divided into two as 1s–complement and 2s–complement.
- Both 1s–complement and 2s–complement rely on the complementation of the bits for the negative integers. Note that complement of 1 is 0, and complement of 0 is 1.
- Ones complement works in the following way. The first bit of the number will again give the sign of the number. However, if the number is negative, then we take the complements of the bits in the binary representation. For example, the binary representation "`00000101`" corresponds to the number 5 since the sign bit is 0 (denoting a positive number). However, "`11111010`" denotes a negative number since the sign bit is `1`. To find out the actual number, we need to take the complement of the bits, which gives "`00000101`".

  Conversely, given a number x, it's binary representation in 1s–complement can be found in the following way. First find the binary representation of $|x|$ (the absolute value of x) in unsigned representation. If x is positive, then the binary representation of $|x|$ is also the binary representation

of x in 1s–complement. However, if x is a negative number, then complement the bits of the binary representation of $|x|$.

Let us consider again 5. $|5| = 5$ and binary representation of $|5|$ is "`00000101`". Since 5 is positive, "`00000101`" is also the binary representation of 5 in 1s–complement.

For -5, we have $|-5| = 5$ which again has the binary representation "`00000101`". Since -5 is negative, we take the complements of the bits and get "`11111010`" as the binary representation of -5 in 1s–complement.

• The disadvantage of 1s–complement is that the number 0 has two different representations. Both "`00000000`" and "`11111111`" corresponds to 0. This is a waste of available representations.

• Most computers use 2s complement: negative numbers are stored by taking the complement of each bit in the positive version and adding 1 to it.

• 2s–complement does not have the drawback of having two representations for 0.

• 2s–complement works in the following way. The first bit of the number will again give the sign of the number. If the number is negative, then we take the complements of the bits in the binary representation and add 1.

• More precisely, given a number x, it's binary representation in 2s–complement can be found in the following way. First we find the binary representation of $|x|$ in unsigned representation. If x is positive, then the unsigned binary representation of $|x|$ is also the binary representation of x in 2s–complement. However, if x is a negative number, then complement the bits of the unsigned binary representation of $|x|$, and add 1.

• For example let us consider -26. We have $|-26| = 26$ which has the unsigned binary representation "`00011010`". Since -26 is negative, we take the complements of the bits and get "`11100101`" and we add 1, which yields "`11100110`" as the 2s–complement representation of -26.

• Conversely, given a 2s–complement binary representation of a number, if the first bit is 0, then it is the positive integer which has this unsigned binary representation. However, if the first bit 1, then we need to subtract 1, and complement the bits. The final binary representation after these operations will give the absolute value of the corresponding negative number.

• For example "`10110110`" in 2s–complement notation corresponds to a negative number. We subtract 1 from this representation and get "`10110101`". Complementing this representation gives "`01001010`" which corresponds to 74, and hence our number is -74.

• Note that "`11111111`" corresponds to -1 in 2s–complement (it was the second 0 of 1s–complement notation).

• Although it may seem more complex to handle complements representations, actually it is better for the computers, since both addition and subtraction can be efficiently implemented if complements representations are used.

Floating point

- They are a model for real numbers.

- In fact, there is also an approach called "fixed point" for representing the real numbers. In the fixed point approach, the real numbers are given by integers scaled by a certain factor. For example, by using a scaling factor of 1000, the real numbers 123.456 1234.56 are given as 123456 and 1234560 (123.456 = 123456/1000 and 1234.56 = 1234560/1000). In the fixed point approach the place of the (decimal) point is fixed (e.g. it is before third digit from the right when the scaling factor is 1000).

- In the case of floating point, the position of the point is floating, i.e. it can be at any position. The position of the point is given with the number by giving the number in the form of

$$\textit{fraction} \times \text{base}^{\textit{exponent}}$$

The *exponent* information gives the actual position of the number.

- They cannot model the real numbers exactly, but just up to a certain precision (e.g. irrational numbers like π and e cannot be represented exactly by a computer – in fact such irrational numbers cannot be stored in any finite space and a computer is a huge but nonetheless a finite space)

- Some rational numbers cannot be represented precisely since numbers are stored in binary representation (e.g. consider representing `0.1` in binary)

- They are stored in memory in two parts : *fraction* and *exponent*

- Floating point operations are computationally expensive operations. Therefore, sometimes the performance of the computers are measured by the number of floating point operations (FLOPS) that they can perform in a unit of time.

Decimal

- Some machines support decimal values directly

- They still use 0s and 1s to represent decimals, however the digits (rather than the entire number) are coded in binary form

- Since at least 4 bits are required to represent a digit, an n digit number requires at least $4n$ bits, although a direct representation in binary might require less space

- Hence this representation is a waste of memory

- E.g. 59 = 01011001

- However, languages like COBOL and C# have built in support for decimal data type.

Boolean types

- Possibly the simplest type
- Only two possible values : *true* and *false*
- Most languages provide boolean as a primitive type
- One popular exception is C, where any non-zero number is used to represent true, and zero is used to represent false
- Although boolean type can be represented using a single bit, it is usually represented by using 1 byte (the reason is that a single bit is not directly addressable)

Character types

- Character data are stored in computers as numeric encodings
- The most commonly used encoding for characters is ASCII, which uses numbers 0–127 to code different characters
- E.g. the letter `A` is encoded by number `65`
- Hence only one byte is used to store a character data if it is encoded in ASCII
- As the computers started to be used for communication around the world, the ASCII set of characters became inadequate
- However later encoding standards like UTF-32 (4 byte long), or UTF-8 (8–bit but variable length encodings) emerged.
- Programming languages provide support for character encodings other than ASCII now.

4.3 Character strings

- A character string is a sequence of characters
- There are two main concerns in the design of character strings in programming language design
  1. Should the strings be simply a special kind of character array, or be a primitive type?
  2. Should strings have static or dynamic length?

- Even when the strings are not a primitive type of a language, they are so important that a set of string functions are provided in a standard library for the language

- In C–C++, the strings are terminated by a `NULL` character, which is represented internall by `\0`.

- Therefore, the length of the string does not have to be kept. When the length of a string is needed, it is sufficient to count the number of characters up to the `NULL` character at the end (speed–memory trade off).

- E.g. `char * str = "abc";` allocates an array of 4 characters, the additional one being for the `NULL` character at the end. The four elements of this array will be : 97, 98, 99, 0 (the ASCII codes of the characters `a`,`b`, `c` and `NULL`).

- There are different design decisions made on handling the length of the strings

- **Static length strings** (FORTRAN) : `CHARACTER(LEN=15) NAMEVAR`
Such a declaration denotes a string of 15 characters. If a shorter string is assigned to `NAMEVAR`, then the remaining characters are filled by blanks. Therefore the string always has the length specified in its declaration.

- **Limited dynamic length strings** (C–C++) : `char NAMEVAR[15]`
The string can have at most the number of characters given in its declaration. Considering that a shorter string can be assigned, the length information must also be kept. C–C++ marks the end of the strings by `NULL` rather than keeping the length explicitly, although the length information can be kept instead.

- **Dynamic length strings** (JavaScript, Perl) : The strings can have any number of characters. When a string which is longer than the currently allocated memory is assigned to the variable, additional memory is allocated. Dynamic allocation–deallocation is needed which slows down the execution, however, it provides maximum flexibility.

4.4 User–defined ordinal types

- **Ordinal types** are the types whose possible values can be easily associated with the set of positive integers.

- For example: integers, char, Boolean.

- The enumeration types and subranges are also ordinal types.

4.4.1 Enumeration types

• An **enumeration type** is one in which all of the possible values (which are called the literal constants and which become symbolic constants) are enumerated in the definition.

• E.g

```
typedef enum { Mon, Tue, Wed, Thu, Fri, Sat, Sun } DayType;
```

• In C, the values are actually enumerated starting from `0`. I.e., the enumeration type given above is equivalent to

```
#define Mon 0
#define Tue 1
#define Wed 2
#define Thu 3
#define Fri 4
#define Sat 5
#define Sun 6
```

• The only difference to this alternative enumeration is that, these values are packed together as the possible values of the type `DayType`. So, when `Sun` is seen in the code, it is understood that it is a value of type `DaysType`.

• The primary design decision required for enumeration types is the following: Is a literal constant allowed to appear in more than one type definition. I.e. are we going to allow the definition of `DaysType` given above and the definition of `SolarObjectType` given below:

```
typedef enum { Sun, Mercury, Venus, Earth, Mars, Jupiter, Saturn,
               Uranus, Neptune, Pluto} SolarObjectType;
```

• As you may have noticed, the literal constant `Sun` appears in both `DayType` and `SolarObjectType`. The problem is that, when an occurrence of `Sun` is seen in the code, how do we infer the correct type of that occurrence?

• Enumeration types increase readability and reliability.

• Note that for a program to execute correctly, the enumerations of the literal constants could directly be used. However, these enumerations will only be meaningful for the programmer. That

is, if the programmer decides to use `0` for Mondays, and consistently does so, the program will execute correctly. However, it will be hard to understand the afterwards, what those 0s correspond to, in the program (even by the programmer).

- It also allows type checking, as mentioned above, and hence increase the reliability of the programs.

4.4.2 Subrange types

- A **subrange type** is contiguous subsequence of an ordinal type.
- For example, `0..9` is a subrange of integer type.
- Subrange types are available in Pascal and Ada.
- The following type definitions are legal in Pascal:

```
type DAYS is (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
subtype WEEKDAYS is DAYS range Mon..Fri;
subtype INDEX is INTEGER range 1..100;
```

- All the operations of the parent type are also available for the subrange types.
- However, you cannot assign a value outside the range, i.e.

```
DAY1 : DAYS;
DAY2 : WEEKDAYS;
...
DAY2 = DAY1;
```

This assignment is legal unless the current value of `DAY1` is `Sat` or `Sun`.

- Subrange types are usually used for array indices.
- Similar to the enumeration types, they increase both readability and reliability.

4.5 Array types

- An **array type** is a homogenous collection of data elements.
- An element in the array is referenced by its position relative to the first element (`A[5]`) which

called **subscript** or **index**.

• The indices of array references can be expressions (`A[x]`) hence run time effort is required to find the correct memory location that is referenced.

• The notation used for array element references is almost universal: either brackets or parenthesis.

• However, using parenthesis decreases readability as the parenthesis notation is also used for function call.

• In some languages the index type of an array has to be integer (e.g. C–C++). However some languages allow index type to be other types, such as boolean, character, enumeration, etc...

4.5.1 Subscript bindings

• The binding of the subscript type of an array variable is usually static, however subscript ranges are sometimes dynamically bound.

• In some languages the lower bound of the subscript range is implicit. E.g. in C it is 0, in Fortran–I it is 1.

• In some languages, the lower bound has to be specified explicitly.

• Similar to the scalar variables, array variables can be categorized in to four classes based on their time and place of memory bindings

1. **Static array**: Subscript range is statically bound and storage allocation is static (occurs at the beginning of the run time) – *speed efficient.*
2. **Fixed stack dynamic array**: Subscript range is statically bound, but the storage allocation is performed at run time – *space efficient*
3. **Stack dynamic array**: Both the subscript range and the storage allocation is dynamic. That is, the subscript range is decided when the declaration elaborated at run time. However, once the allocation is done, the range is fixed during the lifetime of the variable – *flexible*
4. **Heap dynamic array**: Both the subscript range and the storage allocation is dynamic. Subscript range can be changed any time.

• FORTRAN 77 uses static arrays

• Arrays declared in functions in C are fixed stack dynamic arrays

• In C, C++, Java there are array declarations in the following form:

```
int a[]={ 1, 10, 4 };
```

However, in this case, the array length is automatically deduced at compile time from the number of elements in the array.

- Ada supports stack dynamic arrays

- C–C++ allows heap dynamic arrays but they must be handled by the programmer (by library functions) by deallocation and allocation of storage

- JavaScript and Perl directly support heap dynamic arrays

4.5.2 Array implementation (one dimensional)

- An array is represented in memory by using contiguous memory locations.

- The reason for this is to be able to refer to the elements in a fast way.

- Note that, since we may have references in the form `A[x]`, the actual referenced element will only be known in the run time.

- If the address of the first element is known (which could be decided statically for static arrays, or could be decided dynamically when the allocation is done), then the address of an element can be computed as:

$$address(A[k]) \quad = \quad address(A[0]) + (k * element\_size)$$

where $A[0]$ is the address where the array starts.

- Hence an addition and a multiplication is required to find the address of the memory location of `A[k]`.

- Note that, when the lower bound of the array is not fixed to 0, then this formula must be generalized to

$$\begin{aligned} address(A[k]) &= address(A[lb]) + ((k - lb) * element\_size) \\ &= address(A[lb]) - (lb * element\_size) + (k * element\_size) \end{aligned}$$

where lb is the lower bound of the subscript range, and hence $A[lb]$ is the memory location where the array starts.

- The first line requires one subtraction, one multiplication, and one addition for each reference of the form `A[k]`.

- The second line requires one multiplication and one addition for each reference of the form `A[k]` (note that $address(A[lb]) - (lb * element\_size)$ can be computed only once, and can be used in each reference). Hence it will yield a better run time performance.

4.5.3 Array implementation (multidimensional)

- Implementation of multidimensional arrays are little bit more complex.
- Note that, the memory is one dimensional.
- Therefore we need to map a multidimensional structure into the one dimensional memory structure.
- There are two common ways for this mapping: row major and column major
- Let us consider 2–dimensional arrays first, and consider the example

$$\begin{array}{ccc} 1 & 2 & 3 \\ 4 & 5 & 6 \end{array}$$

- In row major order, this 2–D array will be placed into the memory as

1
2
3
4
5
6

- In column major order, this 2–D array will be placed into the memory as

1
4
2
5
3
6

- When a row major order is used, the address of `A[j,k]` can be computed as

$$\begin{aligned} address(A[j,k]) &= address(A[0,0]) + (\text{“no\_of\_rows\_before } j\text{”} * \text{“size of a row”}) + (k * element\_size) \\ &= address(A[0,0]) + (j * element\_size * R) + (k * element\_size) \end{aligned}$$

where R is the number of elements in a row.

- Since “$element\_size * R$” can be computed before the references, we need two multiplications and two additions to find the address of $A[j,k]$
- Note that, the calculation above is given by assuming that subrange starts from 0 for both of

the dimensions.

- The calculation will be similar and symmetric when the column major order is used.

- When there are more than two dimensions, the calculation of the address must be extended to cover the additional dimensions. Considering row major order and assuming that array indices start from 0 for all subscript ranges:

$$address(A[k_1, k_2, \ldots, k_n]) \quad = \quad \ldots$$

4.6 Records (Structs)

- A **record** is possibly heterogenous aggregate of data elements.

- Individual data elements in a record are referenced by using the field names.

- Implementation of records are similar to the arrays in that adjacent memory locations are used.

- The address of a field is computed by the offsets of the other fields in the record.

```
struct foo {
  int a;
  char b;
  double c;
  }

x.c = ...
```

The address of `x.c` is the address of `x` (which is also the address of `x.a`) + 5 (4 bytes for integer, and 1 byte for character).

4.7 Union types

- A **union** is a type that can store different type of values at different times.

```
union MyUnion {
  int i;
  char c;
  }

MyUnion x;
```

- Assume that an integer is represented by 4 bytes, and a character represented by 1 byte.

- Since only one of the fields will be used (will be storing a meaningful data) at a time, maximum of the memory requirements of the fields is enough to store a union variable.

- In the case of `MyUnion` example given above, only 4 bytes will be enough to store the variable `x`.

- When an assignment of the form `x.i = 5` is made, all of these 4 bytes will be used.

- When an assignment of the form `x.c = 'a'` is made, only the first byte (of the 4 bytes) will be used.

- One design decision is to whether enforce type checking on unions. I.e. should the field of a union be allowed to be referred without a prior assignment after an assignment to a different field?

```
x.c = 'a';
x.i = 5;
char d = x.c;
```

Normally, this code fragment is most probably erroneous since after the first assignment, the `i` field of `x` has a meaningful data and the `c` field is useless. The second assignment, therefore, is referring to a possibly garbage data.

- In order to detect this type error, the currently used field information must be kept with the variable, and updated and checked at run time (e.g. Ada does this).

- Most languages (C, C++, FORTRAN, Pascal) do not perform this check, and hence unions are not safe and must be used with extreme care.

4.8 Pointer types

- A **pointer** type is one in which the variables can have values that correspond to the memory addresses. In addition to that, a special value usually called as `NIL` or `NULL` is used to denote that a pointer value currently does not point to a valid memory address.

- Major use of pointers is to be a handle to heap dynamic variables (nameless, anonymous variables) which are allocated at run time from the part of the memory called the **heap**.

- A pointer variable can be referenced as any other variable, that is, its r–value can be accessed. r–value of a pointer variable is simply the address that is currently pointed by the pointer variable.

- The more interesting reference on a pointer variable is the **pointer dereferencing**, in which the r–value of the pointer variable is treated as an l–value. Hence the contents of the memory location

whose address is currently kept at the pointer variable is accessed.

- Pointer arithmetic is also possible for C and C++

```
int a[10];
int * p;
int b;

p = a; // p = &a[0]
p++; // p = &a[1]
b = *(p + 4); // b = a[5]
```

- Pointers can also point to functions. With this feature, one can pass functions as parameters to other functions.

- In Pascal, pointers can only point to the heap dynamic variables. Therefore, a pointer can only be assigned by a dynamic allocation statement, or from another pointer (no address operators like & in C).

- **Reference types** are special pointer types. In C++, reference variables are constant pointers that are implicitly dereferenced.

- Since they are constant, they must be initialized when they are defined (no assignment to a constant after its initialization).

4.8.1 Issues Related to Pointers

- There are two possible major issues with the pointer usage.

  - **Dangling pointers**: A pointer that contains the address of a heap dynamic variable which is not allocated for the use of the program under consideration. The typical reason for dangling pointers is to deallocate an allocated memory (and forget to assign the pointer that is used to access it to `NIL`), or have an uninitialized pointer (declare a pointer variable, and do not assign it to a value). Therefore it is a good programming practice to assign `NIL` to a pointer variable just after deallocation of the memory pointed by it. However, even when this practice is applied, there may still appear dangling pointer because of aliasing.

  - **Lost heap dynamic variables**: A heap dynamic variable which is allocated, but which cannot be accessed through any pointers in the program. Such variables are also called as **garbage**. This problem is also known as **memory leakage**.

- In C and C++, pointers can point to any memory location, including heap part, stack part, (or even the parts of the memory that do not belong to the programs).

- Due to explicit deallocation (`delete` in C++ and `free` in C), dangling pointers are easily created.

Solutions to Dangling Pointers

Tombstones

- One solution is to use tombstones.

- A tombstone is an extra heap cell which points to a heap dynamic variable.

- When a heap dynamic variable is created (that is when a dynamic memory allocation is performed), a tombstone is created for it, and the tombstone is set to point to the newly created heap dynamic variable.

- The programmer's pointers point to the tombstone of the heap dynamic variable, not to the heap dynamic variable.

- When a pointer dereferencing takes place, first the tombstone is reached, then the address pointed by the tombstone is accessed.

- When there are more than one pointers that are alias to each other, then they all point to the tombstone of the heap dynamic variable.

- When explicit deallocation is performed by using any of these aliases, the heap dynamic variable is deallocated, but the tombstone survives and it is set to NIL.

- Further access attempts to the deallocated heap dynamic variable will be caught, as they must also go over the tombstone, which is set to NIL.

- The obvious disadvantage of this approach is speed and memory. Two level indirect access (through tombstones) slows down the references, and tombstone memory cells never return to the free memory pool.

Locks and Keys

- Every pointer variable is a structure : an address + an integer **key**

- Every heap dynamic variable is augmented with an additional integer field called the **lock**

- When a heap dynamic variable is created and it's address is assigned to a pointer variable, the

lock of the heap dynamic variable and the key of the pointer variable are set to the same value.

• When a pointer dereferencing takes place, the key of the pointer and the lock of the heap dynamic variable (whose address pointed by the pointer variable) are compared. The access is valid only if the key and the lock match. Otherwise, a run time error occurs.

• When a pointer is copied from another pointer, the key field must also be copied.

• When a heap dynamic variable is deallocated, its lock field is set to an illegal lock value. Hence any further reference after deallocation will have unmatched key–lock pairs.

Heap management

• Heap management is related to two operations: dynamic allocation and dynamic deallocation.

• Usually, allocation is much simpler than the deallocation.

• When a language has an explicit deallocation statement, then it is user's responsibility to free the unused memory and give it back to the pool of available memory.

• However, when such a statement is not a part of the language, then the run time engine of the language must manage the heap to keep track of the parts that are allocated but not used any more.

• The problem is to detect which parts of the allocated heap memory is not being used any more.

• We will introduce two techniques briefly:

  • **Reference counters:** In each cell there is an integer counter field. This field is set to 1 when the cell is allocated.

```
int *a, *b;
...
a = b;
```

  Such a pointer assignment will decrement the reference counter of the cell that is pointed by `a` before this assignment, and it will increment the reference counter of the cell pointed by `b`.

  When a reference counter becomes 0, it means that there is currently no pointer pointing to that cell. Hence it can be immediately returned to the list of available space.

  There are three main problems related to reference counters:

  1. The memory used for the reference counters.
  2. The additional time spent for updating the reference counters.
  3. Circular pointers.

- **Garbage collection:** The run–time system allows garbage to accumulate. Every memory cell in the heap has an accessibility flag which is used only when the garbage collector kicks in (usually when there is no available memory).

  The garbage collector first sets the flags of the entire heap to false. Then all the pointers in the program are traced into the heap, and the flags of cells that are accessible are marked as true. Then, the memory cells that are marked as false are considered to be garbage and added to the list of available memory space.

Chapter 5

Expressions and assignment statements

- The most basic component of imperative languages are variables and assignment statements.
- An assignment can copy the value of a variable into another variable.
- However, usually the value that will be assigned (RHS of an assignment) is given as an expression.
- Therefore it is important to understand the expressions.
- Arithmetic expressions are the most widely seen type of expressions.
- Operators : unary, binary, ternary, n–ary.
- Precedence and associativity of operators.
- Infix, postfix, prefix notations for writing expressions.
- Also mixfix notations are used:

```
a = (b==0) ? c : d;
```

The ternary operator `?:` is used in mixfix notation, where the operand and the operators are mixed.

5.1 Operator evaluation order

- Variables in expressions are evaluated by fetching their value from the memory.
- An expression in parentheses must be evaluated before its value can be used by an operator.
- If the evaluation of operands have no side effects, then the order of evaluation of the operands has no significance. However, when the expression evaluations have side effects, then the order will

be important.

• A **side effect** of a function (sometimes called the **functional side effect**) occurs when the function changes the value of one of its parameters or a global variable.

```
a = a + f(a)
```

If the function `f` has no side effect on `a`, then the order of evaluations of the operand values in `a + f(a)` will not be significant. Otherwise, order of evaluation of `a + f(a)` is important.

• Assuming that `a` currently has the value 10, and the function `f` multiplies the input parameter by 2 and assigns it back to the input parameter, the expression may have either the value 30 or 40 depending on whether we evaluate `a` or `f(a)` first.

• This is counterintuitive since the addition operator in `a + f(a)` is commutative. Hence `a + f(a)` and `f(a)+a` should mean the same value.

• Disallowing functional side effects would mean elimination of two-way function parameters, access to non–local variables. However these features provide a great deal of flexibility, hence they are hard to be eliminated form a language.

• Short circuit evaluation:

```
x * timeConsumingFunction(a,b,c)
```

When `x=0`, there is actually no need to calculate the value of the other operand, hence time is saved.

Sometimes, this feature can be used to eliminate errors in a readable way:

```
while ((index < upper_bound) && (a[index] != 5)) {
   ...
   }
```

Chapter 6

Statement Level Control Structures

6.1 Introduction

- In imperative languages, expression evaluation and assignment to variables are two basic activities.

- However, these two features are not sufficient to write interesting programs.

- For flexibility and expression power, we also need to be able to:

  1. chose between different alternative paths in the program

  2. repeat the execution of a sequence of statements

- In other words, we need *selection* and *loop* control statements.

- In fact, an *if* statement (to be able to select) and a *goto* statement (to be able to repeat) in the language design is all what we need to write interesting programs.

- A "conditional goto" statement will cover both of these, and hence it will be sufficient.

- The first programming language, FORTRAN, was affected by the machine language, and hence the control statements of FORTRAN was simple reflections of the machine language, which included a conditional goto statement.

- At that time, the difficulty of developing programs was not well understood, and readability and writeability were not considered.

- The research and discussion on control statements between mid-60s and mid-70s, resulted in the following conclusion: an "if statement" and a "logically controlled iteration statement" (e.g. a "while" statement) are enough to describe all algorithms.

- This fact, combined with the difficulty of the readability and writeability of the programs with

"goto" statements, lead to the debate on the usage of "goto" statements.

- Why do we have more than one iteration statement in contemporary languages?

Although just one "logically controlled iteration statement" is enough, to increase readability and writeability of the languages, there are a number of iteration statements (for, repeat, do–while, while–do, etc.).

6.2 Single entry control structures

- A **control structure** is block of statements whose execution is controlled by the control statements.

- For example, the body of a while loop is a control structure, then and else parts of an if statement are control structures.

- Single entry to a control structure means that, if a control structure is executed, there is a unique statement in the control structure that is executed before the other statements in the control structure.

- For example, in the body of a loop, whenever the control flows into the body, the first statement in the body is executed first.

- Single entry approach can be violated if "goto" statements are used. In such a case, one can jump to a statement in the middle of the body of a loop, and start the execution of the control structure.

- Modern, structured programming languages accept the single entry approach for their control structures.

- It is a widely accepted fact that, if the control structures have a single entry, the programs become more readable.

- The reason for this is that, the program text viewed on a piece of paper, or on the screen, reflects the way the program will be executed at run time. Hence, it becomes easier to understand what the program will perform when it is executed. If there are too many jumps within the program, it becomes harder to trace the execution of the program by looking at the program text.

6.3 Selection statements

- Selection statements provide means to select between two or more alternative execution paths.
- Such statements are fundamental to all programming languages.
- Without the selection statements, the programs would not be flexible, and it would not be possible to write programs that considers different conditions.
- There are in general two classes: 2-way and n-way selection statements.
- The selection statements of contemporary languages are quite similar to each other. However, the selection statements today evolved from quite different forms and semantics.
- For example, FORTRAN included an arithmetic "if" statement with 3-way branching

```
     IF (expr) 10, 20, 30
10   ...
     ...
     GO TO 40
20   ...
     ...
     GO TO 40
30   ...
     ...
40   ...
```

The semantics of this statement is as follows:

- If `expr` evaluates to a negative value, then jump to statement labelled as `10`
- If `expr` evaluates to zero, then jump to statement labelled as `20`
- Otherwise (`expr` evaluates to a positive value), jump to statement labelled as `30`

- Another n-way selection statement form FORTRAN

```
GO TO (lbl 1, lbl 2, ..., lbl n), expr
```

First, the `expr` (which is an integer expression) is evaluated. If it evaluates to 1, then a jump to "`lbl 1`" is performed, if it evaluates to 2, then a jump to "`lbl 2`" is performed, so and so forth...

- We arrived at the contemporary form of "if", "switch", "case" statements starting from the forms of selection statements given above.

6.4 Iterative statements

• **Iterative statements** provide means to repeat a block of statements to be executed 0, 1, or more times.

• Without iterative statements, program would be inflexible, and the programmer would need to unfold the iterations resulting in huge programs, requiring large amounts of memory to run.

• There are different design issues for the iterative statements:

  • Iteration control: counting, logical, combined

  • The place of control: pretest, posttest, ...

6.4.1 Counting loops

• Counting loops have a **loop variable**, an **initial value**, a **terminal value**, and a **step size**. The loop variable is assigned to the initial value, and after each iteration the loop variable is increased step size amount, until it reaches to the terminal value.

6.4.2 Logically controlled loops

• Logically control loops have a test condition which terminates the loop.

• In pretest loops (e.g. while–do loops), the test condition is evaluated before the iterations of the loop.

• In the posttest loops (e.g. repeat–until or do–while loops), the test condition is evaluated after the iterations of the loop.

• The difference is that, the body of a posttest loop will be executed at least once.

• This is a typical example of using the programming syntax (at language design time) to reflect the program semantics. Having the condition of posttest loops at the end of the loop structure in the language syntax design, helps the programmer understand that the loop body will be executed at least once before the termination condition of the loop is checked out.

• In other words,

```
do while (termination condition) {
  loop body
  }
```

would not be as good of a syntax as

```
do {
  loop body
  }
while (termination condition)
```

since the latter alternative better reflects the fact that when the control reaches to this statement, first the loop body will be executed and then the termination condition will be evaluated.

- Is the "for statement" of C a counting loop, or logically controlled loop?

```
for (expr1; expr2; expr3) {
 ...
}
```

It is more like a pretest logically controlled loop, since `expr2` is tested before every iteration of the loop, and `expr2` can be any test (not necessarily the test for a loop variable going beyond the terminal value).

Actually, it is equivalent to:

```
expr1;
while (expr2) {
  ...
  expr3;
  }
```

- Some languages allow user–located test condition.

- For example,

```
loop
  ...
  if (flag==false) then exit;
  ...
endloop;
```

or

```
loop
  ...
  exit when (flag==false);
  ...
endloop;
```

• The design and implementation of such features are simple. The only complication is in the situation when we have nested loops. The problem is to decide which loops will be terminated.

The design approach for resolving such situations has been to name the loops, and use these loop names at exit statements:

```
OUTER_LOOP:
  for ROW in 1 .. MAX_ROWS loop
INNER_LOOP:
    for COL in 1 .. MAX_COLS loop
      SUM = SUM + A[ROW,COL];
      exit OUTER_LOOP when SUM > 1000
    endloop INNER_LOOP;
  endloop OUTER_LOOP;
```

• The exit statement in these examples transfer the control to the next statement that follows the loop.

6.4.3 Iterations based on data structures

• Recall that in counting loops, the values that are assigned to the loop variable are controlled by the initial and terminal value, and the step size.

• In data structures based iterations, the loop variable is assigned to the values currently stored in a data structure.

• Perl has this feature:

```
@names = ("Ali", "Veli", "Ayse", "Fatma");
foreach $name (@names) {
  print $name;
  }
```

6.4.4 Unconditional branching statements

- The unconditional jump statement (i.e. goto statement) is quite powerful and flexible.
- However, its use is highly abusable sabotaging the readability of programs.
- Dijkstra noted this fact as "it is too much an invitation to make a mess of one's program"
- Knuth argued that "there are occasions when the efficiency of the goto statements outweigh its harm to readability".
- Following the debate over the use of goto statements, some languages completely banned the goto statements (e.g. Java), some languages allowed it but discouraged the programmers to use it.
- Also, other approaches are taken by restricting the use of gotos (e.g. the exit statements from the loops are camouflaged goto statements).

Chapter 7

Subprograms

7.1 Introduction

- It has been noticed long before that abstraction is of paramount importance for developing correct programs and for readability and writeability.

- Process abstraction and data abstraction are two fundamental abstraction facilities.

- The importance of data abstraction was only appreciated and implemented in 1980s.

- However, process abstraction has been central to all programming languages (even in the era of punch cards).

- Process abstraction through the use of subprograms results in several advantages:

  - Decreased coding time: Each subprogram is written exactly once in the source program. Invocations of the same code are simply requested by the calls to the subprogram. Without the subprograms, the same code fragment would have to be written over and over in the source code.

  - Decrease in memory requirements: A single copy of the subprogram code is enough in the memory. The calls to the subprogram are implemented by jumps to the subprogram code. There is no need to insert the entire body of the subprogram at every call statement.

  - Increased writeability: The programmer can focus on the purpose of the subprogram while writing the code of the subprogram. For example, when writing a subprogram for sorting a list of numbers given in an array, the programmer does not need to know or need to consider how and for what purposes this sorting subprogram will be called. All she/he needs focus is to correctly implement a sorting algorithm. Once the subprogram is written in a correct way, then every invocation of the subprogram will be correct.

  - Increased readability: The functionality performed by a subprogram is abstracted. Whenever this functionality is required, a simple call to the subprogram is placed into the source code.

7.2 Fundamentals of subprograms

- The basic features of subprograms are common to all languages:
  - Each subprogram has a single entry point. When a subprogram is called, the body of the subprogram is executed starting from the first statement in its body. This is the (single) entry point of the subprogram.
  - Execution of the calling subprogram is suspended during the execution of the called subprogram. When a subprogram `A` calls another (not necessarily distinct) subprogram `B`, `A` stops its execution temporarily and waits for the termination of `B`.
  - Control always returns to the caller when the subprogram execution terminates. When a subprogram `B` terminates which was called by a subprogram `A`, the subprogram `A` resumes its execution.

- A **subprogram definition** describes the interface to and the actions of the subprogram abstraction.
- A **subprogram call** is the explicit request that the subprogram be executed.
- A subprogram is said to be **active** if it is currently executing its statements.
- The **subprogram header** serves several purposes:
  - name of the subprogram
  - (optional) list of parameters
  - (optional) return types

- Sometimes the *prototypes* or *signature* of a subprogram must be given in order to be able to call the subprogram before its declaration.

```
#include <stdio.h>

void fact(int, unsigned int *); // signature of fact
```

signature of the subprogram `fact`

```
main() {
  int x = 5;
  unsigned int result = 1;

  // fact is used here before its actual definition
  fact(x, &result);
```

actual variables of the subprogram `fact` for this call

```
  printf("Factorial of %d is %u.\n", x, result);
}
```

formal variables of the subprogram `fact`

```
void fact(int y, unsigned int * out) {
   if (y < 0) {
      fprintf(stderr, "Cannot compute %d!\n", y);
      exit(1);
      }

   if (y > 1) {
      *out = *out * y;
      fact(y-1, out);
      }
}
```

7.3 Parameters

- The parameters in the subprogram header are called the **formal parameters**.
- The parameters in the subprogram calls are called the **actual parameters**.
- In most languages, the binding of actual parameters to formal parameters is done by simple position (the first actual parameter corresponds to the first formal parameter, the second actual parameter corresponds to the second formal parameter, ...).
- Such parameters are called *positional parameters*.
- In some languages, **keyword parameters** are used. The name of the formal parameter to which an actual parameters corresponds to is given at the subprogram call.

```
fact(&result => out, x => y)
```

The disadvantage is that, the names of the formals parameters have to be known.

7.4 Procedures and functions

- There are two classes of subprograms : procedures and functions
- Both of them can be seen as means to extend the programming language.
- Procedures extend the set of statements provided by the language (e.g. if the language does not have a sort statement, a procedure can be written and used as if it were a built-in sort statement).
- Functions extend the set of operators provided by the language (e.g. if the language does not have an exponentiation operator, a function can be written and used as if it were a built–in exponentiation operator).
- Both functions and procedures can produce results by changing a nonlocal variable that is visible inside the subprogram, or by changing the actual parameters passed.
- Functions can also produce results by returning some values.
- If a function does not change a non–local variable, then it is a **pure function** (or equivalently, it has no **side effect**).

7.5 Local subprogram referencing environments

- A **referencing environment** of a (sub)program is the set of variables that can be accessed by that (sub)program.
- Subprograms are generally allowed to define local variables, and hence they define a local referencing environment.
- The variables defined in a subprogram are called local variables because usually the access to these variables are restricted to the subprogram only.
- The local variables of a subprogram can either be static or stack dynamic.
- The storage binding of stack dynamic variables are performed when the subprogram starts to execute, and they are unbound when the subprogram terminates.
- This has several advantages:
  - Recursive subprograms would not be possible with static allocation
  - The memory can be reused

- The disadvantages are:
  - The variables have to be allocated space and initialized when the subprograms starts and have to be deallocated when it terminates (therefore the time cost of subprogram call increases)
  - Indirect referencing has to be used (the address of these variables can only be known at run time) which also slows down the execution
  - History sensitive information cannot be kept at stack dynamic variables

- Since Algol 60, most programming languages chose to implement the variables of the procedures as stack dynamic variables.

- Ada, Pascal, C (except when declared with the keyword `static`) and Java have their local subprogram parameters implemented as stack dynamic.

- Note that, the main disadvantage of static variables is that they do not allow recursion. However, static variables are quite time efficient.

- Therefore, FORTRAN 90 allows the programmer to declare a function to be recursive explicitly. The local variables of such recursive functions are implemented as stack dynamic, whereas the local variables of the other functions are implemented as static for time efficiency.

7.6 Parameter passing methods

- Parameter passing methods are the ways in which the parameters are transmitted to and/or from the called subprograms. Hence they define the assignments between the formal parameters and the actual parameters.

- Formal parameters are characterized by the following semantic models:
  - **in–mode**: An in–mode formal parameter can only receive data from the corresponding actual parameter.
  - **out–mode** : An out–mode formal parameter can only transmit data to the corresponding actual parameter.
  - **inout–mode**: An inout–mode formal parameter can both receive data from and transmit data to the corresponding actual parameter.

- The actual data transfer can either be in the form of value assignment, or by passing a reference to the actual variable which is used to access the actual parameter.

- There are several implementation methods for these semantic models, which are discussed in the following sections.

7.6.1 Pass–by–value

• When a parameter is passed–by–value, the actual parameter is used to initialize the corresponding formal parameter, which is then used as a local variable of the called subprogram. Hence pass–by–value implements the in–mode semantics.

• Pass–by–value is usually implemented by actual data transfer.

• This requires additional storage for the formal parameter.

• It can be implemented as reference passing, but one has to write protect the actual parameters in this case.

• The main disadvantages of implementing pass–by–value by actual data transfer are the additional memory and the time required make the assignment (consider the case where the data being passed is a long array).

• Note that the actual parameter of a pass–by–value formal parameter can be a variable, or a constant, or a complex expression.

7.6.2 Pass–by–result

• When a parameter is passed–by–result, no data is transmitted from the actual parameter to the corresponding formal parameter.

• The formal parameter acts as a local variable in the called subprogram.

• Just before the subprogram terminates, the value of the formal parameter is transferred to the corresponding actual parameter.

• Hence, it implements the out–mode semantics.

• Note that, an actual parameter corresponding to a pass–by–result formal parameter must always be a variable (it cannot be a literal constant, or an expression), to be able to decide the variable that is going to be assigned.

• Pass–by–result is also usually implemented by an actual data transfer.

• One problem that arises in pass–by–result is the "actual parameter collision"

```
sub (x, x)
```

If both of the formal parameters of `sub` are pass–by–result parameters, then the semantics of such a call is not clear.

• Another problem is to decide the time the reference to the actual parameter is fixed.

```
procedure set_element (int y, int z) {
  y = 4;
  z = 5;
  }
...
x = 3;
set_element (x, A[x]);
```

Assume that `y` and `z` are passed–by–result.

If the reference to `A[x]` calculated at the beginning, `A[3]` will be set to 5, however if it is calculated at the end, `A[4]` will be set to 5.

7.6.3 Pass–by–value–result

- This is an implementation of inout–mode semantics.
- It is the combination of pass–by–value and pass–by–result.
- In this method, formal parameters are always separate variables which act as a local variable of the subprogram.
- Hence the data is moved back and forth between the actual and the formal parameters.

7.6.4 Pass–by–reference

- This is another implementation method for inout–mode semantics.
- Rather than moving the data between actual and the formal parameters, the access path of the actual parameters are passed to the called subprogram.
- In effect, the actual parameters are shared with the called subprogram and the caller.
- Advantage is that there is no additional space or copy time needed.
- Disadvantage is that access to such shared variables require more time (since they are in the referencing environment of the caller).
- Aliases can be created

```
procedure foo (int first, int second) {
  // first and second are
  // considered to be different variables
  ...
  }
foo (x, x);
```

7.7 Passing subprograms as parameters

• There are some situations where passing subprograms as parameters to other programs become handy.

• The idea is simple, however it can be quite confusing and there are some design decisions required.

• Passing a simple pointer to the subprogram would be a simple solution. However, some complications have to be considered.

• Type checking the parameters is one of the problems.

• Algol 68 provides the following way of usage of this feature which also enables the type checking of the parameters:

```
procedure integrate(function fun(x:real) : real;
                    lowerbd, upperbd : real;
                    var result : real);


  ...
  var funval : real
  begin
     ...
     funval := fun(lowerbd);
     ...
  end;
```

• Since the protocol is given for the subprograms that will be passed as parameter, type checking on the subprogram that is used as the actual parameter can be performed.

• In C and C++, functions cannot be passed as parameters directly. However pointers to functions can be passed. The type of the pointer to a function must also include the prototype.

```
#include <stdio.h>
int Add (int a, int b) { return a+b; }
int Sub (int a, int b) { return a-b; }

void foo(int a, int b, int (*pt2Func)(int, int)) {
  int result = pt2Func(a, b);
  printf("Result is %d\n", result);
}

main() {
  foo(2, 5, &Add);
  foo(2, 5, &Sub);
}
```

- Another question is the referencing environment of the passed subprogram when it executes.
- Consider the following example:

```
procedure SUB1;
  var x: integer;
  procedure SUB2;
    begin
      write('x = ', x);
    end; { SUB2 }
  procedure SUB3;
    var x : integer;
    begin
      x := 3;
      SUB4(SUB2);
    end; { SUB3 }
  procedure SUB4(SUBX);
    var x : integer;
    begin
      x := 4;
      SUBX;
    end; { SUB4 }
  begin { SUB1 }
    x := 1;
    SUB3;
  end; { SUB1 }
```

- The subprogram can be executed in the referencing environments given below:

- **Shallow binding:** The environment of the call statement that actually calls the subprogram that is passed as the parameter. If shallow binding is used, the program above would print 4.
- **Deep binding:** The environment of the subprogram that is passed as the parameter. If deep binding is used, the program above would print 1.
- **Ad hoc binding:** The environment of the call statement that passes the subprogram as a parameter. If ad hoc binding is used, the program above would print 3.

• Note that, passing subprograms as parameters can be used as a workaround for some visibility problems. In the example given below, `SUB3` would not be visible to `SUB4` normally. Hence, `SUB4` could never call `SUB3`. However, `SUB2` passes `SUB3` to `SUB4` as a parameter, hence provides access to `SUB3`.

```
program SUB1;
  procedure SUB2;
    procedure SUB3;
      begin
      ...
      end; { SUB3 }
    begin
      SUB4(SUB3);
    end; { SUB2 }

  procedure SUB4(SUBX);
    begin
     ...
     SUBX;
     ...
  end; { SUB4}
  begin
    SUB2;
  end;
```

7.8 Overloaded subprograms

• An overloaded operator is an operator that has multiple meanings.

• Note that, addition of two integers and addition of two floating point numbers (although they are quite similar) are actually different operators. The machines have different instructions for these operators since the memory structure that they operate on are different (e.g. 4 bytes vs 8 bytes).

• If an operator is overloaded, then the actual operator referenced in an expression is disambiguated

by the type of the operands.

• **Overloaded subprograms** are those that have the same name as some other subprograms, but they differ in the number of parameters, type of parameters, or in the case of functions, they may differ in their return types.

• Although it is usually the case and a good programming practice, it is not necessary for overloaded subprograms to perform similar jobs. For example a subprogram named as `SELECT` can be a subprogram that selects a random element from an array, or it can be a wrapper subprogram to apply SQL select on a database.

• Overloaded subprograms with default parameters may be ambiguous, hence they must be avoided.

```
void foo (int i = 0) { ... }
void foo () { ... }
...
foo();
...
```

7.9 Generic subprograms

• A concept related to overloaded subprograms is generic subprograms.

• A **generic subprogram** or a **polymorphic subprogram** is one that can operate on different types of parameters in different activations.

• The difference between overloaded and generic subprograms is the following:

  • Generic subprograms implement the same algorithm over different types, whereas overloaded subprograms do not necessarily implement the same algorithm (recall the `SELECT` example in the previous section).

  • The types that an overloaded operator will operate on are fixed at program design time, however, a generic subprogram can be called with different types in each activation.

  • The number of parameters of an overloaded operator may be different in different versions of it. However, a generic subprogram always has the same number of parameters.

• Still, overloaded and generic subprograms are conceptually similar. Overloaded subprograms also provide some kind of polymorphism. The kind of polymorphism provided by overloaded subprograms is called **ad hoc polymorphism**.

Chapter 8

Implementing subprograms

8.1 Issues in subprogram call

- A subprogram call in a typical language has a number of actions associated with it.

  1. The call must include the mechanism required for the kind of parameter passing method used in the call (the formal parameters, if they are viewed as local variable of the callee, must be allocated; data transmission from the actuals to the formals must be handled, etc.)
  2. Memory must be allocated for non–static local variables of the callee.
  3. The execution status of the caller must be saved (so that it can return back to the same position when the called subprogram terminates).
  4. The control must be transferred to the callee.
  5. An access mechanism must be provided to those variables that are not local to the callee but visible in the callee (e.g. global variables)
  6. The data transfer must be handled for out and inout mode formal parameters when the subprogram terminates.
  7. The space allocated for the local variables must be deallocated when the subprogram terminates.
  8. The control must be transferred back to the caller when the subprogram terminates.

8.2 A simple case – no recursion

- Assume that recursion is not allowed (e.g. FORTRAN 77).
- This assumption makes the subprogram implementation fairly easy.
- Note that, under this assumption, we can use static storage binding for the local variables of a

subprogram.

- In such a case, a subprogram needs the following data structure

Functional value
Local variables
Parameters
Return address

- **Functional value** field in this activation record structure is used to store the return value if the subprogram is a function. It will be accessed by the caller of the subprogram after the termination of the subprogram, to read the value returned by the function. This field is not actually necessary if the subprogram is a procedure.

- **Local variables** field is the part of the memory that will be used to bind the local variables.

- **Parameters** field is the part of the memory that will be used for the formal parameters of the subprogram.

- **Return address** field will be used to store the return address that the control should be passed to when the subprogram terminates.

- This data structure is called an **activation record**.

- Normally, each activation of a subprogram would require a different instance of its activation record, which is called an **activation record instance**.

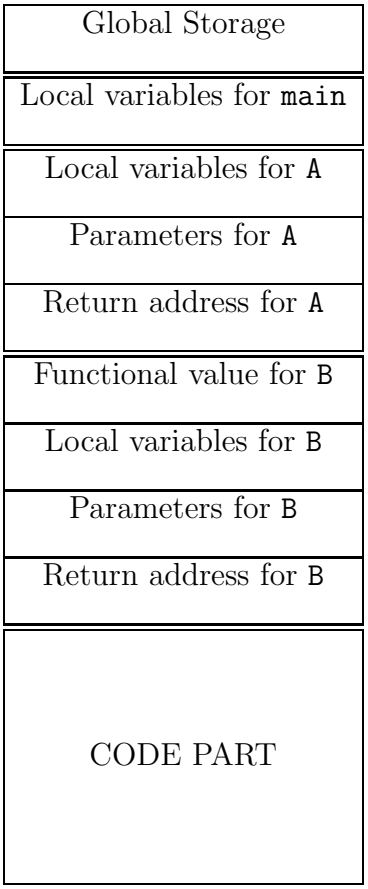
- However, since we are now considering the case where the subprograms are not recursive (directly or indirectly), there can be at most one activation of a subprogram at any given time, and hence a single activation record instance (which can be allocated at load time) would be sufficient. The activation record instance of a subprogram can continue to exist even if the subprogram is not currently active.

- If static scoping is used, then the nonlocal references can also be resolved at load time. Recall that in static scoping, a nonlocal reference of a subprogram `A` for a variable `x` can be resolved at load time. If the declaration to be used for such a nonlocal reference to `x` in `A` is found to be a declaration in a subprogram `B`, then since we have only one activation record for `B` at any given time, we can decide the address to be used for `x` at load time.

- However, if dynamic scoping is used, then the nonlocal references will need to be resolved at run time. Although the subprograms will have only one activation (hence they will have only one activation record instance) at any given time, the dynamic parent of `A` having the nonlocal reference `x` can differ from one activation of `A` to another. Therefore, we may need to use different addresses for the nonlocal reference `x` in `A` in different activations of `A`. If `A` is called by `B` (which has a declaration for `x`), then `x` of `B` should be used. However, if `A` is called by `C` (which has a declaration for `x`), then

`x` of `C` should be used.

• For example, if we have 2 subprograms (a procedure named `A` and a function named `B`) and a main program, the following memory layout can be created at load time for the entire program.



8.3 More complex activation records

• In the previous section, we have considered a very simple case where there are no recursive subprograms.

• When recursive subprograms are allowed, there are certain complications that have to be resolved

1. There can be more than one instance of the activation record of a subprogram.
2. Local variables of the subprograms may need to be dynamically allocated. Note that, in the simple case of no recursion, since there was only one instance of each activation record, we were always able to allocate the local variables at load time once and continue to use the same memory locations for the variables throughout the execution of the program.
3. Since there are more than one instances of activation records, static scoping rules for nonlocal references must also be considered at run time. Recall that we have noticed that if dynamic scoping is used, the nonlocal references must be resolved at run time even in the case where recursion is not allowed.

• The format of the activation records are fixed. However, the size of the activation record can only be decided at run time under some conditions

```
void sub(int size) {
 int myArray[size];
 ...
 }
```

In such a language, the size of the local variables of a subprogram (e.g. `myArray` in the example above) cannot be decided statically. When the procedure `sub` is activated (called), depending on the actual value passed as `size`, the length of the array `myArray`, hence the memory space required within the the activation record instance for the array `myArray` for this activation of `sub` will be known.

• When we have recursion in the picture, we usually need two more fields in the activation record structure

- **Static link:** It is sometimes called **static scope pointer**. It is a pointer to the activation record of the most recent activation of the static parent of the subprogram. This field is used to access the nonlocal references.
- **Dynamic link:** This field is a pointer which points to the activation record instance of the caller subprogram. When the size of an activation record instance cannot be known at compile time, this field provides the required information for the deallocation of the activation record instance.

• With these additional fields, a typical layout of an activation record looks like as shown below:

Functional value
Local variables
Parameters
Dynamic link
Static link
Return address

• Assume that a subprogram `foo1` is activated, hence an ARI (activation record instance) for `foo1` is created. Also assume that `foo1` activates another subprogram `foo2`, hence an ARI for `foo2` is also created. Since, `foo1` cannot terminate before `foo2`, the ARI of `foo1` will not be deallocated before the ARI of `foo2`. Therefore, it reasonable to keep the ARIs of the subprograms in a stack.

• This stack is called **run time stack**, and for each activation of a subprogram `foo` (whether it is recursive or not), an instance of the activation record for `foo` is created and placed at the top of

this stack.

• Note that, since we now allow recursive subprograms, several activation record instances of the same subprogram may exist on the run time stack at a given time.

• When the subprogram terminates, the ARI for that activation of the subprogram is deallocated (popped) from the run time stack.

• A pointer named **Top** always shows the top most ARI on the run time stack.

• Now, let us see an example of how it works. We will first consider a relatively simple case where there are no nonlocal references.

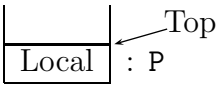
```
program MAIN;
  var P;
  procedure A(X : integer);
    var Y : boolean;
    procedure C (Q : boolean);
      begin { C }
       ... { position 3 }
      end; { C }
    begin { A }
      ... { position 2 }
      C ( Y );
      ...
    end; { A };
  procedure B (R : real);
    var S, T : integer;
    begin { B }
      ... { position 1 }
      A ( S );
      ...
    end; { B }
  begin { MAIN }
   ... { position 0 }
   B ( P );
   ...
  end. { MAIN }
```

[Note that the curly braces are used for comments]

• When this program is executed, `MAIN` will call `B`, then `B` will call `A`, then `A` will call `C`. The snapshots of the run time stack at certain time instances (when the program reaches to the positions 0, 1, 2

and 3) are given and explained below.

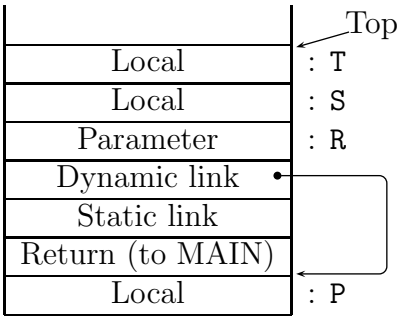
- When the program reaches to the position 0:



Initially, only the activation record of the `MAIN` is on the run time stack. The `Top` pointer therefore points to the border of the activation record of the `MAIN`.

Note that, we did not use a full activation record for the `MAIN` program. Some implementations prefer this way.

- When the program reaches to the position 1:

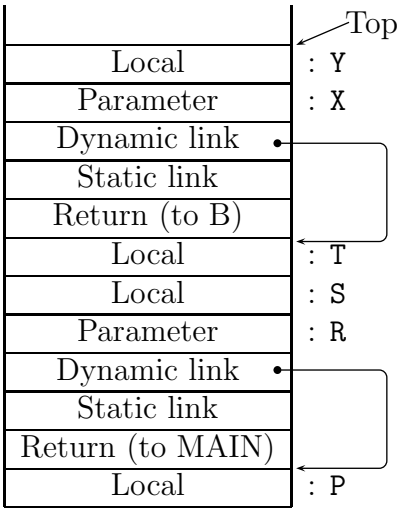


When the `MAIN` program calls the subprogram `B`, an activation record instance for `B` is created and placed on top of the run time stack. The `Top` pointer will be pointing to the top border of this activation record.

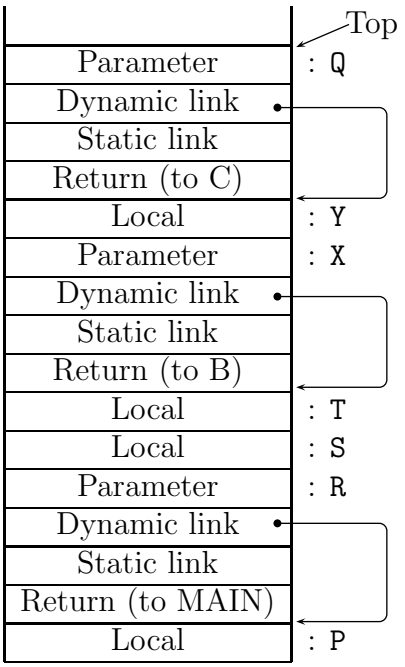
When the subprogram `B` terminates, the dynamic link field of its activation record will be used to reset the `Top` pointer back to the activation record of the program `MAIN`.

Note that, since the subprograms are not functions, the functional value field of a typical activation record given above is not used. Although the setting of static links are not given in this example, they are actually used, hence they are present in the example.

- When the program reaches to the position 2:



- When the program reaches to the position 3:



- The collection of the dynamic links in the run time stack is called the **dynamic chain** or **call chain**. It represents the dynamic execution history of the program.

- Assuming that the size of the local variables are fixed at compile time, the local references can be represented as offsets from the beginning of the activation record instance (where the Top pointer points when that activation record is at the top of the runtime stack) by calculating the size (based on the type of the local variable) and declaration order. Such an offset is called a **local offset**.

- For simplicity assume that each item in the activation record requires only one place. In this case, the local variables `T`, `S` and `R` of the subprogram `B` have the local offset 0, 1 and 2, respectively. The dynamic link of the procedure `B` is at local offset 3.

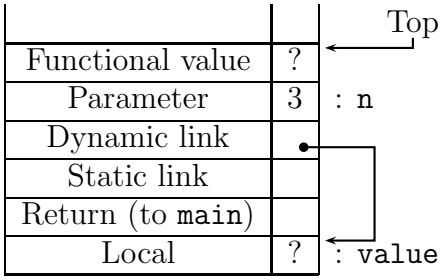
- All the fields of an activation record can also be accessed by using the local offsets.

- Let us now see another, more complicated example. This time we consider recursion, but we still do not have any non-local references:

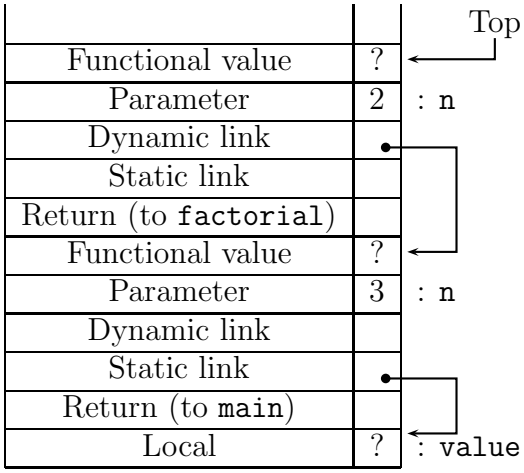
```
int factorial (int n) {
 // position 1
 if (n <= 1)
    return 1;
 return (n * factorial(n-1));
 // position 2
 }

void main () {
  int value = factorial(3);
  // position 3
  }
```

Position 2 represents the moment that the function has returned but the activation record of that activation has not been deallocated yet.

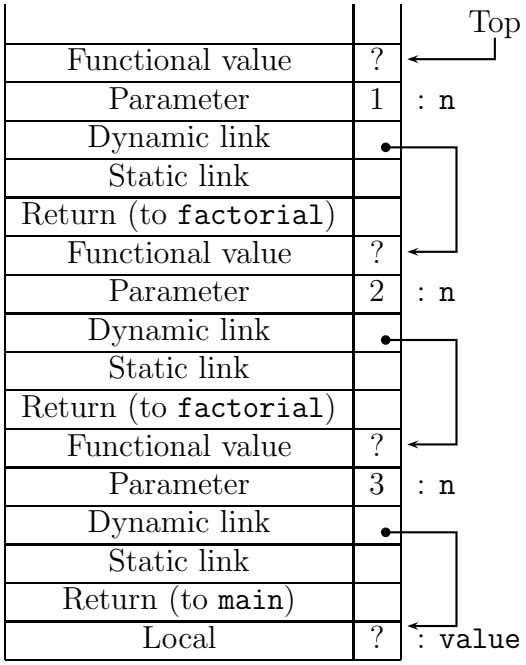
- The first time position 1 is reached:



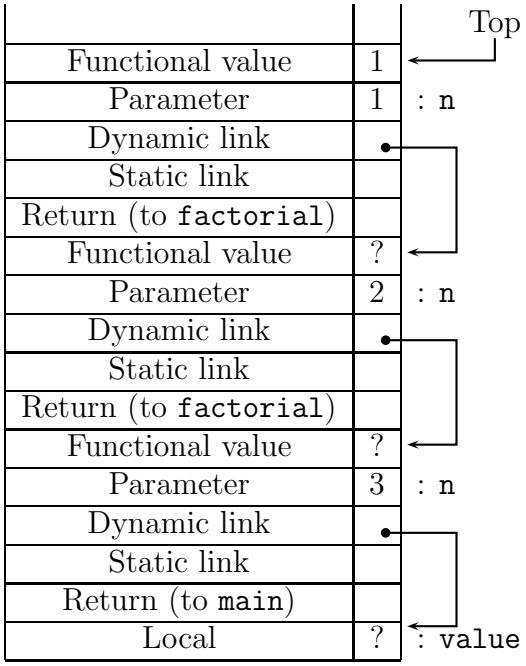
- The second time position 1 is reached:



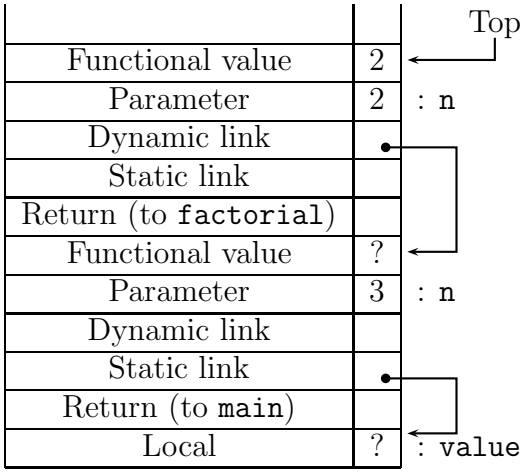
- The third time position 1 is reached:



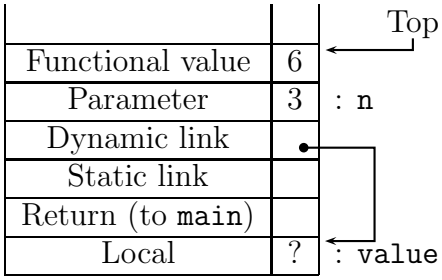
- The first time position 2 is reached:



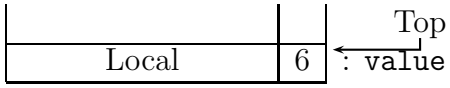
- The second time position 2 is reached:



- The third time position 2 is reached:



- The position 3 is reached:



8.4 Handling nonlocal references in statically scoped languages

- Recall that, in a statically scoped language, a nonlocal variable referenced in a subprogram, must be a local variable of one of the enclosing scopes (ancestor scopes) of that subprogram.

- Since a subprogram can only be called by its static parent (or by its siblings in its static parent), all the static ancestors of a subprogram must also be active (hence at least one ARI of them must be present on the run time stack).

- Therefore, the variable which is nonlocally referenced in a subprogram must be bound to storage somewhere in the run time stack. The problem is to find that place.

- Note that if the ARI in which the variable is bound to a storage is known, the local offset of the variable can be used to find the correct memory location.

- The actual problem is to find the correct ARI, since there can be more than one ARIs of the

same subprogram.

• Therefore, finding the actual memory location for a nonlocal reference is a two step procedure : find the correct ARI, then use the local offset within that ARI.

• Note that, if a subprogram could only be called by its static parent, then the ARI of its static parent would always be adjacent to the ARI of the subprogram. Hence, we could simply follow the "dynamic link" fields to reach the ARI of the static parents.

• However, a subprogram may also be called by the sibling subprograms which are declared directly within the (same) static parent. Hence the ARI of a subprogram is not necessarily adjacent to the ARI of a subprogram in the run time stack.

• There are two main approaches to the problem of finding the correct ARI: static chains and displays.

8.4.1 Static chains

• The collection of static links of the ARIs in the run time stack is called **the static chain**.

• During the execution of a subprogram `P`, the static link field of the ARI of `P` is set to point to the ARI of the most recent activation of the static parent of `P`.

• Hence, if there is a nonlocal reference in `P`, the static link field can be followed to reach the ARI of the static parent of `P`.

• However, not all nonlocal references in a subprogram are directly in the static parent of `P`. A nonlocal reference in `P` may actually be in static grand parent or great grand parent of `P`. Therefore, we may need to follow the static links to reach these grand parents, or to reach to great grand parents, etc. if we fail to find the variable referenced in the static parent and static grand parent of `P`.

• In fact, the compilers can understand that a reference is nonlocal, and they can also calculate the number of static links that has to be followed, since the nesting of subprograms is also a static information.

• Let the **static depth** of a scope be the number of enclosing scopes of it.

```
program MAIN;
  procedure A;
    procedure B;
    ...
    end; { B }
    ...
  end; { A };
  procedure C;
   ...
  end; { C }
  ...
end. { MAIN }
```

In the example above the static depth of `MAIN` is 0 since it has no enclosing scope (its the outermost scope). The static depth of `B` is 2 since it has two enclosing scopes, which are `A` and `MAIN`. The nesting depths of `A` and `C` are both 1, since they are only enclosed by `MAIN`.

• If there is a nonlocal reference in `B`, and if this nonlocal reference is for a variable declared in `MAIN`, then we have to follow 2 static links, one for reaching to the ARI of `A` from the ARI of `B`, and one for reaching to the ARI of `MAIN` from the ARI of `A`.

• In fact this number 2 can be calculated as :

"static depth of `B`" - "static depth of `MAIN`"

or in general, the number of static links to be followed can be calculated as

chain offset = "static depth of the scope in which the nonlocal reference takes place" - "static depth of the scope in which the variable is actually declared"

• Recall that a local reference is accessed by its local offset. Similarly, a nonlocal reference is accessed by (chain offset, local offset).

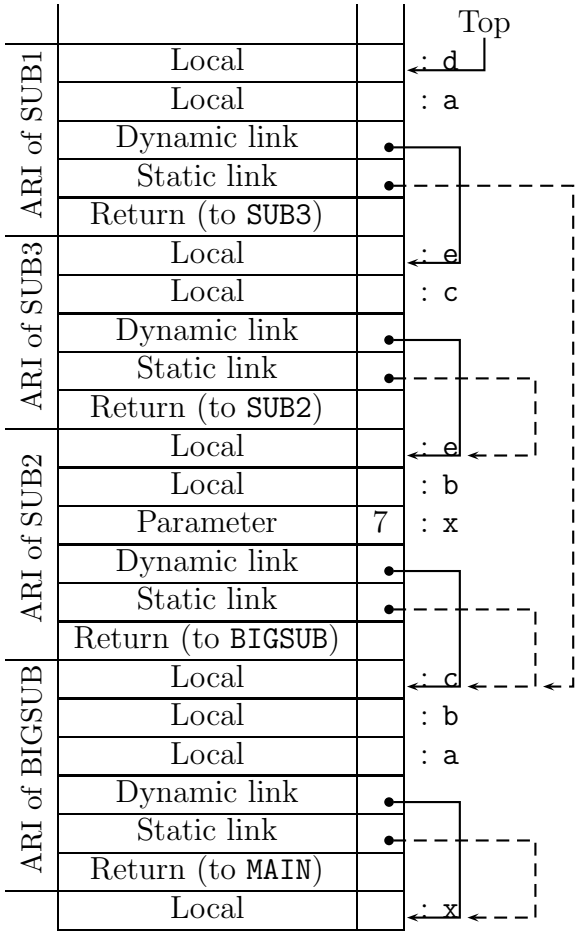
• When using this ordered pair of offsets, the local references can be represented by pairs of the form

(0, local offset)

- Consider the following example:

```
program MAIN;
  var x : integer;
  procedure BIGSUB;
    var a,b,c : integer;
    procedure SUB1;
      var a,d : integer;
      begin { SUB1 }
        a = b + c; { position 1 }
        ...
      end; { SUB1 }
    procedure SUB2 ( x : integer );
      var b,e : integer;
      procedure SUB3;
        var c,e : integer;
        begin { SUB3 }
         ...
         SUB1;
         ...
         e := b + a; { position 2 }
         end; { SUB3 }
      begin { SUB2 }
        ...
        SUB3;
        ...
        a := d + e; { position 3 }
      end; { SUB2 }
    begin { BIGSUB }
      ...
      SUB2(7);
      ...
    end; { BIGSUB }
  begin { MAIN }
    ...
    BIGSUB;
  end. { MAIN }
```

- When position 1 is reached, the run time stack has the following structure:



- The positions, references to the variables and the access pairs are given in the following table

	a	b	c	d	e
Position 1	(0,1)	(1,1)	?	–	–
Position 2	(2,2)	?			(0, 0)
Position 3	?	–	–	? (error)	?

- Now let us see how the static link fields of ARIs are assigned.

- When a subprogram is called and an ARI is created for it, the static link field of the ARI must be set to the ARI of its static parent.

- We are guaranteed that there will be at least one ARI of its static parent in the run time stack. However, the problem is that there can be more than one ARI of its static parent in the run time stack.

- Naturally, the static link field must be pointing to the most recent ARI of its static parent.

- One way of finding this ARI is to follow the dynamic chain until we reach to an ARI that belongs to the static parent. However, the run time effort required by this approach can be reduced by some

compile time effort.

• Note that, the static parent of the called subprogram must also be a static ancestor of the calling subprogram.

• The compiler can find the static parent of the called subprogram, and calculate the static depth of the calling subprogram and the static parent of the called subprogram.

• Let the static depth of the calling subprogram be *cp* and the static depth of the static parent of the called subprogram be *sp*.

• Since the ARI of the calling subprogram is the top most ARI on the run time stack at the time of the call, if one follows $cp - sp$ static links from the ARI of the calling subprogram, the ARI of the most recent activation of the static parent of the called subprogram would be reached.

• One approach is to pass the information $cp - sp$ as a parameter to the called subprogram (as if it is declared by the programmer).

• When the called subprogram starts its execution, before executing the actual subprogram body, it can first reach the ARI of its caller (by following the dynamic link field), then from that ARI it can follow $cp - sp$ static links to reach to the most recent ARI of its static parent, and set its static link field.

• The disadvantage of the static chain method is the following. The number of links that has to be followed in order to access a nonlocal reference is directly related to the difference of the static depths of the scopes in which the variable is referenced and declared. If, for example, there are a lot of variables declared within the main procedure and they are referenced in deeply nested subprograms, a lot of execution time will be spent while following these links.

• Another method is presented in the next section which provides a constant access time for each nonlocal reference, regardless of the nesting depths.

8.4.2 Displays

• Note that, when a nonlocal reference is seen in a subprogram, the actual declaration of that object must be in one of the static ancestors of that subprogram.

• Also note that, all the static ancestors of a subprogram are at different static depths.

• Assume that we keep an array named `display`, where `display[i]` is set to point to the most recent activation record of a subprogram whose static depth is `i`.

• Suppose that a subprogram whose static depth is `i` is called at run time. Then, as soon as its ARI is placed on the run time stack, `display[i]` is set to point to the ARI of this newly called

subprogram.

- For each nonlocal reference, the compiler can also calculate the static depth of the scope in which the variable is declared. Let us call this as the **display offset** of the nonlocal reference.

- Then at run time, when a nonlocal reference is required, we can access the correct ARI by

```
display[display offset of the nonlocal reference]
```

- After reaching to the correct ARI, we use the local offset again to the actual memory location within the ARI.

- This approach allows us to use only two steps to access to the actual variable, regardless of the number of scope levels between the scopes of the nonlocal reference, and the actual declaration.

- Hence, in this approach a nonlocal reference is characterized by an offset pair of the form

(display offset, local offset)

- The `display` array is maintained in the following way:
  - When a subprogram at static depth `i` is called (ARI for it must be created and placed on top of the run time stack), the static link field of its ARI is set to `display[i]`, and `display[i]` is set to point to this new ARI.
  - When a subprogram at static depth `i` terminates, `display[i]` is set to the value stored in the static link field of the ARI of the terminating subprogram.

8.5 Handling nonlocal references in dynamically scoped languages

- Note that, static links in ARIs are used to access the static parents. Since, in a dynamically scoped language, we do not need to access the static parents, the ARIs will not have static link fields.

- If there is a nonlocal variable reference, the actual variable referenced would be the one in the closest dynamic parent.

- There are two approaches for resolving nonlocal references in a dynamically scoped languages.

8.5.1 Deep access

- Note that, ARI of dynamic parent of a procedure can be accessed by following the dynamic link.

- In this approach, ARI of the subprograms also include descriptors of variables.

- Hence one can follow the dynamic chain to access the dynamic parent, dynamic grand parent, dynamic grand–grand parent, etc. and look for a variable declaration in these ARIs until a matching one is found.

8.5.2 Shallow access

- In this approach, the memory for the variables are not reserved in ARIs. Instead, every variable has a separate stack.

- When a new subprogram is activated, its ARI is created as usual (but without the fields for the variables in it).

- In addition to this, for each variable of the newly activated subprogram, a new item is placed on top of the corresponding variable stack.

- When a (local or nonlocal) variable reference is seen, the memory place for this variable is simply the top position of the corresponding place.

- When a subprogram terminates, its ARI is popped from the run time stack, and for each variable of the subprogram, one item is popped from the corresponding variable stack.

Chapter 9

Functional Programming

A Programming Language is a tool that has profound influence on our thinking habits.

– Edsger Wybe Dijkstra (May 11, 1930 - August 6, 2002) –

- In imperative languages, programming is done by executing statements sequentially.
- In functional languages, programming is done by evaluating expressions recursively.
- Examples of functional programming languages : LISP, ML, Miranda, Haskell, **Scheme**
- We will use Scheme as a working example, and try to understand and get used to the functional way of programming.
- The basic building blocks of functional languages are naturally expressions.

9.1 Expressions

- The simplest forms of expressions are literals.
- Numeric literals : `2, -5, -3.5`
- String literals: `"this is a string"`
- Character literals: `#\a` (the character a), `#\space` (the space character)
- Boolean literals: `#t` (true), `#f` (false)
- The next simplest form of expressions are variable references.
- In Scheme, variables are represented by identifiers which are sequences of digits, letters and some

special characters.

- The identifiers can start with a digit, however there must be at least one nondigit character in the identifier.

- Scheme is more permissive than most of the other language in identifiers. The followings are valid identifiers for Scheme

```
x
x1
1x
x-y
x=y
x+y
2xy
+
```

- However an identifier cannot include a space or parenthesis (curly or normal).

- The value of a variable is simply referenced by giving the identifier of the variable (i.e. in the same way as you used to in C, C++).

- Going one step further, a more complex kind of expressions are the ones that we get by applying some operators to operands.

- `+` is the addition operator.

- We have stated above that `+` can be used as an identifier.

- In fact, `+` *is* an identifier in Scheme, and can be assigned to new values.

- Scheme, as most other functional languages, does not distinguish between normal values and procedure values. That is, a procedure is nothing but an ordinary value.

- Initially the value of the variable `+` is set to the addition operator (procedure).

9.2 Procedures and procedure calls

- In imperative languages, we have made a clear distinction between a procedure (a subprogram that does not return a value), and a function (a subprogram that returns a value).

- However, in functional languages there are no procedures in this sense. Every subprogram is a function.

- In Scheme, the subprograms are called procedures, but one has to understand that they are all

functions that evaluate to, or return a value.

- The general syntax of a procedure call in Scheme is as follows:

$$\texttt{(procedure op}_1\ \texttt{op}_2\ \texttt{...}\ \ \texttt{op}_n\texttt{)}$$

- The operands (or actuals) $\texttt{op}_1\ \texttt{op}_2\ \texttt{...}\ \ \texttt{op}_n$ are expressions which are evaluated before the application of the procedure.

- The `procedure` is also considered to be an expression, which when evaluated, returns a procedure.

- For example in

```
(+ 1 2)
```

the procedure is a variable reference `+`, which as explained above initially bound to the value of the addition procedure. Hence, the addition procedure is applied to the values of the operands which are `1` and `2`.

- The operands can be subexpressions:

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

In this case, the value of the second operand `(* 2 3)` will be evaluated first, which is application of the multiplication operator to operands `2` and `3`, and then the value of the entire expression will be evaluated by applying the addition operator to the operands `1` and `6`.

- The procedure can also be a subexpression. Assume that we have a procedure `g` which returns the addition procedure when it is applied to `2`. Then in the expression

```
((g (- 10 8)) 3 5)
```

first the subexpression for the procedure call `(- 10 8)` would be evaluated, and it will obviously return 2 (if `-` is set to the subtraction operation). Then the procedure call `(g 2)` will return the addition procedure. Finally, the entire expression `((g (- 10 8)) 3 5)` would be reduced to `(+ 3 5)`. When this expression is evaluated (by applying the addition procedure to the operands `3` and `5`), the value 8 would be computed.

- Most of the operations in Scheme can be expressed as procedure calls.

- For those that cannot, there are some special forms.

- These special forms syntactically looks exactly the same as a procedure call. In order not to get confused, we have to be careful, and remember these special forms.

- These special forms are explained in the next few sections.

9.3 Binding Variables to Values

- Assigning a value to a variable can be performed by using such a special form.

- The general syntax of the statement that can be used to bind a value to a variable (i.e. the assignment statement of Scheme) is as follows:

```
(define variable expression)
```

  Here `variable` is an identifier, and `expression` is a general expression.

- For example, in order to assign a variable `x` to value 5, the following expression should be evaluated.

```
(define x 5)
```

- Note that the syntax of the `define` expression is not different than the general form of a procedure call.

- Therefore one might easily get confused and think that, with `(define x 5)` we are applying the value of `x` (i.e. r–value of `x`) and 5 to procedure `define`.

- However, since `define` is a special form, the semantics is different than the usual procedure application.

- The evaluation of `(define x 5)` comprises the following steps: Get the l–value of `x`, and assign the value 5 to the memory location denoted by the l–value of `x`.

- Or by using the general syntax of `define`, i.e. `(define variable expression)`, first the `expression` is evaluated, then this value is assigned as the value of the `variable`.

- For example

```
(define x12 (+ 1 (* 2 3)))
```

assigns the value 7 to the variable `x12`.

9.4 Scheme Programs and Read–Eval–Print Loop

- Writing programs in Scheme (or in any other functional language) may feel awkward at the beginning for someone who is used to programming in imperative languages.

- However, after you exercise a little, you'll see that it is not hard. I personally believe that, at least at an introductory level, programming in a functional language should be easier than programming

in an imperative language, as we are all used to functions and functional way of thinking, from our math courses that we have taken starting from the elementary school years.

• However, since we are all now oriented towards imperative way of thinking when writing programs, we will need to do some exercise to get you oriented towards functional way of programming.

• We will use MIT Scheme interpreter while doing these exercises.

• Although there exists some compilers for Scheme, MIT Scheme interpreter is the most famous Scheme implementation.

• Note that, this is an interpreter. Hence you will not get your Scheme programs converted into some other representation. The interpreter will execute your programs directly.

• The MIT Scheme interpreter is installed on our UNIX systems (e.g. `flow.sabanciuniv.edu`) and can be started by simply typing `scheme` at the command prompt.

• There is also a MIT Scheme implementation for Windows. The setup file is available on SU-Course.

• A few remarks about the MIT Scheme interpreter:
– You can exit from the interpreter by typing `^D` (i.e. control–D) in UNIX, and by typing `(exit)`.
– The interpreter will give you the prompt `1 ]=>`
– When you see this prompt, it means that the interpreter is waiting for you to type–in some Scheme expression.
– When you type a correct expression, the interpreter will evaluate this expression, and print out the value of the expression. Then it will again give you the prompt `1 ]=>` in order for you to type in the next expression that you want to evaluate.
– The semicolon character "`;`" can be used as the line comment (i.e. anything that comes after `;` on a line is considered to be comment). Although these comments can be considered as useless while using the interpreter interactively, they will be helpful when writing and storing Scheme programs out side the interpreter, and loading them into the interpreter later.
– If anything goes wrong, instead of the prompt "`1 ]=>`", you will see an error prompt in the form "`2 error`". In order to get out from this error prompt, and go back to normal prompt, you need to type in

On UNIX: `^C^C` (i.e. control–C twice)
On Windows : `^C^G` (i.e. control–C and control–G), or `(RESTART 1)`

• Note that, we have outlined the normal execution of the MIT Scheme interpreter above as:
(i) Read an expression (READ step)
(ii) Evaluate the expression (EVAL step)
(iii) Printout the value of the expression (PRINT step)
repeat these steps forever (until the user breaks out from the interpreter). This behaviour of the main loop of the interpreter is known as the READ–EVAL–PRINT–LOOP, or in other words as

REPL.

• Now let us see our first interaction with the Scheme interpreter. Note that, what you see next to the prompt "`1 ]=>`" is what we write as the input. The next line after this prompt is the response of the interpreter. For example, at the very first line, we enter "`3 ; just a simple value`", and the interpreter responds with "`;Value: 3`".

Scheme interaction

```
1 ]=> 3 ; just a simple value
;Value: 3

1 ]=> * ; another simple value (* is initially bound to multiplication procedure)
;Value 1: #[compiled-procedure 1 ("arith" #xDB) #x3 #x44A3F4]

1 ]=> (* 2 3) ; our first procedure application: multiply 2 and 3
;Value: 6

1 ]=> (define x 3) ; our first variable binding (evaluates to the variable symbol)
;Value: x

1 ]=> x ; our first variable reference (x was bound to 3 by the previous define)
;Value: 3

1 ]=> x ; our first variable reference (x was bound to 3 by the previous define)
;Value: 3

1 ]=> (+ x (* 2 3)) ; multiply 2 and 3 and add the value of x to the result
;Value: 9

1 ]=> (define x (+ x 1)) ; add 1 to x and assign the result as the value of x
;Value: x

1 ]=> x ; x should now be 4
;Value: 4

1 ]=> ^D
End of input stream reached
Happy Happy Joy Joy.
```

• We can also write our programs using an ASCII editor and load the program into the interpreter. For example, let us assume that we an ASCII file named `s0.scm` with the following content:

s0.scm

```
(define x 3) ; define x to be 3
(define y 5) ; define y to be 5
(define z (+ x y)) ; define z to be the sum of x and y
```

- Assume that we start MIT Scheme interpeter and interact with it:

Scheme interaction

```
1 ]=> x ; initially x is not bound, so this should give an error

;Unbound variable: x
;To continue, call RESTART with an option number:
; (RESTART 3) => Specify a value to use instead of x.
; (RESTART 2) => Define x to a given value.
; (RESTART 1) => Return to read-eval-print level 1.


2 error> ^C^C

1 ]=> (load "s0.scm") ; load s0.scm (in which we have definitions for x,y,z)
; Loading "s0.scm" -- done
; Value: z

1 ]=> x
; Value: 3

1 ]=> y
; Value: 5

1 ]=> z
; Value: 8
```

Note that, before loading the file `s0.scm`, referring to `x` produced an error. This is expected since `x` has not been bound to a value yet. We would have get similar errors, if we had referred to `y` or `z`. By using `load`, we can load a file into the interpreter. This has the same effect as if we had written the entire content of the file in the interpreter bu using the keyboard. As you see from the interaction, the value of the load expression is the same as the value of the last expression given in the file.

9.5 Conditional Expressions

• Conditional expression is another special form. In other words, it syntactically looks like a procedure application, but it has a different semantics.

• The syntax of conditional expression is as follows:

```
(if test-exp then-exp else-exp)
```

• The semantics of condition expression is as follows:

1. First the `test-expr` is evaluated
2. If `test-expr` evaluates to false (i.e. `#f`), then the `else-expr` is evaluated and the value of the `else-expr` is taken as the value of the conditional expression.
3. If `test-expr` does not evaluate to false, then the `then-expr` is evaluated and the value of the `then-expr` is takes as the value of the conditional expression.

• Note that, by this semantics any value other than "false" is considered to be true. For example, the following conditional expressions, except the last one, would all evaluate to 1 (i.e. the value of the `then-expr`. The last one evaluates to 2 (i.e. the value of the `else-expr`).

```
(if #t 1 2)
(if 5 1 2)
(if 0 1 2)
(if -1 1 2)
(if #f 1 2)
```

• Note that, a short circuit evaluation is used while evaluating conditional expressions. In other words, we do not evaluate all three expressions (`test-expr`, `then-expr`, `else-expr`) within a conditional expression, and then decide the value of the conditional expression.

• The way a conditional expression is evaluated guarantees that exactly 2 expressions are evaluated for the evaluation of a conditional expression. This semantics not only increases the speed of the evaluation of conditional expressions, but also provides a programming trick to avoid certain errors.

• For example, if one has to evaluate the expression `(/ x y)` (i.e. divide `x` by `y`), such an expression may produce a run time error if `y` is equal to 0. Therefore, as a safe guard, such an expression can be written as

```
(if (zero?  y) 0 (/ x y))
```

• `zero?` is a predefined procedure in Scheme, and checks if the value of its argument is 0 or not. By using such an expression, we guarantee that `(/ x y)` is evaluated only if `y` is different than 0, hence avoid a run time error.

• Note that within the identifier of zero testing procedure `zero?`, the question mark is used. We have mentioned before that in Scheme identifiers can include special characters like `?`, `+`, etc. which are normally not allowed in other languages within the identifiers.

• `zero?` is a procedure that evaluates to true or false. Such procedures are called *predicates*. In Scheme, the identifiers of predefined predicates (we will see examples of these) all end with a question mark, to emphasize that they are predicates.

• It is also a good programming pratice to use identifiers ending with a question mark for the predicate procedures that you will define.

9.6 Data types

• Scheme implementations may vary in the data types they support and the operations on these data types.

• We will introduce the widely used data types in Scheme by examples.

• Scheme uses dynamic type checking.

9.6.1 Boolean

• We have introduced the literals of the boolean data type before : `#t` and `#f` corresponds to the boolean literals true and false, respectively.

• The type predicate for a type is a predicate that evaluates to true if and only if the value provided belongs to that type.

• For example, for the boolean data type there is a predefined type predicate `boolean?`.

Scheme interaction

```
1 =]> (boolean? #f)
; Value: #t

1 =]> (define true #t)
; Value: true

1 =]> (boolean? true)
; Value: #t

1 =]> (boolean? 1)
; Value: #f
```

- The usual boolean operators `and`, `or`, `not` are also available.
- Equality of two boolean values can be checked by using `eq?`, i.e. `eq?` is the equality predicate.

Scheme interaction

```
1 =]> (and #t #t)
; Value: #t

1 =]> (and #t #f)
; Value: #f

1 =]> (or #t #f)
; Value: #t

1 =]> (not #f)
; Value: #t

1 =]> (eq? #f #t)
; Value: #f

1 =]> (define x #t)
; Value: x

1 =]> (define y #f)
; Value: y

1 =]> (eq? x y)
; Value: #f
```

9.6.2 Numbers

- The literals for the numbers data types are the usual numbers: `0`, `5`, `-3`, `25.31`, etc.
- The type predicate is `number?`
- The usual arithmetic operators (`+`, `-`, `*`, `/`) are available.
- `eq?` can be used as the equality predicate. `=` can also be used for equality checking.
- Comparison procedures (`<`, `<=`, `>`, `>=`).

9.6.3 Integers

- Subtype of number data type
- type predicate: `integer?`

Scheme interaction

```
1 ]=> (integer? 1)
;Value: #t

1 ]=> (integer? 1.0)
;Value: #t

1 ]=> (integer? 1.5)
;Value: #f
```

- The type supports the usual arithmetic operators on integers, and comparisons between the integers.
- `eq?` can be used as the equality predicate. `=` can also be used for equality checking.

9.6.4 Characters

- Characters that are visible when printed are represented by literals by preceding the character with `#\`
- For example:

`#\a` → the literal value for the character "a"
`#\%` → the literal value for the character "%"

- Some non–printable characters also have literal representations:

  `#\space` → the literal value for the blank character
  `#\newline` → the literal value for the newline character

- type predicate: `char?`

- equality and order predicates: `eq?`, `char=?`, `char<?`

- some other procedures for the character data type:

```
char->integer
char-alphabetic?
char-numeric?
char-whitespace?
```

9.6.5 Strings

- The literals are denoted by a sequence of characters within double quotes:

```
"this is a string literal"
```

- Type predicate: `string?`

Scheme interaction

```
1 ]=> (define s "This is a")
;Value: s

1 ]=> (define ss (string-append s " longer string"))
;Value: ss

1 ]=> (string? s)
;Value: #t

1 ]=> (string-length s)
;Value: 9

1 ]=> (string-length ss)
;Value: 23

1 ]=> (string-ref s 0)
;Value: #\T

1 ]=> (string-ref ss 3)
;Value: #\s

1 ]=> (string #\a #\b)
;Value: "ab"
```

9.6.6 Symbols

• When an identifier referenced in the Scheme interpreter, the current value bound to the identifier is returned.

Scheme interaction

```
1 ]=> (define x12 5)
;Value: x12

1 ]=> x12
;Value: 5

1 ]=> (+ x12 3)
;Value: 8
```

• Within the second and the third expressions evaluated above, the occurrences of `x12` are refer-

ences to the value of `x12`.

- When an identifier is quoted, this quoted occurrence of the identifier is not understood as a normal variable reference by the Scheme interpreter. Instead, the identifier itself is considered to be a value of symbol type.

Scheme interaction

```
1 ]=> (quote x12)
;Value: x12
```

- The occurrence of `x12` above will not be taken as a variable reference. It is quoted. The expression `(quote x12)` evaluates to "symbol `x12`".

- Quoting used very frequently. Therefore a shorter syntax is available for quoting:

Scheme interaction

```
1 ]=> (quote x12)
;Value: x12

1 ]=> 'x12
;Value: x12
```

- Both of the expressions given above evaluate to the same value, i.e. to the symbol `x12`.

- Type predicate: `symbol?`

- Equality predicate: `eq?`

Scheme interaction

```
1 ]=> (define x 3)
;Value: x

1 ]=> (number? x)
;Value: #t

1 ]=> (symbol? x)
;Value: #f

1 ]=> (number? (quote x))
;Value: #f

1 ]=> (number? 'x)
;Value: #f

1 ]=> (symbol? 'x)
;Value: #t

1 ]=> (eq? 'x 'x)
;Value: #t

1 ]=> (eq? 'x 'y)
;Value: #f

1 ]=> (define y 'abc)
;Value: y

1 ]=> (eq? y 'abc)
;Value: #t

1 ]=> (eq? y 'y)
;Value: #f

1 ]=> (number? 5)
;Value: #t

1 ]=> (symbol? 5)
;Value: #f

1 ]=> (symbol? '5)
;Value: #f

1 ]=> (number? '5)
;Value: #t
```

9.6.7 Lists

- A list is an ordered sequence of elements, where each element may be of arbitrary type.

- A list is represented in Scheme by surrounding the representations of its elements by a pair of parenthesis.

- For example:

```
(a 3 #t)
```

This is a list of three elements. The first element is the symbol `a`, the second element is integer 3, and the third element is the boolean value true.

- Another example:

```
((a 3 #t) cs ())
```

This is again a list of 3 elements. The first element is actually a list. In fact the first element is the list we considered above. The second element is the symbol `cs`. The last element is another list, which actually is the empty list, or null list, or null object.

- Note that, the syntax for procedure application and the syntax for list denotation are the same.

- The context dictates whether it is a procedure application or a list.

- Quoting can be used to produce list values.

Scheme interaction

```
1]=> (a b c) ; This is a procedure application.
;Value: ...

1]=> (quote (a b c)) ; This is a list with 3 elements
;Value: (a b c)

1]=> '(a b c) ; This is a list with 3 elements
;Value: (a b c)
```

- The type predicate for lists is "`list?`"

- Other than quoting, there are several ways to create lists.

- For example, the procedure "`list`" can be used to create a list. It takes any number of arguments, the values of of the arguments are evaluated, and each value becomes an element of the list produced. The expression "`(list` $\texttt{exp}_1$ $\texttt{exp}_2$ ... $\texttt{exp}_n$`)`" produces the list "`(`$\texttt{val}_1$ $\texttt{val}_2$ ... $\texttt{val}_n$`)`" where $\texttt{val}_i$ is the value of the expression $\texttt{exp}_i$.

Scheme interaction

```
1]=> (list 1 2 3) ; each argument becomes an element of the list
;Value: (1 2 3)

1]=> (list 1 (+ 2 3)) ; note the variable number of arguments
;Value: (1 5)

1]=> (define x 3)
;Value: x

1]=> (define y 'a)
;Value: y

1]=> (list x y) ; if arguments are expressions, they are evaluated
;Value: (3 a)

1]=> (define l1 '())
;Value: l1

1]=> (define l2 '(a))
;Value: l2

1]=> (define l3 '((b)))
;Value: l3

1]=> (list l1 l2 l3 '(((c))))
;Value: (() (a) ((b)) (((c))))

1]=> (list) ; no arguments => empty list
;Value: ()
```

• Another list creation procedure is "`cons`". It always takes two arguments. The first argument is the first element of the list to be constructed, and the second argument is the rest of the list to be constructed. The expression "`(cons exp '(`v_1 v_2 ... v_n`))`" evaluates to the list "`(val `v_1 v_2 ... v_n`)`" where `val` is the value of the expression `exp`.

Scheme interaction

```
1]=> (cons 'a '(c d))
;Value: (a c d)

1]=> (list 'a '(c d))
;Value: (a (c d))

1]=> (cons '(a b) '(c d))
;Value: ((a b) c d)

1]=> (cons '() (c d))
;Value: (() c d)

1]=> (cons 'a '())
;Value: (a)

1]=> (cons '(a b) '())
;Value: ((a b))

1]=> (define l1 (list 'a))
;Value: l1

1]=> (define x 'b)
;Value: x

1]=> (cons x l1)
;Value: (b a)

1]=> (define l2 (cons '() (cons x l1)))
;Value: l2

1]=> l2
;Value: (() b a)
```

• There is an exception in the usage of the procedure `cons`. The second argument of the `cons` does not have to be a list. However, if it is not a list, then the value produced by `cons` is not a list either.

Scheme interaction

```
1]=> (cons 3 5)
;Value: (3 . 5)
```

This is called a pair (or dotted pair), and we will consider pairs soon.

- Another way of creating lists is to append two or more lists. Let $\texttt{l}_i$ be a list in the form

$$(\texttt{v}_1^i \ \texttt{v}_2^i \ \ldots \ \texttt{v}_{n_i}^i)$$

Then the expression `(append` $\texttt{l}_1$ $\texttt{l}_2$ `...` $\texttt{l}_m$`)` evaluates to the list

$$(\ \texttt{v}_1^1 \ \texttt{v}_2^1 \ \ldots \ \texttt{v}_{n_1}^1 \ \texttt{v}_1^2 \ \texttt{v}_2^2 \ \ldots \ \texttt{v}_{n_2}^2 \ \ldots \ \texttt{v}_1^m \ \texttt{v}_2^m \ \ldots \ \texttt{v}_{n_m}^m \ )$$

Scheme interaction

```
1]=> (append '(a b) '(c d))
;Value: (a b c d)

1]=> (append '() '(c d))
;Value: (c d)

1]=> (append '(a b) '())
;Value: (a b)
```

- We will also need to divide the lists. The easiest way to divide a list is to consider the first element of the list and the rest of the list.

- For historical reasons, the first element of a list is known as the "`car`" of the list, and the rest of the (with the first element removed) is known as the "`cdr`" (pronounced as "could-er") of the list.

> IBM 704 had a memory word structured as | Address Register | Decrement Register |.
> The **C**ontent of **A**ddress **R**egister (i.e. car) and the **C**ontent of **D**ecrement **R**egister (i.e. cdr) are used to keep the first element and the rest of the element of the lists.

- `car` and `cdr` are reverse of `cons`.

Scheme interaction

```
1]=> (car (cons 'a '(b c)))
;Value: a

1]=> (cdr (cons 'a '(b c)))
;Value: (b c)

1]=> (car '(a b c))
;Value: a

1]=> (cdr '(a b c))
;Value: (b c)

1]=> (car (cdr '(a b c)))
;Value: b

1]=> (cdr '(a))
;Value: ()

1]=> (car '())
2 error>

1]=> (cdr '())
2 error>
```

- Nested calls the `car` and `cdr` are used very frequently in Scheme programs. Therefore there is a shorthand notation defined to make such nested calls.

Scheme interaction

```
1]=> (cadr '(a b c))
;Value: b

1]=> (cddr '(a b c))
;Value: (c)

1]=> (caddar '((a b c) '(d e f)))
;Value: c
```

- Empty lists are represented by the same object called the "empty list" or "null" object.
- `null?` predicate tests if an object is the empty object or not.

Scheme interaction

```
1 ]=> (null? '())
;Value: #t

1 ]=> (define l '(a))
;Value: l

1 ]=> (null? l)
;Value: #f

1 ]=> (null? (cdr l))
;Value: #t
```

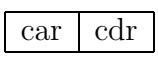
Note that, `null?` returns true only if the argument is the empty list. Otherwise, it returns false.

Scheme interaction

```
1 ]=> (null? #f)
;Value: #f

1 ]=> (null? 5)
;Value: #f
```

9.6.8 Pairs

- Most of the time we consider lists, at an abstract level, as a sequence of elements.
- However, it is necessary to understand how they are actually implemented.
- Non–empty lists are implemented as pairs, which are also known as *dotted pairs* or as *cons cells*.
- A pair is a structure with two fields



- The `cons` procedure creates a new pair with the car field set to the first, and the cdr field set to the second field.
- The procedures `car` and `cdr` access to the car and cdr field of pairs.
- The type predicate for pairs is `pair?`

Scheme interaction

```
1 ]=> (pair? (cons 'a '(b c)))
;Value: #t

1 ]=> (pair? '(a b c))
;Value: #t

1 ]=> (pair? (cons 'a 'b))
;Value: #t

1 ]=> (pair? 'a)
;Value: ()
```

- Note that, the empty list (i.e. the null object) is not a pair

Scheme interaction

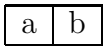
```
1 ]=> (pair? '())
;Value: ()
```

- A pair with its car field set to symbol `a` and cdr field set to symbol `b` is shown as

`(a . b)`

(hence the name **dotted pair**)

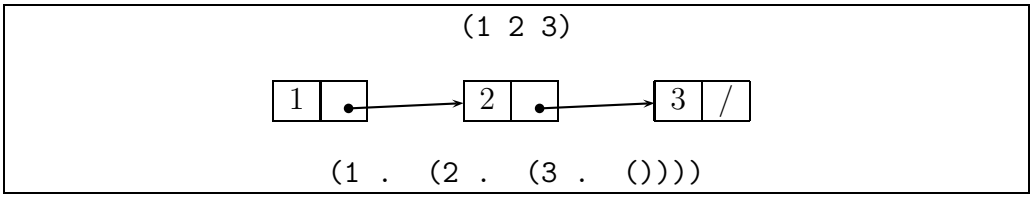
- It is also shown by using a **box diagram** as given below

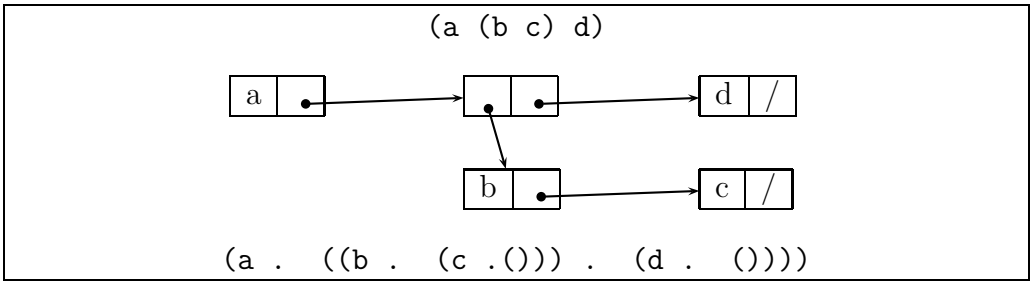


- If a field (either car or cdr) of a pair is set to null, we show it by using a `/` in box diagrams, and a null object in dotted pair notation.

`( a . () )`

- Note that a list can be considered as a chain of pairs. In fact, this is how a list actually implemented:



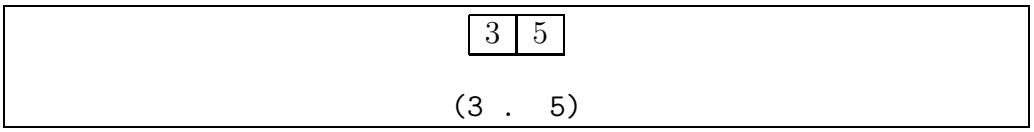


- Recall that the second argument of `cons` does not have to be a list.

Scheme interaction

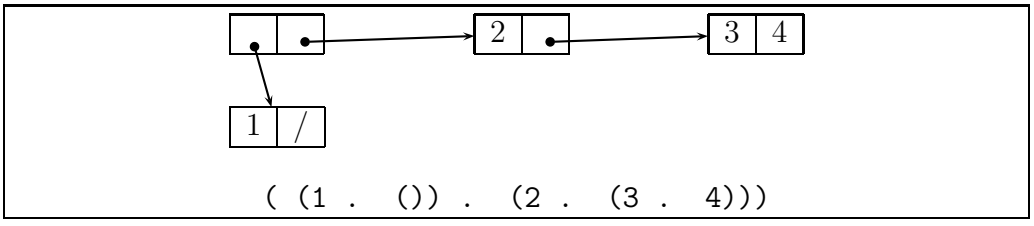
```
1 ]=> (list? (cons 3 5))
;Value: ()

1 ]=> (pair? (cons 3 5))
;Value: #t
```

- The cons-cell structure that will be created by the expression `(cons 3 5)` is given below by using both the box diagram notation, and by using the dotted pair notation:



- Every non–empty list is a pair. However, not every pair is a list.

- Pairs whose cdr–link chain do not end with an empty list (NULL object) are not lists. For example, the following is not a list:



- `eq?` can be used to test the equivalence of two objects. The behaviour of `eq?` procedure may feel strange, if the implementation details are not known.

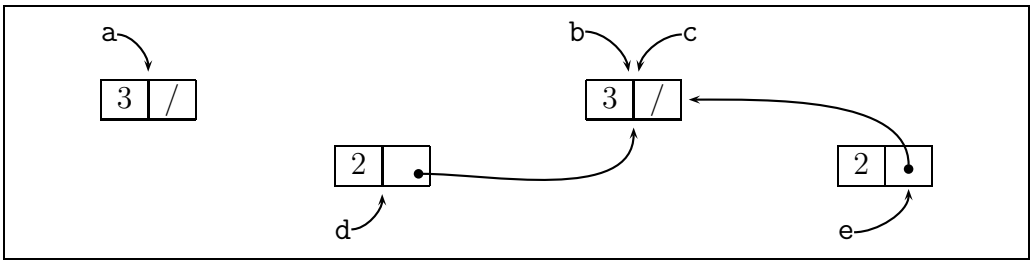
- For simple typed objects like, symbols, booleans, characters, etc., `eq?` returns true iff the objects have the same written representation.

- However, for complex objects like pairs, lists, etc. `eq?` returns true iff the two objects are the same object. If the objects are different objects, `eq?` would return false even if they print out the same expression.

Scheme interaction

```
1 ]=> (define a (cons 3 '()))
;Value: a

1 ]=> (define b (cons 3 '()))
;Value: b

1 ]=> a
;Value: (3)

1 ]=> b
;Value: (3)

1 ]=> (eq? a a)
;Value: #t

1 ]=> (eq? a b)
;Value: ()

1 ]=> (define c b)
;Value: c

1 ]=> (eq? c b)
;Value: #t

1 ]=> (define d (cons 2 b))
;Value: d

1 ]=> (define e (cons 2 c))
;Value: e

1 ]=> (eq? d e)
;Value: ()

1 ]=> (eq? (cdr d) (cdr e))
;Value: #t
```

- The interaction given above would create the following objects in the memory:



- Note that, each evaluation of `cons` expression creates a new cons–cell in the memory.

9.6.9 Vectors

- The vector data type is similar to the list data type in that a vector consists of a sequence of heterogenous values.
- However unlike lists, vector provides random access to its elements.
- `vector` is a procedure that can be used to produce vector values.
- `vector?` is the type predicate

Scheme interaction

```
1 ]=> (vector 1 (+ 1 2))
;Value: #(1 3)

1 ]=> (define v0 #(1 b)) ; # is the quote that generates vector values
;Value: v0

1 ]=> (define v1 (vector 'a (+ 1 2)))
;Value: v1

1 ]=> (define v2 #(b (+ 1 2)))
;Value: v2

1 ]=> (define v3 (vector v1 v2))
;Value: v3

1 ]=> (define v4 #(v1 v2))
;Value: v4

1 ]=> v3
;Value: #( #(a 3) #(b (+ 1 2)))

1 ]=> v4
;Value: #(v1 v2)

1 ]=> (vector-length v3)
;Value: 2

1 ]=> (vector-ref v3 1); 0 based indexing
;Value: #(b (+ 1 2))

1 ]=> (vector->list v3)
;Value: ( #(a 3) #(b (+ 1 2)))

1 ]=> (vector? v3)
;Value: #t

1 ]=> (vector? (vector-ref v3 1)
;Value: #t

1 ]=> (vector? (vector-ref (vector-ref v3 1) 1))
;Value: #f
```

9.6.10 Procedures

• Note that we are still talking about a data type. In functional programming languages in general, and in Scheme, procedure values are just like any other value.

• The type predicate for the procedure data type is `procedure?`

Scheme interaction

```
1 ]=> (procedure? 'car)
;Value: ()

1 ]=> (procedure? car)
;Value: #t

1 ]=> (procedure? (car (list cdr)))
;Value: #t

1 ]=> ((if (procedure? 3) car cdr) '(x y z))
;Value: (y z)

1 ]=> (((if (procedure? procedure?) cdr car) (cons car cdr)) '(car in the car))
;Value: (in the car)
```

• A special expression called "*lambda expression*" is used to generate procedure values. Here is the syntax for a lambda expression:

`(lambda (`*formals*`)` *body*`)`

• Here "`lambda`" is a keyword that indicates that this is a procedure value.

• "*formals*" will be the formal parameters of the procedure. It will be a list of parameter names in the concrete syntax.

• "*body*" is the body of the procedure. It will be an expression in the concrete syntax. The expression will possibly refer to the formal parameters of the procedure.

• When the procedure is applied on a list of values (i.e. when the procedure is called by providing a list of actual parameters), the body will be evaluated and the result of this evaluation will be considered (returned) as the value of the procedure application.

• Let us now go over a couple of very simple examples:

```
(lambda (n) (+ n 1))
```

- This expression, being a lambda expression, evaluates to a procedure value.

Scheme interaction

```
1 ]=> (procedure? (lambda (n) (+ n 1)))
;Value: #t

1 ]=> (lambda (n) (+ n 1))
;Value: #[compound-procedure]
```

- It actually evaluates to a procedure which would return 1 + its input parameter when called.

- Since it evaluates to a procedure value, we can actually use this value in our usual procedure application syntax.

Scheme interaction

```
1 ]=> ((lambda (n) (+ n 1)) 5)
;Value: 6
```

- This applies the procedure value onto the actual parameter 5. In other words, the procedure is called with argument 5. As expected, the body of the expression "`(+ n 1)`" is evaluated (by taking `n` to be 5) and the procedure application evaluates to 6.

- Since a procedure value is just like any other value we can bind a procedure value as the value of a variable or have a procedure value as a list element, or a vector element. Here are some examples on this.

Scheme interaction

```
1 ]=> (define foo1 (lambda (n) (+ n 1)))
;Value: foo1

1 ]=> (define foo2 (lambda () 5))
;Value: foo2

1 ]=> foo1
;Value: #[compound-procedure]

1 ]=> (foo1 5)
;Value: 6

1 ]=> foo2
;Value: #[compound-procedure]

1 ]=> (foo2)
;Value: 5

1 ]=> (foo1 (foo2))
;Value: 6

1 ]=> (foo1 ((lambda () 5)))
;Value: 6

1 ]=> (define lst (list foo1 foo2))
;Value: lst

1 ]=> lst
;Value: (#[compound-procedure] #[compound-procedure])

1 ]=> ((car lst) ((cadr lst)))
;Value: 6

1 ]=> (foo2 12)
;The procedure #[compound-procedure foo2] has been called with 1 argument
;it requires exactly 0 arguments.

2 error>
```

- As can bee seen above, when a procedure is applied to a list of actual parameters, the number of actual parameters provided must be exactly the same as the number of formal parameters.

- It is possible to create procedure values which take more than one parameter.

Scheme interaction

```
1 ]=> (define foo3
         (lambda (b lst)
           (if b
               (car lst)
               (cadr lst)
           )
         )
       )
;Value: foo3

1 ]=> (foo3 #f '(a b))
;Value: b

1 ]=> ((foo3 #t (list cdr car)) '(a b c))
;Value: (b c)
```

• It is also possible to create procedure that takes any number of parameters by using the following syntax:

(`lambda` *fpar body*)

Note that we do not have the paranthesis around the formal paramaters and there is only one formal parameter. Within the body of the procedure, the formal parameter should be interpreted as a list. Here are some examples:

Scheme interaction

```
1 ]=> (define foo4 (lambda lst lst)) ; foo4 can take any number of parameters
;Value: foo4

1 ]=> (foo4 1)
;Value: (1)

1 ]=> (foo4 1 2 3)
;Value: (1 2 3)

1 ]=> (foo4)
;Value: ()

1 ]=> (define foo5 (lambda (lst) (if (null? lst) 0 (+ (car lst) (foo5 (cdr lst))))))
;Value: foo5

1 ]=> (define foo6 (lambda lst (foo5 lst)); foo6 can take any number of parameters
;Value: foo6

1 ]=> (foo5 (foo4 1 2 3)) ; foo5 needs a single paramater which is a list
;Value: 6

1 ]=> (foo6 1 2 3) ; foo6 applied on 3 parameters
;Value: 6

1 ]=> (foo6 1 2 3 4 5) ; foo6 applied on 5 parameters
;Value: 15
```

- `foo4` above is essentially the same procedure as the built–in "`list`" procedure of Scheme.

- Don't forget that procedures are first class values and they can be used as any other value. We can define a procedure returning a procedure value.

Scheme interaction

```
1 ]=> (define add2 (lambda (n) (+ n 2)))
;Value: add2

1 ]=> (define compose
               (lambda (f g)
                  (lambda (x) (f (g x))))) ; a procedure returning a procedure value
;Value: compose

1 ]=> (define add4 (compose add2 add2))
;Value: add4

1 ]=> (add4 7)
;Value: 11
```

- As we know, procedures are applied by using the procedure application syntax (e.g. `(+ 1 2)`).
- Another way applying a procedure is to use a predefined procedure named "`apply`".
- The general syntax is

(`apply` *proc_exp list_of_operands*)

where *proc_exp* is an expression that must evaluate to a procedure value and *list_of_operands* is a list of operands (which can be in the form a list of expressions). The procedure `apply`, applies the procedure given by the *proc_exp* to the list of operands given by the *list_of_operands*.

- For example

```
(apply + '(1 2 3))
```

is equivalent to

```
(+ 1 2 3)
```

Scheme interaction

```
1 ]=> (define abc '(a b c))
;Value: abc

1 ]=> (apply cons (cdr abc))
;Value: (b . c)

1 ]=> (apply apply (list procedure? (list apply)))
;Value: #t
```

- Yet another way of applying a procedure is to use "`map`". The syntax is as follows:

$$\texttt{(map } \textit{proc\_exp } l_1 \ l_2 \ \ldots \ l_n\texttt{)}$$

where *proc_exp* is again an expression evaluating to a procedure value and l_1, l_2, ..., l_n are all lists. *proc_exp* must evaluate to a procedure that accepts n parameters. Each of the lists l_1, l_2, ..., l_n must have the same number of elements, let's say k. In this case,

$$\texttt{(map } \textit{proc\_exp } l_1 \ l_2 \ \ldots \ l_n\texttt{)}$$

evaluates to a list with k elements which is in the form

$$(L_1 \ L_2 \ \ldots \ L_k)$$

where L_i is the result of the following expression

$$(\textit{proc\_exp } l_1^i \ l_2^i \ \ldots \ l_n^i)$$

where l_j^i is the i^{th} element of l_j.

- For example,

```
(map cons '(a b c) '(1 2 3))
```

evaluates to

```
((a .  1) (b .  2) (c .  3))
```

- Example: Let us try to create a procedure to do the following: We will be given two symbols `old` and `new`, and a possibly nested list of symbols `slst`. We will calculate a list of symbols exactly as `slst` with every occurence of the `old` symbol replaced by an occurence of the `new` symbol. Suppose that we bind this procedure to a variable named `subst` (for "substitute"). The following is a couple of interaction examples with this procedure:

Scheme interaction

```
1 ]=> (subst 'a 'b '()) ; no occurence of old
;Value: ()

1 ]=> (subst 'a 'b '(c)) ; no occurrence of old
;Value: (c)

1 ]=> (subst 'a 'b '(a a a)) ; all occurences of old will be replaced
;Value: (b b b)

1 ]=> (subst 'a 'b '(a c a)) ; all occurences of old will be replaced
;Value: (b c b)

1 ]=> (subst 'a 'b '((a c) a (a d)) ; slst can have sublists
;Value: ((b c) b (b d))
```

Here is how we can create such a procedure:

Scheme interaction

```
1 ]=> (define subst
         (lambda (old new slst)
             (if (null? slst)
                 '()
                 (if (symbol? (car slst))
                     (if (eq? (car slst) old)
                         (cons new (subst old new (cdr slst)))
                         (cons (car slst) (subst old new (cdr slst)))
                     )
                     (cons (subst old new (car slst))
                           (subst old new (cdr slst))
                     )
                 )
             )
         )
      )
;Value: subst
```

- When we bind a value to a variable using `define`, it becomes visible to all the other parts of the program.

- As we can see in the procedure declaration above, there are some expressions that occurs multiple times within the body, for example: "`(subst old new (cdr slst))`".

- For the purposes of information hiding, it would be nice if we could define a local variable binding for these expressions that can be seen only within the body of the procedure, since they don't need to be visible outside the procedure.

- Another special form called "`let`" can be used to do this.

- The general syntax is as follows:

`(let ((`v_1 exp_1`) (`v_2 exp_2`) ... (`v_n exp_n`)) `$expr$`)`

where v_1, v_2, ..., v_n are some variables and exp_1, exp_2, ..., exp_n, $expr$ are some expressions.

- Sometimes the semantics of a construct is explained by giving an equivalent program fragment. For example, assuming that we know about `lambda` expressions, the semantics of a `let` expression can be given in terms of `lambda` expressions in the following way.

`(let ((`v_1 exp_1`) (`v_2 exp_2`) ... (`v_n exp_n`)) `$expr$`)`

is equivalent to

$$\texttt{((lambda (}v_1\ v_2\ \ldots\ v_n\texttt{)}\ expr\texttt{)}\ exp_1\ exp_2\ \ldots\ exp_n\texttt{)}$$

Or in plain English, the semantics of a `let` expression can informally be described as:

- First evaluate all the expressions $exp_1\ exp_2\ \ldots\ exp_n$
- Second bind the value of exp_i to v_i for all $1 \leq i \leq n$.
- Finally, evaluate the expression *expr* by using the bindings of the variables v_i's performed in the previous step.

Scheme interaction

```
1 ]=> (let ((x 3)) (+ x x))
;Value: 6

1 ]=> (define x 4)
;Value: 4

1 ]=> (let ((x 3)) (+ x x))
;Value: 6

1 ]=> x
;Value: 4

1 ]=> (let ((x 3) (y 4)) (+ x y))
;Value: 7

1 ]=> (let ((x 3) (y x)) (+ x y))
;Value: 7
```

- We can rewrite the `subst` procedure by using `let` construct as follows:

Scheme interaction

```
1 ]=> (define subst
         (lambda (old new slst)
             (if (null? slst)
                 '()
                 (let (
                       (first (car lst))
                       (rest (cdr lst))
                      )
                      (if (symbol? first)
                          (if (eq? first old)
                              (cons new (subst old new rest))
                              (cons old (subst old new rest))
                           )
                           (cons (subst old new first)
                                 (subst old new rest)
                           )
                      )
                  )
             )
         )
      )
;Value: subst
```

• Another form of `let` called "`let*`" has a slightly different semantics. The general syntax is the same as `let`, the only difference being the use of keyword "`let*`" instead of the keyword "`let`". Let us explain the semantics by again mapping it in to some other constructs.

$$\texttt{(let* ((}v_1\ exp_1\texttt{))}\ expr\texttt{)}$$

is equivalent to

$$\texttt{(let ((}v_1\ exp_1\texttt{))}\ expr\texttt{)}$$

and

$$\texttt{(let* ((}v_1\ exp_1\texttt{) (}v_2\ exp_2\texttt{) ... (}v_n\ exp_n\texttt{))}\ expr\texttt{)}$$

is equivalent to

$$\texttt{(let ((}v_1\ exp_1\texttt{))}\ \texttt{(let* ((}v_2\ exp_2\texttt{) ... (}v_n\ exp_n\texttt{))}\ expr\texttt{))}$$

Scheme interaction

```
1 ]=> (let* ((x 3)) (+ x x))
;Value: 6

1 ]=> (define x 4)
;Value: 4

1 ]=> (let* ((x 3) (y x)) (+ x y))
;Value: 6
```

- For the purposes of information hiding, if a procedure is not needed globally, it would be better to restrict the scope that it is visible.

- One approach could be to bind the procedure to a variable by using `let`, as follows:

```
(let (
      (abs (lambda (n)
               (if (< n 0)
                   (* n -1)
                   n
      )
      body of let
)
```

- In this case, the procedure will be bound to the variable `abs`, which will be visible only in the body of the let expression.

- When we want declare a recursive procedure, this approach will cause a problem:

```
(let (
      (fact (lambda (n)
               (if (= n 1)
                   1
                   (* n (fact (- n 1))))))
      )
      (fact 3)
)
```

- It would complain that `fact` is not bound. The reason of the problem would be clear if we consider the semantics of a `let statement`

(let ((v_1 e_1) (v_2 e_2) ... (v_n e_n)) expr)

- The semantics of `let` is as follows:
  1. The expressions $\texttt{e}_1$, $\texttt{e}_2$, ..., $\texttt{e}_n$ are evaluated in some unspecified order
  2. Then the variables $\texttt{v}_1$, $\texttt{v}_2$, ..., $\texttt{v}_n$ are bound to storage
  3. Then the values of the expressions $\texttt{e}_1$, $\texttt{e}_2$, ..., $\texttt{e}_n$ are bound to the variables $\texttt{v}_1$, $\texttt{v}_2$, ..., $\texttt{v}_n$

- If we consider the example of `fact` given above, while evaluating $\texttt{e}_1$ (which would be the lambda expression), there is an occurrence of `fact`, but `fact` is not bound to storage yet during this evaluation. Therefore, we get the unbound variable error.

- The solution to this problem is to use another form of the `let` statement, which is `letrec`

- The syntax of `letrec` is the same as that of `let`

$$\texttt{(letrec ((v}_1\ \texttt{e}_1\texttt{) (v}_2\ \texttt{e}_2\texttt{) ... (v}_n\ \texttt{e}_n\texttt{)) expr)}$$

- However, the semantics of `letrec` is a little bit different.
  1. First, the variables $\texttt{v}_1$, $\texttt{v}_2$, ..., $\texttt{v}_n$ are bound to storage (of course with some unspecified values)
  2. Then the expressions $\texttt{e}_1$, $\texttt{e}_2$, ..., $\texttt{e}_n$ are evaluated in some unspecified order
  3. Then the values of the expressions $\texttt{e}_1$, $\texttt{e}_2$, ..., $\texttt{e}_n$ are bound to the variables $\texttt{v}_1$, $\texttt{v}_2$, ..., $\texttt{v}_n$

- If we declare the function `fact` in the following way:

```
(letrec (
      (fact (lambda (n)
                (if (= n 1)
                    1
                    (* n (fact (- n 1))))))
      )
      (fact 3)
)
```

then we will not get the previous error, because the variable `fact` will already be bound to a memory location when the lambda expression is evaluated.

9.7 Multi way selection

- Instead of using a chain of if statements, you can use multi way selection statement, `cond`.
- The syntax of `cond` is as follows:

```
(cond
   ( test_1 expr_1 )
   ( test_2 expr_2 )
   ...
   ( test_n expr_n )
   ( else   expr_else )
)
```

- The semantics of `cond` can be given by providing the equivalent expression using if statements as follows:

```
(if test_1
    expr_1
    (if test_2
        expr_2
        (if test_3
            expr_3
            ...
            (if test_n
                expr_n
                expr_else))...)
```

- We can rewrite the `subst` procedure using the `cond` statement in the following way:

```
(define subst2 (lambda (new old slst)
  (cond
    ((null? slst) '())
    ((symbol? (car slst))
        (if (eq? (car slst) old)
            (cons new (subst new old (cdr slst)))
            (cons (car slst) (subst new old (cdr slst)))))
    (else (cons
             (subst new old (car slst))
             (subst new old (cdr slst)))))))
```

9.8 Interpreting Scheme using Scheme

- Notice that, the syntax of the expressions in Scheme looks a lot like the lists in Scheme.
- Consider the expression that adds the numbers 1 and 2: `(+ 1 2)`
- Also onsider a list of three elements which are the symbol `+` and the numbers 1 and 2: `(+ 1 2)`.
- All the other expressions can also be shown to have spellings in the form of lists.
- We will start implementing Scheme interpreters (getting more and more complicated at each step), using Scheme.
- Our interpreters will be a procedure with a single argument, which would be a Scheme expression, and will print out the value of this Scheme expression after evaluating it.
- In our first interpreter, we restrict the language into a very very small subset, let's call it `s1`, which is described by the following grammar:

```
<s1> -> NUMBER
```

- In this subset of Scheme, as you can understand from the grammar, we will allow only expressions which are simple numbers (e.g. 1, -5, -.5, etc.).
- In fact, interpreting such a simple language is quite easy and can be performed by the following procedure `s1-interpret`.

s1.scm

```
(define s1-interpret (lambda (e)
  (cond
    ;; If the input expression is a number, then
    ;; the value of the expression is that number.
    ((number? e) e)

    ;; If the expression is not a number then,
    ;; we cannot handle it. Produce an error.
    (else (error "s1-interpret: cannot evaluate -->" e)))))
```

- If the actual parameter of `s1-interpret` is a number, then it will simply return the number. However, if the argument is not a number, since `s1-interpret` is not meant to interpret such arguments, it prints out an error message by providing the erroneous argument. Usage examples of `s1-interpret` are given below:

Scheme interaction

```
1 ]=> (load "s1.scm")
;Loading "s1.scm" -- done
;Value: s1-interpret

1 ]=> (s1-interpret 5)
;Value: 5

1 ]=>  (s1-interpret -5.4)
;Value: -5.4

1 ]=> (s1-interpret 0.3)
;Value: .3

1 ]=> (s1-interpret 'a)
;s1-interpret: cannot evaluate --> a
...

1 ]=> (s1-interpret '(+ 1 2))
;s1-interpret: cannot evaluate --> (+ 1 2)
...
```

- Now let us enlarge our subset a little bit. Let the name of the new subset be `s2`. The grammar for `s2` is given below:

```
<s2> -> NUMBER
     |  ( + NUMBER NUMBER )
```

- As you can see, in addition to the simple expressions that are composed of single numbers, `s2` also allows the expressions that adds two numbers.

- The interpreter for `s2` is given below.

s2.scm

```
(define s2-interpret (lambda (e)
  (cond
    ;; If the input expression is a number, then
    ;; the value of the expression is that number.
    ((number? e) e)

    ;; Otherwise, we must see a list.
    ((not (list? e))
          (error "s2-interpret: cannot evaluate -->" e))

    ;; The length of the list must be 3.
    ((not (= (length e) 3))
          (error "s2-interpret: cannot evaluate -->" e))

    ;; The 2nd element of the list (1st operand of the
    ;; expresion)) must be a number.
    ((not (number? (cadr e)))
          (error "s2-interpret: cannot evaluate -->" e))

    ;; The 3rd element of the list (2nd operand of the
    ;; expression) must be a number.
    ((not (number? (caddr e)))
          (error "s2-interpret: cannot evaluate -->" e))

    ;; The 1st element of the list (the operator of the
    ;; expression must be +.
    ;; If it is, evaluate the value of the expression.
    ((equal? '+ (car e))
          (+ (cadr e) (caddr e)))

    ;; The expression is not in a form that we can handle.
    ;; Produce an error.
    (else (error "s2-interpret: cannot evaluate -->" e)))))
```

• The interpreter `s2-interpret` will be able to handle expressions like `(+ 1 2)` in addition to single number expressions that `s1-interpret` were able to handle. However, we are still restricted to the exactly two number additions. Expressions of the form `(+ 1 2 3)` (where there are three operands), or of the form `(+ 1 (+ 2 3))` (where an operand is not a number) will produce errors as required. Usage examples of `s2-interpret` are given below:

Scheme interaction

```
1 ]=> (load "s2.scm")
;Loading "s2.scm" -- done
;Value: s2-interpret

1 ]=> (s2-interpret '3)
;Value: 3

1 ]=> (s2-interpret '(+ 3 5))
;Value: 8

1 ]=> (s2-interpret '(+ 2.5 -0.2))
;Value: 2.3

1 ]=> (s2-interpret '(+ 1 2 3))
;s2-interpret: cannot evaluate --> (+ 1 2 3)
...

1 ]=> (s2-interpret '(+ 1 (+ 2 3)))
;s2-interpret: cannot evaluate --> (+ 1 (+ 2 3))
...
```

- Actually, the interpreter will get a little simpler when we allow subexpressions, since some of the sanity checks can be performed by simply calling the interpreter recursively.

- Assume that we define the language `s3` with the following grammar:

```
<s3> -> NUMBER
     |  ( + <s3> <s3> )
```

so that we cover expressions like `(+ 1 (+ 3 4))`.

- When the interpreter `s2-interpret` sees an expression with two operands, it is sure that (or makes sure that) those operands are simple numbers. Hence getting the values of these operands is quite easy (just use those numbers). However, for `s3-interpret`, this is not the case. The operands may not be simple numbers. Therefore `s3-interpret` has to evaluate the values of these operands first, which can be performed by calling `s3-interpret` recursively on these operands.

- Note that, we will get rid of the `number?` checks for the first and the second operand, since they don't have to be numbers anymore. The values of these operands must turn out to be numbers, but this check will be performed by the innermost recursive call to the `s3-interpret` where the expression is not a list. This will simplify the body of the interpreter.

- However, as it is forced by the grammar, we still have to make sure that an expression has

exactly two operands.

- The implementation of `s3-interpret` is given below:

s3.scm

```
(define s3-interpret (lambda (e)
  (cond
    ;; If the input expression is a number, then
    ;; the value of the expression is that number.
    ((number? e) e)

    ;; Otherwise, we must see a list.
    ((not (list? e))
          (error "s3-interpret: cannot evaluate -->" e))

    ;; The length of the list must be 3.
    ((not (= (length e) 3))
          (error "s3-interpret: cannot evaluate -->" e))

    ;; The 1st element of the list (the operator of the
    ;; expression must be +.
    ;; If it is, first evaluate the value of the operands
    ;; by recursive calls, and then evaluate the
    ;; the value of the entire expression.
    ((equal? '+ (car e))
          (+ (s3-interpret (cadr e)) (s3-interpret (caddr e))))

    ;; If the expression is not in a form that we can handle
    ;; then produce an error.
    (else (error "s3-interpret: cannot evaluate -->" e)))))
```

- The interpreter `s3-interpret` will be able to handle expressions like `(+ 1 (+ 2 3))` in addition to the form of expressions that `s2-interpret` were able to handle. However, we cannot handle yet the expressions with 0, 1, 3, 4, 5, ... operands. Usage examples of `s3-interpret` are given below:

Scheme interaction

```
1 ]=> (load "s3.scm")
;Loading "s3.scm" -- done
;Value: s3-interpret

1 ]=> (s3-interpret '3)
;Value: 3

1 ]=> (s3-interpret '(+ 3 5))
;Value: 8

1 ]=> (s3-interpret '(+ 1 (+ 2 3)))
;Value: 6

1 ]=> (s3-interpret '(+ (+ 1 2) (+ 3 (+ 4 5))))
;Value: 15

1 ]=> (s3-interpret '(+ 1 2 3))
;s3-interpret: cannot evaluate --> (+ 1 2 3)
...
```

- Going one step further, let us allow any number of operands for the addition operator, which will make the interpreter even simpler. The grammar for this new subset, `s4`, is given below.

```
      <s4> -> NUMBER
           |  ( + <s4_exprs> )
<s4_exprs> -> <s4> <s4_exprs>
           |  empty
```

- Note that, the nonterminal `<s4_exprs>` can produce any number of operands to the operator `+`, including 0.

- We can eliminate the check for the number of operands ("`(= (length e) 3)`") that we used to have in the previous interpreters, which will simplify the code.

- However, not knowing the number of operators may also cause some difficulty. Note that we evaluate the values of the operands in `s3-interpret`, by calling `s3-interpret` on "`(cadr e)`" and "`(caddr e)`", which are the first and the second operands respectively. For `s4-interpret`, the number of operands cannot be decided statically, "`(caddr e)`" (when there is only 1 operand), or even "`(cadr e)`" (when there are no operands) can be undefined. On the other hand, if there are more than two operands, we must call `s4-interpret` recursively on "`(cadddr e)`", "`(caddddr e)`", etc.

- Let us take a step back, and think about what we want to do:
We have a list of operands, which is `(cdr e)`, and we want to call `s4-interpret` on every element

of this list. This is a perfect job for the `map` procedure of Scheme, which applies a given procedure on every element of a given list, and returns the results in the form of a list. In other words,

`(map s4-interpret (cdr e))`

will produce a list, where n^{th} element of the list will be the value of the n^{th} operand as evaluated by `s4-interpret`.

- After having the values of the operands evaluated by `s4-interpret`, all we have to do is to add all these values. This is a perfect job for the `apply` procedure of Scheme. Recall that,

`(apply` *proc list*`)`

applies the procedure *proc* to the operands given in the *list*. Therefore, if we `apply` the addition procedure `+` to the list of values returned by the `map` procedure, we would have the desired result.

- Please note the difference between `apply` and `map`. If we consider a list `l` defined to be `(`$\mathtt{v}_1$ $\mathtt{v}_2$ ... $\mathtt{v}_n$`)`, then

`(map` *proc* `l)`

returns the list

`((`*proc* $\mathtt{v}_1$`) (`*proc* $\mathtt{v}_2$`) ... (`*proc* $\mathtt{v}_n$`))`

whereas,

`(apply` *proc* `l)`

returns the value

`(`*proc* $\mathtt{v}_1$ $\mathtt{v}_2$ ... $\mathtt{v}_n$`)`

- An interpreter for the subset `s4` is given below:

s4.scm

```
(define s4-interpret (lambda (e)
  (cond
    ;; If the input expression is a number, then
    ;; the value of the expression is that number.
    ((number? e) e)

    ;; Otherwise, we must see a list.
    ((not (list? e))
          (error "s4-interpret: cannot evaluate -->" e))

    ;; First evaluate the value of the operands
    ;; by recursive calls, and then evaluate the
    ;; the value of the entire expression.
    ((equal? '+ (car e))
          (apply + (map s4-interpret (cdr e))))

    ;; If the expression is not in a form that we can handle
    ;; then produce an error.
    (else (error "s4-interpret: cannot evaluate -->" e)))))
```

- Now we can use the interpreter `s4-interpret` for almost all kind of expressions as given below.

Scheme interaction

```
1 ]=> (load "s4.scm")
;Loading "s4.scm" -- done
;Value: s4-interpret

1 ]=> (s4-interpret '3)
;Value: 3

1 ]=> (s4-interpret '(+ 1 2))
;Value: 3

1 ]=> (s4-interpret '(+ (+ 1 2) (+ 3) 4))
;Value: 10

1 ]=> (s4-interpret '(+ 1 2 3 4 5))
;Value: 15

1 ]=> (s4-interpret '(+ 3))
;Value: 3

1 ]=> (s4-interpret '(+))
;Value: 0
```

- Of course, when we say "almost all kind of expressions", it is still a restricted subset of the actual Scheme expressions.

- However, extending to the following subset where other operations (procedures) on numbers are allowed is quite easy.

- Let `s5` be the name of the new subset whose grammar is given below.

```
      <s5> -> NUMBER
           |  ( + <s5_exprs> )
           |  ( - <s5_exprs> )
           |  ( * <s5_exprs> )
           |  ( / <s5_exprs> )
<s5_exprs> -> <s5> <s5_exprs>
           |  empty
```

- Within this new subset, we will be able to cover expressions like `(- (+ 5 3) (* 2 4) (/ 3 2))`.

- An interpreter for `s5` is given below:

s5t.scm

```
(define s5-interpret (lambda (e)
  (cond
    ;; If the input expression is a number, then
    ;; the value of the expression is that number.
    ((number? e) e)

    ;; Otherwise, we must see a list.
    ((not (list? e))
          (error "s5-interpret: cannot evaluate -->" e))

    ;; First evaluate the value of the operands
    ;; by recursive calls, and then evaluate the
    ;; the value of the entire expression.

    ;; If the operator is addition, then apply +
    ((equal? '+ (car e))
          (apply + (map s5-interpret (cdr e))))

    ;; If the operator is subtraction, then apply -
    ((equal? '- (car e))
          (apply - (map s5-interpret (cdr e))))

    ;; If the operator is multiplication, then apply *
    ((equal? '* (car e))
          (apply * (map s5-interpret (cdr e))))

    ;; If the operator is division, then apply /
    ((equal? '/ (car e))
          (apply / (map s5-interpret (cdr e))))

    ;; If the expression is not in a form that we can handle
    ;; then produce an error.
    (else (error "s5-interpret: cannot evaluate -->" e)))))
```

• As you may have noticed, this approach of implementing various operators will get quite lengthy as we keep adding new procedures. Let us reconsider our approach and try to switch to a design which will make the interpreter shorter. Another interpreter for `s5` is given below.

s5.scm

```
(define get-operator (lambda (op-symbol)
  (cond
    ((equal? op-symbol '+) +)
    ((equal? op-symbol '-) -)
    ((equal? op-symbol '*) *)
    ((equal? op-symbol '/) /)
    (else (error "s5-interpret: operator not implemented -->" op-symbol)))))

(define s5-interpret (lambda (e)
  (cond
    ;; If the input expression is a number, then
    ;; the value of the expression is that number.
    ((number? e) e)

    ;; Otherwise, we must see a list.
    ((not (list? e))
          (error "s5-interpret: cannot evaluate -->" e))

    ;; First evaluate the value of the operands
    ;; and the procedure of the expression.
    (else
       (let ((operands (map s5-interpret (cdr e)))
             (operator (get-operator (car e))))

         ;; And finally apply the operator to the
         ;; values of the operands
         (apply operator operands))))))
```

- Below is an example of interaction with the `s5-interpret`

Scheme interaction

```
1 ]=> (load "s5.scm")
;Loading "s5.scm" -- done
;Value: s5-interpret

1 ]=> (s5-interpret '3)
;Value: 3

1 ]=> (s5-interpret '(* 2 4))
;Value: 8

1 ]=> (s5-interpret '(- (+ 5 3) (* 2 4) (/ 3 2)))
;Value: -3/2
```

• The interpreter for `s5` as given by the file `s5.scm` works well, however, it is still missing an REPL. Let us now try to build an environment where we continuously interact with our interpreter.

• Note that, in order to implement an interactive REPL, we will need a procedure to read an input from the user. MIT Scheme's `read` procedure can be used for this purpose. When `read` is called (it takes no arguments), it will read an input from the keyboard, which is supposed to be an external representation of a Scheme object, and it will evaluate to that object.

• In other words, if the input is a representation of a symbol, than it will return the object of that symbol. If the input is a representation of a list, it will return that list as an object, and so on so forth.

• If the input provided is not a parsable external representation, then it will produce an error.

• The following Scheme interaction is provided to explain the behavior of the `read` procedure.

Scheme interaction

```
1 ]=> (read)
abc
;Value: abc

1 ]=> (symbol? (read))
abc
;Value: #t

1 ]=> (read)
(+ 1 2)
;Value 1: (+ 1 2)

1 ]=> (symbol? (read))
(+ 1 2)
;Value: ()

1 ]=> (list? (read))
(+ 1 2)
;Value: #t

1 ]=> (list? (read))
#(+ 1 2)
;Value: ()

1 ]=> (vector? (read))
#(+ 1 2)
;Value: #t
```

- Below is an interpreter for `s5` with an REPL.

s5repl.scm

```
(define get-operator (lambda (op-symbol)
  (cond
    ((equal? op-symbol '+) +)
    ((equal? op-symbol '-) -)
    ((equal? op-symbol '*) *)
    ((equal? op-symbol '/) /)
    (else (error "s5-interpret: operator not implemented -->" op-symbol)))))

(define repl (lambda ()
  (let* (
         ; first print out some prompt
         (dummy1 (display "cs305> "))

         ; READ an expression
         (expr (read))

         ; EVALuate the expression read
         (val (s5-interpret expr))

         ; PRINT the value evaluated together
         ; with a prompt as MIT interpreter does
         (dummy2 (display "cs305: "))
         (dummy3 (display val))

         ; get ready for the next prompt
         (dummy4 (newline))
         (dummy4 (newline)))
     (repl))))

(define s5-interpret (lambda (e)
  (cond
    ;; If the input expression is a number, then
    ;; the value of the expression is that number.
    ((number? e) e)

    ;; Otherwise, we must see a list.
    ((not (list? e))
          (error "s5-interpret: cannot evaluate -->" e))

    ;; First evaluate the value of the operands
    ;; and the procedure of the expression.
    (else
       (let ((operands (map s5-interpret (cdr e)))
             (operator (get-operator (car e))))

         ;; And finally apply the operator to the
         ;; values of the operands
         (apply operator operands))))))
```

- Let us examine the `repl` procedure first. In every activation of this procedure, a prompt is display (which is chosen to be `cs305>`, and then an input is `read` from the user. After that, the previous `s5-interpreter` is called in order to evaluate the value of the expression, that is entered by the user. The value returned by `s5-interpret` is bound to the variable `val`. Then this value is printed out, which is followed by emitting two newline characters. After that, a new activation of `repl` takes place.

- Note that, for displaying the prompts, some bindings to dummy variables are used.

- The following is an example interaction with the our new interactive interpreter for `s5`.

Scheme interaction

```
1 ]=> (load "s5repl.scm")
;Loading "s5repl.scm" -- done
;Value: s5-interpret

1 ]=> (repl)
cs305> 3
cs305: 3

cs305> (+ 1 2)
cs305: 3

cs305> (* 2 4)
cs305: 8

cs305> (+ 1 (* 2 4))
cs305: 9
```

- Let us now extend our language one more time, and include the `define` statements.

- Note that, when we include the `define` statement, it means that we now allow values to be bound to variables. Therefore, it will be meaningful to allow variables to be referenced in the expressions.

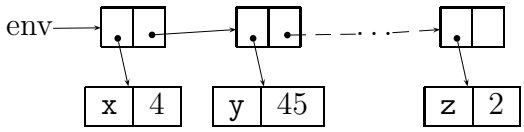
- The grammar for the new subset, whose name is `s6` is given below:

```
      <s6> -> <s6_expr>
           |  <define>
  <define> -> ( define IDENT <s6_expr> )
<s6_expr> -> NUMBER
           |  IDENT
```

```
           |  ( + <s6_exprs> )
           |  ( - <s6_exprs> )
           |  ( * <s6_exprs> )
           |  ( / <s6_exprs> )
<s6_exprs> -> <s6_expr> <s6_exprs>
           |  empty
```

- Note that, when a value is bound to a variable using `define`, we need to remember its value, so that when that variable is referenced later in an expression, we can use its value.

- An easy way of doing this is to keep a list of variables together with their associated value. For example, the following structure can be used to keep the list of currently defined variables and their values.



- The current variable bindings are called an **environment**. When a variable reference is made, the value of the variable must be searched and accessed within the current environment.

- In our approach given above, we keep an environment as a list of pairs, where the `car` of each field gives the name of the variable, and the `cdr` of a pair gives the current binding of that variable.

- An interpreter for `s6`, must extend the environment with a new such pair when a `define` statement is seen.

- And when a variable reference is seen, it must scan the environment at that moment, and find the value of the variable.

- An interpreter for `s6` is given below:

s6-NoComment.scm

```
(define get-operator (lambda (op-symbol)
  (cond
    ((equal? op-symbol '+) +)
    ((equal? op-symbol '-) -)
    ((equal? op-symbol '*) *)
    ((equal? op-symbol '/) /)
    (else (error "s6-interpret: operator not implemented -->" op-symbol)))))

(define define-stmt? (lambda (e)
    (and (list? e) (equal? (car e) 'define) (symbol? (cadr e)) (= (length e) 3))))

(define get-value (lambda (var env)
    (cond
      ((null? env) (error "s6-interpret: unbound variable -->" var))
      ((equal? (caar env) var) (cdar env))
      (else (get-value var (cdr env))))))

(define extend-env (lambda (var val old-env)
      (cons (cons var val) old-env)))

(define repl (lambda (env)
  (let* (
         (dummy1 (display "cs305> "))
         (expr (read))
         (new-env (if (define-stmt? expr)
                      (extend-env (cadr expr) (s6-interpret (caddr expr) env) env)
                      env))
         (val (if (define-stmt? expr)
                  (cadr expr)
                  (s6-interpret expr env)))
         (dummy2 (display "cs305: "))
         (dummy3 (display val))
         (dummy4 (newline))
         (dummy4 (newline)))
     (repl new-env))))

(define s6-interpret (lambda (e env)
  (cond
    ((number? e) e)
    ((symbol? e) (get-value e env))
    ((not (list? e)) (error "s6-interpret: cannot evaluate -->" e))
    (else
       (let ((operands (map s6-interpret (cdr e) (make-list (length (cdr e)) env)))
             (operator (get-operator (car e))))
         (apply operator operands))))))

(define cs305-interpreter (lambda () (repl '())))
```

• The comments are removed from the file to be able to show the entire interpreter on one page. The version with comments can be found in the file `s6.scm`.

• The procedure `get-value` is responsible for looking up the current value of a variable in an environment. It is called with two arguments. The first one is `var`, which is the variable whose current value is referenced. The second parameter, `env` is an environment within which the variable will be searched for. It simply scans through all the variable bindings in the provided environment. It either returns the current value of `var` in `env`, if `var` is bound to a value in `env`, or otherwise it produces the error of "unbound variable".

• The procedure `extend-env` extends an environment with a new variable binding. Note that, it simply adds the new binding to the head of the old environment. Since `get-value` scans the environments starting from the head, `get-value` will always find the last binding of a variable. However, by this approach we will be creating duplicate bindings of multiply defined variables, hence it is not a very efficient use of the memory.

• Note that, the procedure `repl` now takes a parameter. It is the environment in which the an iteration of `repl` will take place. It is initially called with the empty environment. When a `define` statement is seen, the environment will be extended, and the next iteration of the `repl` will use the new extended environment properly.

• Similarly, `s6-interpret` also makes use of an additional parameter, which is the environment in which the evaluations would take place.

• There is a different use of `map` in the body of `s6-interpret`. Note that, `map` also has another parameter. In the previous usages of `map`, the procedure that was `map`ped used to accept a single parameter. However, `s6-interpret` needs two arguments as explained above. We can still use `map` to evaluate the values of the operands. However, this time we use the general form of the `map` procedure, which is explained below.

• In its general form, the `map` procedure had the following syntax and semantics. Let l_i be defined to be a list of values

$$(\mathtt{l}_i^1\ \mathtt{l}_i^2\ \ldots \mathtt{l}_i^n\ )$$

If *proc* is a procedure that takes k arguments, then

$$(\mathtt{map}\ \ \mathit{proc}\ \mathtt{l}_1\ \mathtt{l}_2\ \ldots\ \mathtt{l}_k)$$

evaluates to the list

$$(\ (\mathit{proc}\ \mathtt{l}_1^1\ \mathtt{l}_2^1\ \ldots\ \mathtt{l}_k^1)\ (\mathit{proc}\ \mathtt{l}_1^2\ \mathtt{l}_2^2\ \ldots\ \mathtt{l}_k^2)\ \ldots\ (\mathit{proc}\ \mathtt{l}_1^n\ \mathtt{l}_2^n\ \ldots\ \mathtt{l}_k^n)\ )$$

• The Scheme procedure `make-list` has the following syntax and semantics.

$$(\mathtt{make\text{-}list}\ n\ \mathit{expr})$$

where n is an integer, and *expr* is an expression, produces the following list

$$(val \ \ val \ \ \ldots \ \ val)$$

where there are n copies of *val*, and where *val* is the value of the expression *expr*.

- Hence by duplicating the current environment by using `make-list` in the body of the `s6-interpret`, we create the environments that will be used while evaluating the values of the expressions in `(cdr e)`.

- Now let us interact with our interpreter a little bit:

Scheme interaction

```
1 ]=> (load "s6.scm")
;Loading "s6.scm" -- done
;Value: cs305-interpreter

1 ]=> (cs305-interpreter)
cs305> 3
cs305: 3

cs305> (+ 1 2)
cs305: 3

cs305> (+ (* 2 4) (- 4 2) (/ 3 2 1))
cs305: 23/2

cs305> (define x 3)
cs305: x

cs305> (define y (+ 1 2))
cs305: y

cs305> x
cs305: 3

cs305> y
cs305: 3

cs305> (* (- x 1) (+ y 1))
cs305: 8
```

- Note that, when a `define` statement is used, the binding performed by this statement is permanent. Hence, it directly influence the top level environment. We may have different, temporary

environments that are used while evaluating the body of the `let` statements, as will be shown below.

• Now let us try to include the `let` statements into our language. First the grammar for the new subset `s7`.

```
          <s7> -> <s7_expr>
               |  <special>
     <special> -> <define>
               | <let>
      <define> -> ( define IDENT <s7_expr> )
         <let> -> ( let ( <var_bindings> ) <s7_expr> )
<var_bindings> -> <var_binding> <var_bindings>
               |  empty
 <var_binding> -> ( IDENT <s7_expr> )
     <s7_expr> -> NUMBER
               |  IDENT
               |  ( + <s7_exprs> )
               |  ( - <s7_exprs> )
               |  ( * <s7_exprs> )
               |  ( / <s7_exprs> )
    <s7_exprs> -> <s7_expr> <s7_exprs>
               |  empty
```

• Note that, when a `let` is seen, the variable bindings in the `<var_bindings>` part must extend the current environment temporarily, only for use while evaluating the body of the let expression.

• While evaluating an `<s7_expr>` in `<var_bindings>`, the environment of the `let` statement must be used.

• However, these bindings must be reflected in the top level environment.

• After the evaluation of a `let` statement, the top level environment should stay the same as it was before the evaluation of the `let` statement. In other words, the following interaction must be possible:

Scheme interaction

```
1 ]=> (load "s7.scm")
;Loading "s7.scm" -- done
;Value: cs305-interpreter

1 ]=> (cs305-interpreter)
cs305> (define x 3)
cs305: x

cs305> x
cs305: 3

cs305> (let ((x 5)) x)
cs305: 5

cs305> x
cs305: 3
```

Chapter 10

Parallel and Concurrent Programming

- The terms **parallel** and **concurrent** denote related but different notions.

- Parallelism usually denotes the execution at the same time, literally. For example, in a dual processor computer, two programs may be executed in two different CPUs in parallel.

- Whereas concurrency usually denotes the possibility of executing one or more programs (or parts of the same program) in parallel.

- For example, the following two statements can be executed concurrently:

```
x = x + 2;
y = y / 2;
```

- The reason why we call these two statements as concurrent is as follows: Note that, normally when the two statements above appear in a program, they would be executed sequentially. In other words, first the statement `x = x + 2;` would be executed, and then the statement `y = y / 2;` would be executed. However, there is no reason (other than the programmer's state of mind while putting these statements in this order) for them to be executed in this order. These two statements could be executed in the reverse order, and it will make no difference at the output of the program. In fact, these two statements could also be executed in parallel, and it would still make no difference.

- Note that, the statements above are not valid for the code fragment below:

```
x = y + 2;
y = y / 2;
```

In this case, since the first statement depends on the second statement, the order of execution of these two statements are important. Changing the order of execution for these two statements may affect the outcome of the program.

- When there are multiple processors, concurrent execution of programs can be realized by executing different parts of a program or different programs simultaneously on different processors. However, even with a single processor, a logical form of concurrency can be realized by time sharing the processor between different programs.

10.1 Multiprocessor architectures

- We need to revisit the computer architecture for parallel programming.

- In the classical von Neumann architecture, there used to be only a single processor. However, in computers that are used for parallel execution of programs, there are multiple processors, and the connection of these processors to each other and to memory may vary.

- In fact, the term "multiprocessor" does not apply only to the machines that actually have multiple CPUs (general purpose processing units). We can consider a machine with a single CPU, but with several special purpose processing units (e.g. an I/O processor, a math processor) as multiprocessor as well.

- When there are multiple processors, the compilers that are used to generate the executables must also take care of the job scheduling. That is, the compiler must decide the processing units on which the instructions will be executed. This is valid even for single general purpose CPU plus co–processor architectures. For example, if we consider the code fragment:

```
x = x + 2;
y = y / z;
```

the first statement can be performed by the CPU, and at the same time, the right hand side of the second statement can be calculated by the floating point co–processor.

- The machines with multiple general purpose processors are classified as follows:

  - SIMD (Single Instruction – Multiple Data) machines:
    The same program runs on different CPUs simultaneously. However, each copy of the program acts on a different data. This is actually equivalent to executing the same program multiple times on the same CPU at different times. These machines are usually used in the area of scientific computing. One of the best examples is the **vector processors**. Each CPU has its local memory, and no synchronization is required between the programs.

  - MIMD (Multiple Instruction – Multiple Data) machines:
    Each processor executes a different program, and acts on different data. Usually synchronization of these concurrent programs are needed.

- MIMD machines are further divided into two groups based on the connection of the CPUs to the memory.

  - Shared memory architecture: All the CPUs use the same memory. The synchronization between the concurrent programs can be established by using the same memory location as shared by different processors.

  - Distributed memory architecture: Each processor has its own memory. In this configuration, the synchronization between the processors has to be established by message passing.

10.2 Fundamental Concepts

- A **task** is a unit of program that can run concurrently with other tasks. A program consists of several tasks.

- A task is simply a sequential program and it has a thread of control of itself.

- Different from subprogram calls, when a task of a program is started, the control is not transferred from the caller to the called task. Instead, the new task has another thread of control, different from all the active tasks.

- The program has a certain job to do. The tasks of a concurrent program cooperate in order to achieve that job. For this reason, the tasks will have to communicate (synchronize) with each other at some points. This synchronization among tasks are accomplished either by message passing, or by using shared memory.

- Synchronization among tasks controls the order in which the tasks execute. Another reason of synchronization is to pass information from one task to another (for example to pass the result of a computation that is performed by one task to another).

- Rarely, a task may be **disjoint**, meaning that it does not need to communicate with other tasks.

10.3 Ada

- Ada is a parallel programming language.

- The following is the template of Ada procedure declarations

```
procedure <name> is
  <declarations>
begin
  <statements>
end <name>;
```

- The `<name>` after the `procedure` keyword, and the `<name>` after the `end` keyword must be the same.

- In the `<declarations>` part, some declarations that will be used by the procedure are made (e.g. subprocedures, local variables, tasks, etc.)

- The `<statements>` part is the body of the procedure.

- A more concrete example of a procedure in Ada is given below:

```
with text_io; use text_io; -- import i/o procedures
procedure HelloWorld is
  -- we do not need any declaration for this procedure
begin
  put_line ("Hello world!!!");
end HellOwOrld;
```

- As can be seen from the example above, the keyword `--` is used to start a comment on a single line.
- The keyword "`with`" is used to import a library.
- The keyword "`use`" is used for scope resolution.
- The procedure above will print out `Hello world!!!` when executed.
- The concurrent units in Ada are described by using the tasks.
- A task declaration is performed in two steps: task specification and task body
- The specification of a task is simply performed by using the keyword `task` and by giving a name to the task as:

```
task <name>;
```

- The body of a task is given as follows:

```
task body <name> is
  <declarations>
begin
  <statements>
end <name>;
```

Again, the `<name>`s at the beginning and at the end of the body must match.

- The following is a complete example of an Ada procedure with tasks:

```
with text_io; use text_io;

procedure identify is

  -- here is one task
  task p;

  task body p is
  begin
```

```
    put_line("p");
  end p;

  -- here is another task
  task q;

  task body q is
  begin
    put_line("q");
  end p;

begin -- of the procedure
  put_line("r");
end identify;
```

- When a procedure starts executing, all the tasks in it also start.

- Hence, the execution of the bodies of the tasks `p` and `q` will also start when the procedure `identify` starts.

- Note that, there are three execution threads (one for the task `p`, another for the task `q`, and yet another for the procedure `identify`). Since these threads do not synchronize with each other, the output of this procedure can be the output of `p`, `q` and `r` in some order.

10.3.1 A quick Ada tutorial

- Ada is a case–insensitive language. Recall that we have mentioned Ada before as one of the safest languages. In other words, the features of this language force users to write safe programs. Case–insensitivity forbids using similar looking identifiers for different objects. Hence it increases readability, which is very important for identifying errors in a program.

- A procedure can accept parameters. However it cannot return a value.

```
procedure sum (x, y :  in integer; z :  out integer )
```

- A function can also accept parameters, and it can return a value.

```
function mult (x, y :  integer) return integer
```

- The parameter passing method can be described for each formal parameter. There are three possible modes for formal parameters: `in`, `out`, `in out`

- The default mode is `in`

- Functions and procedures can be declared within other functions and procedures for information hiding.

```
procedure sum (x, y : in integer; z : out integer ) is begin
  z := x + y;
end sum;

function mult (x, y : integer) return integer is
  result : Integer;
begin
  result := x * y;
  return result;
end mult;

function sum_sqr (x,y : integer) return integer is
  sqr1, sqr2, res : integer;
  procedure sqr (x: integer; res : out integer) is
  begin
    res := x * x;
  end  sqr;

begin
  sqr(x,sqr1);
  sqr(y,sqr2);
  res := sqr1+sqr2;
  return res;
end sum_sqr;
```

• As numerical data types, integer and floats are available. A variable can be initialized during its declaration.
Integer type → `A : integer := 1;`
Float type → `B : Float := 0.0;`

• Coercion (i.e. implicit type conversion) is a dangerous feature and it may cause errors since it may not be intentional. Being safe language Ada does not allow coersion:

```
C : Float;
C := Float(A) + B;
```

• The usual arithmetic operators are available.
`+ - * / **` (`**` is exponentiation)

• The following comparison operations are available:

`= /= > >= < <=`

- Boolean type → `D : Boolean := True;`

- Boolean Operations : `and or xor not`

- `type MyRange is range 25 .. 74;`

- `type MyOtherRange is range 30 .. 70;`

- `X,Y : MyRange;`
`Z : MyOtherRange;`
`X = Y + Z;`
This would produce an error since the operands of the addition are not the same. We should have an explicit casting for `Z` to make this statement type safe.
`X = Y + MyRange(Z);`
Note that, the result of the addition `Y + MyRange(Z)` may produce a value that is out of range for the type of `X`. Such a check cannot be performed statically. However, if it goes out of range, a runtime exception will be raised.

- Another way of making the expression `Y+Z` to be of type `MyRange` is to declare `MyOtherRange` as a subtype as follows:
`subtype MySubRange is MyRange range 30 .. 70;`

- `type Day is (Mon, Tue, Wed, Thu, Fri, Sat, Sun);`

- `subtype WeekDay is Day range Mon .. Fri;`

- `CurrDay : Day;`

- `CurrDay := Wed;`

- `type Table is array (1 .. 100) of integer;`

- `type Schedule is array(Day) of Boolean;`

- `type Coordinate is array (-10 .. 10, -10 .. 10) of Float;`

- `WorkingOn : Schedule;`

- `WorkingOn(Sunday) := false;`

- Record types:

```
type Date is record
  Day   : Integer range 1 .. 31;
```

```
  Month : Integer range 1 .. 12;
  Year  : Integer range 1 .. 4000 := 2004;
end record;
```

- `MyBirthDay :  Date;`
- `MyBirthDay.Day := 17;`
- `MyBirthDay.Month := 3;`
- `MyBirthDay.Year := 1987;`
- If statement:

```
if <condition> then
   <statements>
elseif <condition> then
   <statements>
elseif <condition> then
   <statements>
...
else
   <statements>
end if;
```

- Case statement:

```
case <expr0> is
  when <expr1> => <statements>
  when <expr2> => <statements>
  ...
  when others  => <statements>
end case;
```

```
case X is
  when 1           => foo1;
  when 2 .. 5      => foo2;
  when 6 | 8       => foo3;
  when 7 | 9 .. 11 => foo4;
  when others      => foo5;
end case;
```

- Infinite loop:

```
loop
  <statements>
end loop;
```

```
  X := 1; Y := 5;
loop
  X := X * Y;
  Y := Y - 1;
  exit when Y = 1;
end loop;
ada.Integer_Text_IO.Put(X);
```

- While loop:

```
while <condition> loop
  <statements>
end loop;
```

```
X := 1;
Y := 5;
while Y /= 1 loop
  X := X * Y;
  Y := Y - 1;
end loop;
ada.Integer_Text_IO.Put(X);
```

- For loop:

```
for <identifier> in <range> loop
  <statements>
end loop;
```

```
X := 1;
Y := 5;
for count in 1 .. Y loop
  X := X * count;
end loop;
ada.Integer_Text_IO.Put(X);
```

```
for <identifier> in reverse <range> loop
  <statements>
end loop;
```

```
X := 1;
Y := 5;
for count in reverse 1 .. Y loop
  X := X * count;
end loop;
ada.Integer_Text_IO.Put(X);
```

10.4 Synchronization in Ada

• When concurrent programs works cooperatively to accomplish some job, they have to synchronize with each other. Incorrect synchronization results in two types of error:

- Deadlock: A set of concurrent programs are waiting for each other, and none of them is able to perform anything. Such an error is a violation of a safety property. Safety properties of programs state that "nothing bad will ever happen" (e.g. a resource will never be allocated to more than one entity).
- Livelock: A set of concurrent programs are waiting for each other again, but they are not completely blocked. They are able to perform some activity, but these activities do not correspond to any useful objective. Such an error is a violation of a liveness property. Liveness properties of programs state that "something good will eventually happen" (e.g. an entity requesting a resource will eventually get it).

• Tasks need to communicate with each other in order to guarantee safety and liveness properties.

• Let us consider the *Dining Philosophers* problem to give examples of these notions

• There are $n \geq 2$ philosophers sitting around a circular dining table.

• There are n plates.

• There is one plate infront of each philosopher.

• Between each plate on the table, there is a fork (so we have n forks).

• The philosophers continuously *think* and *eat*. When a philosopher wants to eat, he has to pick the two forks on the left and on the right of his plate. Only after picking both of the forks, he can start to eat. The order of picking the forks is not important.

• After eating enough, a philosopher puts the forks back on the table, and starts to think.

• Note that, a philosopher has to share the forks with the philosophers on the left and on the right.

• Therefore, a philosopher may not be able to pick the forks, if his neighbors are currently holding the shared forks.

• Note that, in this example, the forks stand for a shared resource, and the philosophers are entities competing for that shared resource.

• Consider a scenario where every philosopher picks the fork on his right (or left). If the philosophers are stubborn, and do not want to put a fork back on the table before eating something, then

philosophers will be in a deadlock, since none of them will ever get the other fork.

- Such a scenario would be possible if each philosopher implements the following pseudo code:

```
loop:
  pick the left fork;
  pick the right fork
  eat;
  put the left fork back onto the table;
  put the right fork back onto the table;
  think;
goto loop;
```

- Consider the set of philosophers that are not so stubborn. So when a philosopher picks a fork, and cannot pick the other fork, he places the fork back on the table. With these philosophers, we may have the following problem.

  Each philosopher picks the fork on his left. Since they will see that their right forks are not on the table, they would be polite to each other, and place their forks back onto the table. Then, after a while, they decide to eat again, and all them picks their left fork at the same time. Although, they are not completely blocked, they can pick and put their forks, nothing useful is actually happening. Such a situation is called a livelock.

  This scenario can be possible if the philosophers implement the following pseudo code:

```
loop :
  pick the left fork;
  if cannot pick the right fork then
     put the left fork back onto the table;
     goto loop;
  else
     pick the right fork;
  endif
  eat;
  put the left fork back onto the table;
  put the right fork back onto the table;
  think;
goto loop;
```

- In order these philosophers not to starve to death, they have to become more civilized and communicate to each other.

- In Ada, communication between concurrent tasks can be accomplished by rendezvous synchro-

nization using **entries**.

- It is similar to a procedure call.
- The caller is said to be the *client*
- The callee is said to be the *server*
- An entry in a task is a point where when the task reaches that point, it starts waiting until a client synchronizes with it.
- An example Ada program with tasks using entries is given below.

```
with text_io; use text_io;

procedure pqr is

  -- here is one task
   task P is
      entry Startp;
   end p;

  task body p is
  begin
     put_line ("p starts");
     accept startp do
        Put_Line("p");
     end Startp;
     put_line ("p ends");
  end p;

  -- here is another task
  task q is
      entry Startq;
   end q;

  task body q is
  begin
     put_line ("q starts");
     accept startq do
        Put_Line("q");
     end Startq;
     put_line ("q ends");
  end q;

begin -- of the procedure
```

```
   p.startp;
   Put_Line("r");
   q.startq;
end pqr;
```

• Note that, the entries must be declared in the task specification. The body of the entry is given within the body of the task.

• When the task p starts, it performs its execution until it reaches to a point where it accepts an entry. It waits at that point, until a client connects to that entry. And when a client connects to that entry, they would be synchronized.

• In order to see the task type declaration, and passing parameters to entries, let us rewrite the same example in the following way:

```
with text_io; use text_io;

procedure pqr_typed is

  -- here is a task type
  task type emitter is
     entry get_a_char(c : character);
  end emitter;

  -- declare tasks of this type;
  P, Q :  Emitter;

  task body emitter is
  begin
     put_line ("an emitter starts");
     accept get_a_char (c : character) do
        Put(c); New_Line;
     end get_a_char;
     put_line ("an emitter ends");
  end emitter;

begin -- of the procedure
   p.get_a_char('p');
   Put_Line("r");
   q.get_a_char('q');
end pqr_typed;
```

• The tasks can also be created dynamically as shown in the example below:

```
with text_io; use text_io;
```

```
procedure pqr_dynamic is

  task type emitter is
     entry get_a_char(c : character);
  end emitter;

  type Emitter_Ptr is access Emitter;

  P,Q : emitter_ptr;

  task body emitter is
  begin
     put_line ("an emitter starts");
     accept get_a_char (c : character) do
        Put(C); New_Line;
     end get_a_char;
     put_line ("an emitter ends");
  end emitter;

begin -- of the procedure
   P := new Emitter;
   Q := new Emitter;
   p.get_a_char('p');
   Put_Line("r");
   q.get_a_char('q');
end pqr_dynamic;
```

10.5 Conditional and unconditional selective acceptance

- Several entries can be waited for acceptance:

```
select
  when <guard> =>
    accept <entry name> do
      ...
    end <entry name>;
or
  when <guard> =>
    accept <entry name> do
      ...
    end <entry name>;
...
end select;
```

where <guard> is a boolean condition. The entries whose guard evaluates to true are waited for acceptance.

- It is also possible for unconditional selective acceptance:

```
select
    accept <entry name> do
      ...
    end <entry name>;
or
    accept <entry name> do
      ...
    end <entry name>;
...
end select;
```

- Conditional and unconditional selective acceptances can be used together.

10.6 Critical sections and semaphores

- A critical section of a program is a fragment of the program that must be executed atomically (without any interruption).

- The CS problem (aka mutual exclusion problem) can be solved by using a construct called a **semaphore**.

- A semaphore is a structure with an integer value, and two operations:

  - p operation: The value of the semaphore is decremented by one by an execution of a p operation. Sometimes the semaphore has a lower limit "low" which is used to bound the value of the semaphore from below. For such a semaphore, if the value of the semaphore is currently equal to low, it can not be decremented any more. A process requesting a p operation in this case (when the value is equal to low) has to wait until the value of the semaphore is incremented (possibly by another process having access to the semaphore).

  - v operation: The value of the semaphore is incremented by one by an execution of a v operation. Sometimes the sempahore has an upper limit "$high$" which is used to bound the value of the sempahore from above. For such a semaphore, if the value of the semaphore is currently equal to $high$, it can not be incremented any more. A process requesting a v operation in this case (when the value is equal to $high$) has to wait until the value of the semaphore is decremented (possibly by another process having access to the semaphore).

• If a semaphore has lower limit 0 and upper limit 1, then it is called a *binary semaphore*, since the value of such a semaphore can only be 0 or 1. Otherwise, it is called a *counting semahore*.

• Semaphores are not supported directly by Ada, but they can be implemented easily using entries. For example, the following can be used as a binary semaphore:

```
task type binary_semaphore is
  entry p;
  entry v;
end binary_semaphore;

task body binary_semaphore is
begin
  loop
    accept p;
    accept v;
  end loop;
end binary_semaphore;
```

• Note that `loop--endloop` can be used to implement an infinite loop.

• When the body of an entry is empty, a simple accept statement for the entry is enough.

• A general semaphore can be implemented as follows:

```
task type semaphore is
  entry p;
  entry v;
  entry initialize ( n : integer );
end semaphore;

task body semaphore is
  value : integer;
begin
  accept initialize (n : integer) do
    value = n;
  end initialize;

  loop
    select
      when value > 0 =>
         accept p do
           value := value - 1;
         end p;
```

```
    or
       accept v do
          value := value + 1;
       end v;
    end select;
  end loop;
end semaphore;
```

10.7 Synchronized Access to Shared Variables

- We will examine development of a producer–consumer example.
- There is a buffer which is implemented as a circular queue.
- The producer inserts elements into the buffer as long as the buffer is not full.
- The consumer removes elements from the buffer as long as the buffer is not empty.
- Initially, we will implement the buffer as an array shared by the two tasks.

```
with Text_Io; use Text_Io;

procedure Direct is

   Size : constant Integer := 5;
   Buf : array (0 .. Size - 1) of Character;
   Front, Rear : Integer := 0;

   function Notfull return Boolean is
   begin
      return ((Rear+1) mod size) /= Front;
   end Notfull;

   function Notempty return Boolean is
   begin
      return Front /= Rear;
   end Notempty;

   task Producer;
   task body Producer is
      c : character;
   begin
      while not End_Of_File loop
         if Notfull then
```

```
            Get(C);
            Buf(Rear) := C;              -- the critical section
            Rear := (Rear+1) mod Size;   -- of the producer
         end if;
      end loop;
   end Producer;

   task Consumer;
   task body Consumer is
      C : Character;
   begin
      loop
         if Notempty then
            C := Buf(Front);              -- the critical section
            Front := (Front+1) mod Size; -- of the consumer
            Put(C);
         end if;
      end loop;
   end Consumer;


begin
   null;
end direct;
```

• As can be seen from the bodies of the producer and the consumer, the buffer is directly accessed by these tasks.

• Note that, a shared resource should usually be accessed in a mutually exclusive way. In this example above, the shared resource is the buffer. The parts of the tasks that access to the buffer are the critical sections of them.

• If there are multiple concurrent programs with critical sections accessing the same shared resource, then it is a nice programming practice to allow only one of them to be in its critical section at any time.

• The example below uses a semaphore to let producer and consumer synchronize while going in and out their critical sections.

```
with Text_Io; use Text_Io;

procedure SyncDirect is

  Size : constant Integer := 5;
  Buf : array (0 .. Size - 1) of Character;
```

```
Front, Rear : Integer := 0;

task Binary_Semaphore is
  entry P;
  entry V;
end Binary_Semaphore;

task body Binary_Semaphore is
begin
  loop
    accept P;
    accept V;
  end loop;
end Binary_Semaphore;

function Notfull return Boolean is
begin
  return ((Rear+1) mod size) /= Front;
end Notfull;

function Notempty return Boolean is
begin
  return Front /= Rear;
end Notempty;

task Producer;
task body Producer is
  c : character;
begin
  while not End_Of_File loop
    if Notfull then
       Get(C);
       Binary_Semaphore.P;          -- Since after P is accepted, V has
       Buf(Rear) := C;              -- to be accepted before another P
       Rear := (Rear+1) mod Size;   -- can be accepted, CS is guaranteed
       Binary_Semaphore.V;          -- to be mutually exclusive.
    end if;
  end loop;
end Producer;

task Consumer;
task body Consumer is
  C : Character;
begin
  loop
    if Notempty then
```

```
         Binary_Semaphore.P;           -- Since after P is accepted, V has
         C := Buf(Front);              -- to be accepted before another P
         Front := (Front+1) mod Size;  -- can be accepted, CS is guaranteed
         Binary_Semaphore.V;           -- to be mutually exclusive.
         Put(C);
      end if;
    end loop;
  end Consumer;

begin
   null;
end SyncDirect;
```

• Note that the code above will suffer from **busy waiting**. In other words, when the buffer is full for example, the producer will iterate in its while loop, without ever going into the then part of its if statement, until the consumer consumes a slot from the buffer.

• Similarly, the consumer will suffer from busy waiting problem, when the buffer is empty.

• Since they will be doing nothing useful but heating up the CPU while they are busy waiting, it would be nice to be able to stop their iterations.

• The following code solves the problem of busy waiting by using an additional semaphore.

```
with Text_Io; use Text_Io;

procedure SyncDirectNoBusyWaiting is


   Size : constant Integer := 5;
   Buf : array (0 .. Size - 1) of Character;
   Front, Rear : Integer := 0;

   task type Binary_Semaphore is
      entry P;
      entry V;
   end Binary_Semaphore;

   task body Binary_Semaphore is
   begin
      loop
         accept P;
         accept V;
      end loop;
   end Binary_Semaphore;
```

```
task type Semaphore is
   entry Init (N : Integer);
   entry P;
   entry V;
end Semaphore;

task body Semaphore is
   Value : Integer;
begin
   accept Init (N : Integer) do
      Value := N;
   end Init;
   loop
      select
         when Value > 0 =>
         accept P do
            Value := Value - 1;
         end P;
         or
         accept V do
            Value := Value + 1;
         end V;
      end select;
   end loop;

end Semaphore;


Filling, Emptying : Semaphore;
Critical : Binary_Semaphore;

task Producer is
   entry Init;
end Producer;

task body Producer is
   c : character;
begin
   accept init;
   while not End_Of_File loop
      Filling.P;
      Get(C);
      Critical.P;
      Buf(Rear) := C;
      Rear := (Rear+1) mod Size;
```

```
         Critical.V;
         Emptying.V;
      end loop;
   end Producer;

   task Consumer is
      entry Init;
   end Consumer;

   task body Consumer is
      C : Character;
   begin
      accept init;
      loop
         Emptying.P;
         Critical.P;
         C := Buf(Front);
         Front := (Front+1) mod Size;
         Critical.V;
         Put(C);
         Filling.V;
      end loop;
   end Consumer;
begin
   Filling.Init (Size);
   Emptying.Init (0);
   Consumer.Init;
   Producer.Init;
end SyncDirectNoBusyWaiting;
```

10.8 Using Monitors

A monitor is a set of procedures together with a set of associated data structures. It actually corresponds to the data encapsulation concept in object oriented paradigm. The data of a monitor should only be accessed by the procedures of the monitor. The tasks call the procedures to access the data of the monitor.

One important property of a monitor is only one of the procedures of the monitor can be active at a given time. Hence it provides a solution to the mutual exclusion problem naturally.

```
monitor buffer is
  -- buffer declarations here

  procedure add (c : in character);
  begin
    if buffer full then wait(adding);
```

```
    add c into buffer;
    signal(removing);
  end add;

  procedure remove (c : out character);
  begin
    if buffer empty then wait(removing);
    c := next char in buffer;
    signal(adding);
  end remove;

begin
  initializations ...;
end buffer;
```

Since only one of the procedures of the monitor procedures can be active at any given time, if during the execution of a procedure, the execution gets stuck, then this would block all future calls to the procedures of the monitor.

The `wait` and `signal` concepts can be used to avoid such cases. If a process/task `P` is currently executing a procedure of a monitor and if it sees that it will have wait (e.g. producer started executing the procedure of monitor , then it can call `wait(q)` to suspend its execution. The parameter of `wait`, which is `q` here, is the name of a queue that accumulates the list of suspended processes/tasks. There can be more than one such queue in a monitor. After executing the `wait` statement and putting the process/task on a queue, the monitor is ready to accept another call for one of its procedures (which might be a different procedure than the last one called).

How do we resume the executions of those processes/tasks that are placed in the queues? When the monitor executes `signal(q)` statement, the execution of the first process/task in the queue `q` is resumed. If the queue `q` is currently empty, then `signal(q)` will have no effect.

10.9 Using a Dedicated Process

```
task body buffer is
  buffer declarations;
begin
  loop
    select
      when not full =>
         accept addchar (c : in character) do
             ...
         end addnew;
    or
      when not empty =>
          accept removechar (c : out character) do
             ...
          end removechar;
```

```
    end select;
  end loop;
```

Chapter 11

Logic Programming

• If functional programming is to be considered as programming with functions, then logic programming is programming with relations.

• Programming with relations is considered to be more flexible than programming with functions. The reason is that, with a function we have an explicit direction of computation. If we consider the addition function, then application of this function will always produce the addition of two numbers. For example, we would ask "what is the addition of 2 and 3" and we would get the result 5.

• However, the addition function can also be modelled as a relation, as a 3-tuple:

```
(op1, op2, result)
```

For any two numbers `x` and `y`, the tuple `(x, y, x+y)` would be in this tuple, e.g. `(2, 3, 5)`.

• With such a relation, we can still find out the addition of two numbers `x` and `y`, by finding the tuple of the relation which is in the form `(x, y, z)`. The last element of the tuple, `z` would give us the result of adding `x` and `y`. For example, to find out the addition of `2` and `3`, we should search the elements of the relatio for a tuple of the form `(2, 3, ?)`. Since `(2,3,5)` would be the only tuple in the relation that matches this query, the last element, `5`, would be the addition of `2` and `3`.

• However for a tuple, all the elements can be interpreted in the same way. I.e., we can actually compute the result of the question: "what is the number that should be added to 3 in order to get the result 5?".

• In such a case, we should look for a tuple of the form `(?, 3, 5)` in the relation. The first element of the tuple matching this template would give us the answer.

• As can be seen from the example above, working with relations will blur the direction of computation, i.e. the distinction between the operands and the result of an operation will disappear.

• Logic Programming is linked to a programming language called PROLOG (PROgramming in

LOGic).

• There are various dialects derived from the original version in 1972. We will study Edinburgh Prolog.

11.1 Relations

• Conceptually, a relation is a (possibly infinite) collection of tuples of the form

$$\begin{array}{ll} \{ & (\mathtt{a}_1^1, \mathtt{a}_2^1, \ldots, \mathtt{a}_n^1) \\ & (\mathtt{a}_1^2, \mathtt{a}_2^2, \ldots, \mathtt{a}_n^2) \\ & (\mathtt{a}_1^3, \mathtt{a}_2^3, \ldots, \mathtt{a}_n^3) \\ & \ldots \qquad\qquad\qquad\quad \} \end{array}$$

• n is the arity of such a relation, and the relation is called an *n–ary relation*.

• For example, consider the addition relation example above. This relation will be the infinite set that has the tuples:

$$\{\ \texttt{(0,0,0)},\ \texttt{(0,1,1)},\ \texttt{(1,0,1)},\ \texttt{(1,1,2)},\ \ldots\ \}$$

• An n–ary relation can also be viewed as a table with n columns, and a row for each tuple of the relation.

0	0	0
0	1	1
1	0	1
1	1	2
0	2	2
2	0	2
⋮	⋮	⋮

• A relation is also called a *predicate* because the use of a relation $\mathcal{R}$ often is in the form :

Is a given tuple in the relation $\mathcal{R}$?

(producing a boolean answer, and as you may remember from our discussions on function programming, an operation with a boolean result is called a predicate)

11.2 Your first program in Prolog

• What programming (in general) is all about is modelling the real life in computers and looking for the answers of the questions which can not be (at least easily) found by a human being.

• The modelling part is the activity of coding the rules of the real life, sometimes by bending them a little, or by abstracting them, into the computer.

• After creating a model of the real world, we run our program to ask "given this as a model of the real world, what would happen in the real world, under such and such conditions".

• To visualize these activities consider the problem of travelling salesman.

• We create the data structures to hold the cities and the distances between the cities, and try to create a model of how our brain would solve the problem on a given map.

• Then we execute our program by providing an input map of cities, to find the answer that otherwise our brain would find, if we, ourselves solved the problem.

• Programming in a logic programming language is no different.

• In fact, in a logic programming language, it is much more explicit that while writing a program, we are creating a model of the real world.

• A logic programming language works as based on deductions as our brain does, hence it is much more closer to the way we think.

• Programming in Prolog is performed by declaring relations, and later querying these relations.

• As the first example, let us declare the following relation:

```
female(ayse).
```

• With this statement, we have declared a relation named `female`, and inserted a tuple (actually a singleton) `ayse` into this relation.

• Note the dot at the end of the statement.

• We can insert more tuples into a relation previously defined as:

```
female(zehra).
female(alice).
```

• With these declarations, we are creating a knowledge base, which models the real world (or part

of it, that is sufficient for the purposes of the answers that we will be looking for).

- After such declarations for developing the knowledge base, we can ask questions. For example:

```
?- female(ayse).
```

- Note that, queries in Prolog (and in this document) are prefixed by `?-`.
- With this query, we are asking if `ayse` is in the relation `female`.
- The actual interaction with SWI Prolog would be as follows:

```
?- female(ayse).

Yes
```

- In other words, it would respond by saying "Yes", meaning that `ayse` is in the relation `female`.
- Let us interact a little bit more:

```
?- female(fatma).

No
```

- As can be seen, although `fatma` is a female name, it says "No". This is because we did not introduce `fatma` into the relation `female`.
- Note that, the world we create in Prolog is limited with the information we provide to it. It knows nothing more.
- Furthermore, the relations we introduce has nothing to do with the actual meanings of those relations. For example, consider the following declarations:

```
like(zehra,ayse).
like(ali,veli).
like(ali,ali).
like(fatma, veli).
```

- The relation `like` defined with these statements can represent the meaning of like as in "`zehra` *like*s `ayse`" or it can represent the meaning of like as in "`zehra` looks *like* `ayse`".
- Prolog has no way distinguishing between these two meanings of "like". The actual meanings of the relations is only in the programmer's mind.
- Note that, the type of queries we have seen so far are not so interesting. Going back to the

example of the addition relation, the queries we have seen above in the nature of asking if "Is 2+3 equal to 5?", i.e. is the tuple `(2,3,5)` in the addition relation?

- It will be more interesting to ask "what is the result of adding 2 and 3", or "who is female", or "who likes veli".

- Asking such questions is also possible in Prolog. For example,

```
?- like(Who,veli).
```

is the query that asks who likes `ali`, or more technically is there a pair in the relation `like` such that the second element of the pair is `veli`.

- Note that, syntactically

```
?- like(Who,veli).
```

and

```
?- like(ali,veli).
```

look very much alike.

- The query

```
?- like(ali,veli).
```

as we have seen, asks if the pair `(ali,veli)` is in the relation `like` or not. So, why does not the statement

```
?- like(Who,veli).
```

interpreted as asking if the pair `(Who,veli)` is in the relation `like` or not.

- Actually, there is a syntactic difference between these two statements. Unlike `ali`, `Who` starts with a capital letter. And any identifier that starts with a capital letter is a variable in Prolog, that should be substituted for a value.

- Note that, `Who` is not a special word in Prolog, it is the capital `W` at the beginning of it, that makes it a variable.

- We could have performed the same query by:

```
?- like(Kim,veli).
```

- The full interaction would be as follows:

```
?- like(Kim,veli).

Kim = ali
```

meaning that, when `ali` is substituted for `Kim`, the query holds.

- At this point, it will start waiting for an input from the user. If we press enter, it will do the following:

```
?- like(Kim,veli).

Kim = ali

Yes
?-
```

and start waiting for the next query. "Yes" here means that it were able to find a value for "Kim".

- If you press semicolon character `;` at this point, it will discard the substitution of `ali` for `Kim`, and will look for another substitution. In this case, the full interaction would be:

```
?- like(Kim,veli).

Kim = ali;

Kim = fatma
```

Again, pressing enter would complete the interaction as

```
?- like(Kim,veli).

Kim = ali;

Kim = fatma

Yes

?-
```

- However, pressing `;` one more time would produce

```
?- like(Kim,veli).

Kim = ali;

Kim = fatma;

No
?-
```

Meaning that no more substitution could be found.

• As we have mentioned before, programming with relations have the advantage of being directionlessness (if there is such a word). The previous query using the relation `like` asked "who likes veli". We can take the advantage of directionlessness of relations, and ask also "who is liked by ali".

```
?- like(ali,X).

X = veli;

X = ali;

No
?-
```

• Note that, by directionlessness, we mean the direction of computation. Direction of relation of course still exist. That is,

```
?- like(veli,X).

No
?-
```

`ali` may like `veli` as dictated by the `(ali,veli)` pair we have put into the relation `like`. However, this does not mean that `veli` likes `ali`, since there is no pair in the relation whose first element is `veli`.

11.3 Rules & Facts

• We have stated previously that, the working principle of Prolog is tries to mimic the working principal of our brains.

• However, as human beings, we do not store huge database of relations for the entire knowledge base we acquire. Rather than that, we try to identify the rules behind the relations, and try to use logical deductions in order to get concrete results out of these symbolic relations.

• When we perform the logical derivation, "all men are mortal, socrates is a men, therefore, socrates is a mortal", we are actually using two relations `men` and `mortal`. After knowing that `men(socrates)`, we are deducing the result `mortal(socrates)`, by using a logical inference.

• Likewise, it is not suitable to declare a relation in Prolog by explicitly listing all the tuples in the relation.

• Instead, a relation can be defined by giving some rules based on the other relations.

• In pseudo syntax, a rule is given as follows:

$$P \text{ if } Q_1 \text{ and } Q_2 \text{ and } \ldots Q_k$$

which can be read in both directions as

- if Q_1 and Q_2 and $\ldots$ Q_k are all true, then so is P; or
- to deduce P, Q_1 and Q_2 and $\ldots$ Q_k must all be true

- Such rules are called *Horn Clauses* (introduced by Alfred Horn, 1951).

- A *fact* is a special case of a rule where $k = 0$. In such a case, P will hold trivially, without depending on the validity of any other thing.

11.4 A more detailed introduction to Prolog

- A Horn Clause of the form

$$P \text{ if } Q_1 \text{ and } Q_2 \text{ and } \ldots Q_k$$

is given in Prolog as

`P :- Q1, Q2, ..., Qk .`

which is also called a rule.

- `P` is called the *head* of the rule, `:-` is called the *neck* of the rule, and the rest is called the *body* of the rule.

- A fact in Prolog is given as

```
P.
```

A fact is a rule without a neck and body.

- `P`, `Q1`, `Q2`, ..., `Qk` are called *terms*.

- A term is either a constant, or a variable, or has the form `rel(T1, T2, ..., Tn)` where each `Ti` is a term.

- Variables must start with a capital letter.

- Relations and constants must start with a lowercase letter.

- An example of a fact rule would be:

```
female(ayse).
```

- An example of a normal rule would be:

```
mortal(X):-man(X).
```

- A program consists of two parts: rules and queries.
- Rules defines facts and inference rules to deduce new facts based on the already known facts.
- An example rules part:

```
father(ali,hasan).
father(ayse,hasan).
father(hasan,osman).
father(fatma,zeki).
mother(fatma,turkan).
mother(ali,fatma).
mother(ayse,fatma).
mother(hasan,seval).
sibling(X,Y):-father(X,Z),father(Y,Z).
sibling(X,Y):-mother(X,Z),mother(Y,Z).
parent(X,Y):-father(X,Y).
parent(X,Y):-mother(X,Y).
```

- In the query part, we ask questions on the relations we have defined.
- The general form of a query is :

$$\mathtt{T}_1\mathtt{,T}_2\mathtt{,}\ldots\mathtt{,T}_n$$

- An example query part:

```
?- sibling(ali,X),father(X,Y).
```

- Such a query asks for the following: (performs an existential search)

  Do there exist `X` and `Y` (all the variables used in the query), such that `ali` is sibling of `X` and `Y` is father of `X`?

- The reading of rules is slightly different. For example, the rule

```
sibling(X,Y):-father(X,Z),father(Y,Z).
```

reads as (using a universal quantifier)

> For all `X` and `Y` (all the variables in the head), if there exists `Z` (all the variables in the body except the variables in the head), such that `Z` is father of `X` and `Z` is father of `Y`, then `X` is a sibling of `Y`.

• Note that, for two people to be siblings we have considered a common father *or* a common mother to be enough. In order to give this "*or*", we have two separate rules.

• Let us try out the following query:

```
?- sibling(ali,X).
```

```
X=ali
```

As you see, the Prolog interpreter reports that `ali` is a sibling of `ali`, or in other words `(ali, ali)` is a tuple in the relation `sibling`, which is perfectly correct with respect to the rules we have provided. Prolog cannot know that someone cannot be a sibling of himself/herself, unless we tell it so.

• Let us rewrite the sibling relation in order to teach this to Prolog:

```
sibling(X,Y):-father(X,Z),father(Y,Z),not(X=Y).
sibling(X,Y):-mother(X,Z),mother(Y,Z),not(X=Y).
```

11.5 Inferencing in Prolog

• A query in Prolog is called as a *goal*.

• The components within a goal are called the *subgoals*.

• For example, for the goal (query):

$$\texttt{?- T}_1\texttt{, T}_2\texttt{, ..., T}_n$$

each $\texttt{T}_i$ is a subgoal.

• A goal is *satisfied* if there exist variable bindings that makes the query true.

• There is no semantic difference between a goal and a subgoal. The only distinction is that, the subgoals must be satisfied for the goal to be satisfied.

• For example, consider the goal:

```
?- mother(X,Y), father(X,Y).
```

• In real world, at least with the current state of the art in science, this is not possible (provided that `mother` and `father` relations correspond to the biological motherhood and biological fatherhood respectively) . Someone cannot be both father and mother to someone.

• Although, it is feasible to define the relations `father` and `mother` in such a way that the goal above is satisfiable, with the definitions of these relations given above, this goal is not satisfiable.

• Note that, each of the subgoals `mother(X,Y)` and `father(X,Y)` are satisfiable.

• For example, `X=ali, Y=fatma` bindings satisfy the subgoal `mother(X,Y)` and `X=ali, Y=hasan` bindings satisfy the subgoal `father(X,Y)`.

• However, in a goal, variable bindings must be consistent across the subgoals.

• As can be noted, even if the subgoals are satisfiable separately, the entire goal may not be satisfiable.

11.6 Lists in Prolog

• Lists, as in the case of Scheme, are again a fundamental data structure for Prolog.

• Lists in Prolog are displayed by enclosing the elements between a pair of square brackets and separating the elements of the lists by comma, e.g.

```
[ a, b, c ]
```

• The empty list is written as `[ ]`

• The head and trailing parts of a list can be separated by `|`. For example all of the following denotes the same list:

```
[ a, b, c ]
[ a, b, c | []]
[ a, b | [ c ]]
[ a | [ b, c ]]
```

• You can consider `|` as an operator, where to the left of it there has to be a list of elements (let's call it left list), and to the right of it, there has to be a single list (let's call it right list). Then `|` can be considered as an operator that appends (in the sense of Scheme) the left list and the right list. For example:
`[ a, b | [ c, d ]]` is the same list as `[ a, b, c, d]`
`[ [ a, b ] | [ c, d ]]` is the same list as `[ [ a, b], c, d]`

`[ a, b | c, d ]` is the same list as `[ a, b, (c, d)]`

- `|` has a slightly different usage in queries which will be described later.

11.7 Unification

- Deduction in Prolog is based on the concept of unification.

- For example assume that, we have the following fact in our knowledge base:

```
identity(X, X).
```

and we ask for a solution of the following query

```
?- identity(f(a,Y),f(Z,b)).
```

- Note that, the fact says two terms are in the relation `identity` if they are the same terms.

- For this purpose, Prolog will try to make `f(a,Y)` and `f(Z,b)` the same terms.

- This will be achieved if `Y` is bound to `b` and `Z` is bound to `a`, in which case both the first element and the second element in the query will become `f(a,b)`. Hence the Prolog will respond by "yes when `Y=b, Z=a`" to our query.

- The process of trying to make two terms the same is called the *unification*.

- Note that, while unifying two terms T_1 and T_2, if a variable occurs in both T_1 and T_2, then that variable must be substituted by the same value in both T_1 and T_2.

- The symbol `=` is used for unification in Prolog.

- Hence the query

```
?- f(a,Y)=f(Z,b)
```

will perform the same unification as above.

- Contrary to most of the other programming languages, it should not be considered as assignment.

- It can be thought as an assignment when the left hand side and the right hand side terms use disjoint set of variables. In this case, it will work as an assignment in two directions, i.e. both the variables on the left hand side (e.g. `Y` in the example given above) and the variables on the right hand side (e.g. `Z` the example given above) will be bound to some values.

- However, you should always keep in mind that `=` operator tries to find a unification by binding a consistent value to variables across the entire query. Otherwise, you'll have a hard time understanding why the query `?- X = X + 1.` reports

```
X = ...  +...  +1+1+1+1+1+1+1+1+1
```

This is because, these two terms `X` and `X+1` can be unified only if `X` is bound to this value (infinite sum of 1's).

- Revisiting the `|` operator in lists, let us explain the semantics of this operator in queries.

- The term `[H|T]` (where `H` and `T` are made up names) when unified with a list, will cause `H` to be bound to the first element of the list, and `T` to be bound to the rest of the list. In other words, `H` will correspond to `car` procedure of Scheme, and `T` will correspond to the `cdr` procedure of Scheme.

- For example:

```
?- [H|T] = [a,b,c].

H = a
T = [b, c]

Yes
?- [X|Y] = [a].

X = a
Y = []

Yes
```

- With this special usage of `|` we do not actually need special functions (or relations to use the Prolog terminology) for `car` and `cdr` procedures.

11.8 How does it work?

- When a query is performed, all the subgoals of the query are tried to be satisfied from left to right. For example in the following query:

```
?- member(X,Y), member(X,Z).
```

First `member(X,Y)` will be tried to be satisfied, and then `member(X,Z)` will be tried to be satisfied.

- Of course, satisfaction of `member(X,Y)` will substitute values for `X` and `Y`. Since `X` is used in the second subgoal, this value of `X` will be used while trying to satisfy the second subgoal.

- Note that, there may be multiple possible substitutions for `X` and `Y` to satisfy the first subgoal `member(X,Y)`. After finding a satisfying substitution for the variables of the first subgoal, the second subgoal will be tried with these values. If the second subgoals are not satisfiable with these subgoals,

then a *backtracking* will occur (i.e. we will go back and try to find another satisfying solution for the first subgoal), and the second subgoal will be retried with the new substitutions (if exists).

- Satisfaction of a goal is performed by unifying it with a fact or a (head part of a) rule that is present in the knowledge base.

- If a goal can be unified with a fact, then it is directly satisfied.

- If a goal is unified with the head part of a rule, then the goal is not considered to be satisfied immediately. For it to be satisfied, the body of the rule is considered to be new subgoals to be satisfied.

- The facts and the rules in the knowledge base are considered for unification in the order they appear in the knowledge base (which is the same order they are entered in the rules part of a Prolog program).

- If no fact or a rule can be unified, then the goal is reported to be unsatisfiable.

- Note that, when the entire query is satisfied, the substitutions for the variables that appear in the query will be displayed. If the user presses `;` at this point, this solution will be considered as not acceptable, and a backtracking will occur to find other substitutions for the variables of the query.

- Consider the following example. We want to write a binary relation `member` whose first element is a term, and second element is a list that contains the first element.

- The following (one fact and one rule) is sufficient to describe such a relation.

```
member(H, [H|T]).
member(H,[X|T]):-member(H,T).
```

- Assume we tried the following query:

```
?- member(1, [1,2]).
```

- First the fact (since it is given before the rule) will be tried to be unified with this query. Unification will be successful with the substitutions `H=1` and `T=[2]`. And since this is a fact, the query will also reported as holding with these substitutions.

- However, assume we tried the following query:

```
?- member(2, [1,2]).
```

- Again, first the fact will be tried to be unified with this query. However, there is no way to instantiate `H` and `T` in the fact such that it is unified with the term `member(2, [1,2])`. Hence the

fact will fail.

• Next, the head of the rule (i.e. `member(H,[X|T])` will be tried to be unified with the query term. We will be able to unify them when `H=2`, `X=1` and `T=[2]`. However, the goal would not be satisfied immediately at this point. We will have to satisfy the body of the rule, which with the current substitutions of the variables is : `member(2,[2])`.

• This goal `member(2,[2])` will be satisfied by the fact during the second search started as explained above for the goal `member(1,[1,2])`.

• Hence the entire query `?- member(2,[1,2])` will be reported as satisfiable.

• If we consider again the knowledge base for this example:

```
member(H, [H|T]).
member(H,[X|T]):-member(H,T).
```

- You can read the fact as:
  Relational view: "If the element we are looking for is the same as the head of the list, then this pair should be in the relation"
  Functional view: "If the element we are looking for is the same as the head of the list, then this element is in the list"

- You can read the rule as:
  Relational view: "If the element we are looking for is not the same as the head of the list (not because we have used two different variable names `H` and `X` but because the fact given before this rule has failed), then the current pair is in the relation if the element we are looking for, and the rest of the list is in the relation.
  Functional view: "If the element we are looking for is not the same as head of the list, then the element is in the list if it is in the rest of the list".

11.9 Anonymous Variables

• Sometimes, we do not care about the substitutions for a variable.

• For example consider the example of `member` relation given above one more time:

```
member(H, [H|T]).
member(H,[X|T]):-member(H,T).
```

• When a goal is being unified with the fact `member(H, [H|T])`, we do not actually care about the substitution for `T`. We are only interested on the equality of the element and the head of the list.

• For such cases, the symbol _ can be used as

```
member(H, [H|_]).
member(H,[X|T]):-member(H,T).
```

• Similarly, within the rule (being sure that the head of the list is different than the element we are looking for), we do not care about the head of the list. Hence we can further change the knowledge base to

```
member(H, [H|_]).
member(H,[_|T]):-member(H,T).
```

11.10 Programming Techniques

• Note that, knowing how subgoals are handled in a query by unification and backtracking, you can increase the efficiency of your programs.

• A powerful programming technique for Prolog is the use of *guess–and–verify* or *generate–and–test* rules.

• Such rules have their bodies conceptually separated into two: a guess (generate) part and a verify (test) part.

• This programming technique is based on the fact that Prolog will try to satisfy the subgoals in a query from left to right, and the substitutions of variables found by a subgoal will be used by the subgoals that appear to the right of it.

• Hence, if a subgoal have a small number of substitution possibilities for satisfiability, it is wiser to put it as close as possible to the neck of the rule. This will restrict the possible values for the variables of this subgoal that are shared with the subgoals that appear to the right.

• For example assume that we want to define a binary relation named `overlap` on two lists. Two lists will be in this relation if they have a common element.

• The following knowledge base can be used to declare this relation:

```
member(H, [H|_]).
member(H,[_|T]):-member(H,T).
overlap(X,Y):-member(M,X),member(M,Y).
```

• Note that, the body of the rule for `overlap` is in guess–and–verify style. The first subgoal keep guessing possible substitutions for `M`, and those substitutions for `M` will be verified by the second subgoal of the body.

• Consider the query

```
?- overlap([a,b,c,d],[1,c]).
```

• This query will first be unified with the head of the only rule for `overlap` which will end up in the requirement of satisfaction of the following query:

```
?- member(M,[a,b,c,d]),member(M,[1,c]).
```

• After this point:
`a` will be generated and tested (it will fail)
`b` will be generated and tested (it will fail)
`c` will be generated and tested and `c` will be reported as a solution
If continued (by pressing `;`) `d` will be generated and tested (it will fail)

• To exaggerate the importance of ordering of subgoals, consider the following query:

```
?- X=[1,2,3], member(a,X).
```

• The first subgoal (which is only a unification) has one possible substitution which is `X=[1,2,3`. Hence the second subgoal will only be tried with this substitution as `member(a,[1,2,3])`, and the query will fail.

• However consider the following slightly modified form of the same query:

```
?- member(a,X),X=[1,2,3].
```

• The first subgoal now have infinitely many solutions (all lists that include `a` in it is a solution for the first subgoal) for `X`.

• For each of such lists, the second subgoal will be tried and it will fail since none of them will be the same list as `[1,2,3]`.

Appendix A

ACM Press Release for the Announcement of 2005 Turing Award Winner

Copied verbatim from "http://www.acm.org/2005_turing_award/" on March 03, 2006

SOFTWARE PIONEER PETER NAUR WINS ACM'S TURING AWARD

Dane's Creative Genius Revolutionized Computer Language Design

New York, March 01, 2006 – The Association for Computing Machinery (ACM) has named Peter Naur the winner of the 2005 A.M. Turing Award. The award is for Naur's pioneering work on defining the Algol 60 programming language. Algol 60 is the model for many later programming languages, including those that are indispensable software engineering tools today. The Turing Award, considered the "Nobel Prize of Computing" was first awarded in 1966, and is named for British mathematician Alan M. Turing. It carries a $100,000 prize, with financial support provided by Intel Corporation.

Dr. Naur was editor in 1960 of the hugely influential "Report on the Algorithmic Language Algol 60." He is recognized for the report's elegance, uniformity and coherence, and credited as an important contributor to the language's power and simplicity. The report made pioneering use of what later became known as Backus-Naur Form (BNF) to define the syntax of programs. BNF is now the standard way to define a computer language. Naur is also cited for his contribution to compiler design and to the art and practice of computer programming.

"Dr. Naur's ALGOL 60 embodied the notion of elegant simplicity for algorithmic expression," said Justin Rattner, Intel senior fellow and Chief Technology Officer. "Over the years, programming languages have become bloated with features and functions that have made them more difficult to learn and less effective. This award should encourage future language designers who are addressing today's biggest programming challenges, such as general-purpose, multi-threaded computation, to achieve that same level of elegance and simplicity that was the hallmark of ALGOL 60."

Contributions Signal Birth of Computing Science

In 2002, former Turing Award winner Edsger Dijkstra characterized the development of Algol 60 as "an absolute miracle" that signaled the birth of what he called "computing science" because it showed the first ways in which automatic computing could and should become a topic of academic concern. The development of Algol 60 was the result of an exceptionally talented group of people, including several who were later named Turing Award winners.

Dr. Naur's contribution to Algol 60, was seminal. John Backus, another former Turing Award winner, acknowledged Naur as the driving intellectual force behind the definition of Algol 60. He commented that

Naur's editing of the Algol report and his comprehensive preparation for the January 1960 meeting in which Algol was presented "was the stuff that really made Algol 60 the language that it is, and it wouldn't have even come about, had he not done that."

Before publication of the Algol 60 Report, computer languages were informally defined by their prose manuals and the compiler code itself. The report, with its use of BNF to define the syntax, and carefully chosen prose to define the semantics, was concise, powerful, and unambiguous.

The 17-page Algol 60 Report presented the complete definition of an elegant, transparent language designed for communication among humans as well as with computers. It was deliberately independent of the properties of any particular computer. The new language was a major challenge to compiler writers. Dr. Naur went on to co-author the GIER Algol Compiler (for the transistorized electronic computer developed in Denmark known as GIER), one of the first compilers to deal fully and correctly with the language's powerful procedure mechanism.

"Dr. Naur's contribution was a watershed in the computing field, and transformed the way we define programming languages," said James Gray of Microsoft Research, and Chair of the 2005 Turing Committee. "Many of the programming constructs we take for granted today were introduced in the Algol Report, which introduced a concise block-structured language that improved the way we express algorithms."

Dr. Naur was instrumental in establishing software engineering as a discipline. He made pioneering contributions to methodologies for writing correct programs through his work on assertions that enable programmers to state their assumptions, and on structured programming. "His work, though formal and precise, displays an exceptional understanding of the limits and uses of formalism and precision," said Gray. Through these activities, and his development of an influential computer science curriculum, Dr. Naur contributed fundamental components of today's computing knowledge and skills.

Early Experience in Practical Calculations and Applications

Dr. Naur began his scientific pursuits as an astronomer, where he was involved in computations of the orbits of comets and minor planets. He obtained a magister of science degree (the equivalent of a master's degree) from Copenhagen University in 1949. He later returned there to earn a doctorate in astronomy in 1957. During the 1950-51 academic year, Dr. Naur studied astronomy at King's College in Cambridge, U.K., and came to the U.S. to further his work in the field. This work involved using early computers (starting with EDSAC, the world's first practical stored program electronic computer) for his astronomical calculations. In 1953, he returned to Denmark and served as a scientific assistant at Copenhagen Observatory.

In 1959, he joined the staff of the compiler design group at Regnecentralen, the first Danish computer company. There he organized the Algol Bulletin and was editor of the 13-person international Algol 60 team's report that defined Algol 60. He became a professor at the Copenhagen University Institute of Datalogy in 1969, retiring in 1998.

Dr. Naur was awarded the G. A. Hagemann Gold Medal from the Danish Technical University in 1963, the Jens Rosenkjaer Prize from the Danish Radio in 1966, and the Computer Pioneer Award from the Institute of Electrical and Electronics Engineers in 1986. ACM will present the Turing Award at the annual ACM Awards Banquet on May 20, 2006, at the Westin St. Francis Hotel in San Francisco, CA.