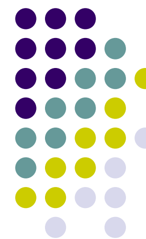
System Software

OS managing resources
Processor management & the need for scheduling
Memory management (paging & virtual memory)
Disk thrashing & Virtual machine
Stages in the compilation of a program
Syntax diagrams & Backus-Naur Form (BNF) notation
Reverse Polish Notation (RPN)



Purposes of operating systems

The operating system (OS) must provide:

- Manage all hardware resources
- Interface user and machine
- Interface applications software and machine
- Data security
- Utility software

Purposes of operating systems

The operating system (OS) must provide:

- management of all hardware resources:
  - processor
  - secondary storage filing system
  - input/output devices
- an interface between the user and the machine
- an interface between applications software and the machine
- security for the data on the system
- utility software to allow maintenance to be done

Purposes of operating systems

- A computer system needs a program that begins to run when the system is first switched on
- The operating system programs are stored on disk so there is no operating system
- The Basic Input Output System (BIOS) which starts a bootstrap program is stored in ROM
- The bootstrap program loads the operating system into memory and sets it running

Purposes of operating systems

- An operating system can provide facilities to have more than one program stored in memory
- However, only one program accesses the CPU at any given time; the rest are waiting
- For single user system ➔ **multi-programming**
- For multi user system ➔ **time-sharing system**

- OS can be considered from two viewpoints; an internal viewpoint and an external viewpoint
- The internal viewpoint focuses on the best usage of available resources
- The external viewpoint focuses on the facilities made available for system usage

Purposes of operating systems

Multi-tasking

- Processor works much faster than the human user
- Store more than one program in memory at the same time to make full use of the processor,
- The processor gives time to each of these programs in turn

The “boot up” process

Step 1:

- Run the power-on-self-test (POST) routine
- The POST routine resides in permanent memory (ROM)
- It clears the registers in the CPU and loads the address of the first instruction in the boot program into the program counter (PC) register

The “boot up” process

Step 2:

- Run the boot program, which first checks itself and the POST program
- The boot program is stored in read-only memory (ROM) and contains the basic input/output system (BIOS) structure
- The CPU then sends signals to check that all the hardware is working properly
- This includes checking the buses, system clock, RAM, disk drives and keyboard

The “boot up” process

Step 3:

- The boot program looks for an OS on the available drives
- If no OS is found, an error message is produced
- Once found, the boot program looks for files which contain the kernel - the core part of the OS
- The prime task of the kernel is to act as the interrupt handler

The “boot up” process

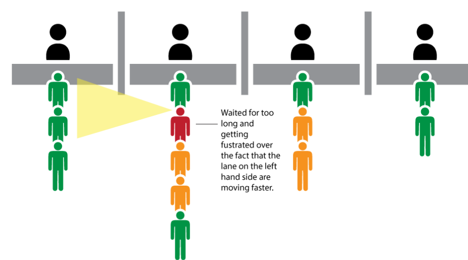
Step 4:

- The OS searches the root directory on the disk for a boot file (such as CONFIG.SYS) that contains instructions to load various device drivers
- A file (such as AUTOEXEC.BAT) is then loaded to ensure that the computer starts with the same configuration each time it is switched on

Purposes of operating systems

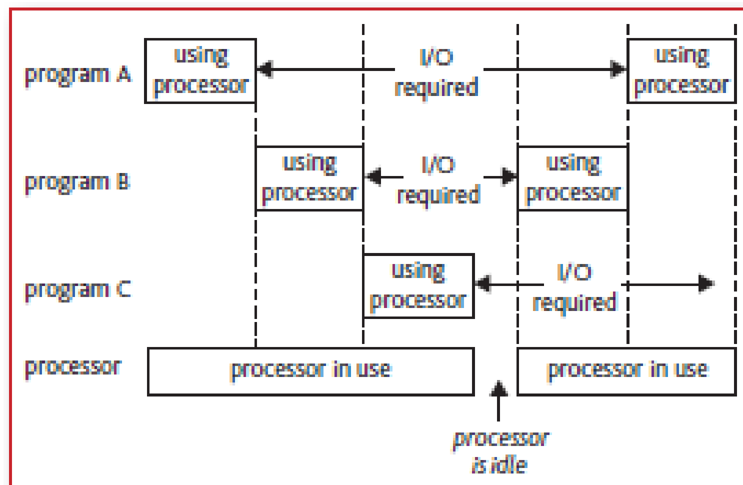
Multi-tasking

- Program A is waiting for the user's input
- Program B is waiting to replenish paper in the printer
- Then allocate processor time to program C



Purposes of operating systems

Multi-programming

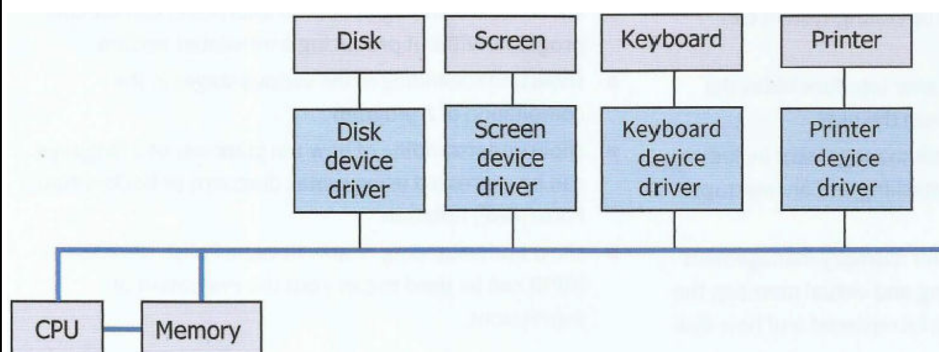


Purposes of operating systems

- The three fundamental resources in a computer system are:
  - the CPU
  - the memory
  - the I/O (input/output) system
- Resource management relating to the CPU concerns **scheduling** to ensure efficient usage
- Resource management relating to the memory concerns optimum usage of main memory
- The I/O system relates to:
  - Input and output that directly involves the user
  - Input and output to storage devices while a program is running

Purposes of operating systems

- The I/O system



Purposes of operating systems

- Bus structure is used to allow transfer of data between an I/O device and memory
- The OS can ensure that I/O passes via the CPU but for large quantities of data the operating system can ensure direct transfer between memory and an I/O device

Purposes of operating systems

OS facilities provided for the user

- User interface may be made available as a **command line**, a **graphical display** or a **voice recognition system**
- The function is always to allow the user to interact with running programs
- When a program involves use of a device, the operating system provides the device driver
- OS provides a file system for a user to store data and programs

Purposes of operating systems

OS facilities provided for the user

- User chooses filenames and folder organisation
- User does not have to organise the physical data storage on a disk
- For programmers, the OS supports the provision of a programming environment
- This allows a program to be created and run without the programmer being familiar with how the processor functions

Purposes of operating systems

OS facilities provided for the user

- OS provides a set of system calls that provide an interface to the services it offers
- E.g. When data needs to be read from file, the request for the file is converted into a system call that causes the operating system to take charge, find the file and make it available to the program
- An extension of this concept is when an operating system provides an application programming interface (API)
- Each API call fulfils a specific function such as creating a screen icon
- The API might use one or more system calls
- The API aims to provide portability for a program

Purposes of operating systems

OS structure

- An OS is structured to provide a **platform for** both **resource management** and the **provision of facilities for users**
- The logical structure of the OS provides two modes of operation
  - User mode is the one available for the user or an application program
  - The alternative has a number of different names of which the most often used are **'privileged mode'** or **'kernel mode'**
- The difference between the two is that kernel mode has sole access to part of the memory and to certain system functions that user mode cannot access

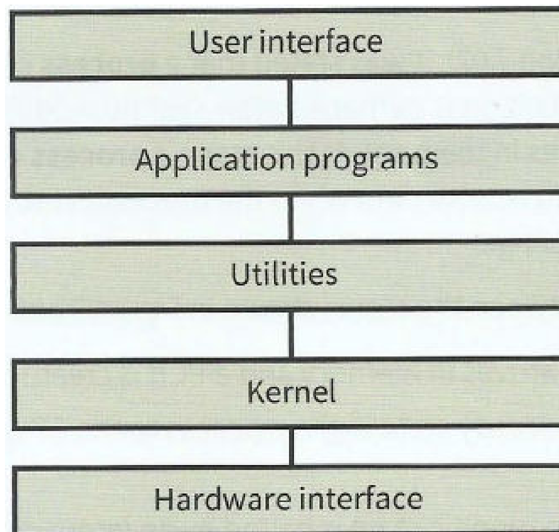
Purposes of operating systems

OS structure

- It is normal for the operating system to be separated into:
  - a kernel which runs all of the time and
  - the remainder which runs in user mode
- In this model, application programs or utility programs could make system calls to the Kernel
- However, to work properly each higher layer needs to be fully serviced by a lower layer (as in a network protocol stack)

Purposes of operating systems

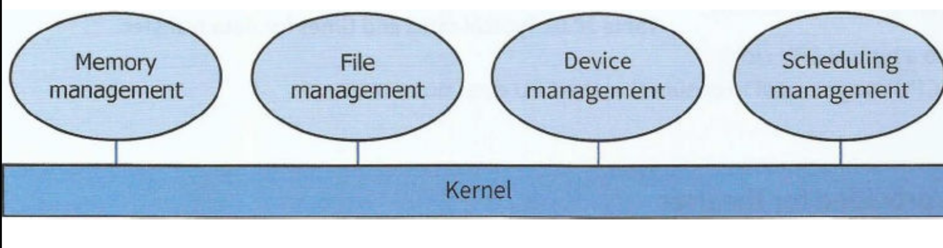
OS structure



Purposes of operating systems

OS structure

- A more flexible approach uses a modular structure
- The structure works by the kernel calling on the individual services when required
- It could possibly be associated with a micro-kernel structure where the functionality in the kernel is reduced to the absolute minimum



Process scheduling

- A **long-term or high-level scheduler** program controls the selection of a program stored on disk to be moved into main memory
- Occasionally a program has to be taken back to disk due to the memory getting overcrowded
- This is controlled by a **medium-term scheduler**
- When the program is loaded in memory, a **short-term or low-level scheduler** controls when it has access to the CPU

Process scheduling

- The OS must have a strategy for deciding which program is next given use of the processor
- **Low-level scheduling:** The process of deciding on the allocation of processor time/usage
- **High-level scheduling:** The process of deciding the order in which new programs are loaded into primary memory

Process scheduling

- OS must allocate processor time in an appropriate way to all tasks / requests (low-level scheduling)
- The scheduling algorithm should ensure that the **processor is working** to its **full potential**
- Simplest technique is to allocate equal time slots to each task
- Priorities then will be meaningless (???)

Process scheduling

I/O-bound job

- When a program makes little use of the processor
  - Because of few simple calculations,
  - It performs a lot of printing,
  - Requests user feedback continuously,
  - Continuously reading data from the disk drive

Process scheduling

Processor-bound job

- When a program makes a great deal of use of the processor
- Collects one set of data from the drive
- Carries complex calculations,
- Produces very few printouts at the end of processing

Process scheduling

Input/output-bound and processor-bound jobs

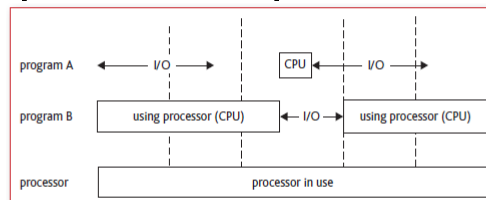


Figure 3.1.7 Scheduling two programs: B has priority so A spends time waiting.

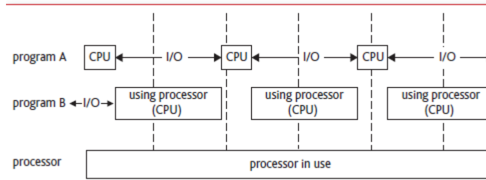


Figure 3.1.8 Scheduling two programs: A has priority so spends less time waiting.

Process scheduling

Input/output-bound and processor-bound jobs

- The OS should give:
  - high priority to I/O-bound programs
  - low priority to processor-bound programs.
- The following must have high priority:
  - safety-critical applications,
  - online applications,
  - interactive programs,
  - real-time applications
- Batch-processing programs should be given low priority

Process scheduling

Objectives of scheduling

- Maximise the use of the whole of the computer system
- Be fair to all programs
- Provide a reasonable response time:
  - to users in front of online applications programs at a terminal
  - to batch-processing programs (i.e. to meet the deadline for production of the final output)

Process scheduling

Objectives of scheduling

- Prevent the computer system failing if it is becoming overloaded
- Ensure that the system is consistent by giving similar response times to similar activities

Process scheduling

Objectives of scheduling

- Factors to consider when assessing if the objective can be met:
- Priority: low priority to processor-bound jobs
- Type of job:
  - real-time processing must be done quickly so that the next input can be affected
  - batch processing only has to meet the deadline for final outputs

Process scheduling

Objectives of scheduling

- Resource requirements: the amount of processor time needed to complete the job,
- The memory required, the amount of I/O time needed
- Waiting time: the time the job has been waiting to be loaded into main memory

Process scheduling

Scheduling algorithms

- A scheduling algorithm can be **preemptive** or **non-preemptive**
- A preemptive algorithm can halt a process that would otherwise continue running undisturbed
- If an algorithm is preemptive it may involve prioritising processes
- The simplest possible algorithm is first come first served (FCFS)
- This is a non-preemptive algorithm and can be implemented by placing the processes in a first-in first-out (FIFO) queue

Process scheduling

Scheduling algorithms

- A **round-robin algorithm** allocates a time slice to each process
- Therefore it is preemptive, because a process will be halted when its time slice has run out
- It can be implemented as a FIFO queue
- It normally does not involve prioritising processes
- However, if separate queues are created for processes of different priorities then each queue could be scheduled using a **round-robin** algorithm

Process scheduling

Scheduling algorithms

- A **priority-based scheduling** algorithm is more complicated
- One reason for this is that every time a new process enters the ready queue or when a running process is halted, the priorities for the processes may have to be re –evaluated
- The other reason is that whatever scheme is used to judge priority level it will require some computation

Process scheduling

Scheduling algorithms

- Possible criteria are:
- Estimated time of process execution
- Estimated remaining time for execution
- Length of time already spent in the ready queue
- Whether the process is I/O bound or CPU bound

Process scheduling

Scheduling algorithms

- More than one of these criteria might be considered
- Clearly, estimating a time for execution may not be easy
- Some processes require extensive I/O, for instance printing wage slips for employees
- There is very little CPU usage for such a process so it makes sense to allocate it a high priority so that the small amount of CPU usage can take place
- The process will then change to the waiting state while the printing takes place

Process scheduling

Scheduling strategies

- **Shortest job first**

- Jobs in the ready queue are sorted in ascending order of the total processing time the job is expected to need
- New jobs are added to the queue in such a way as to preserve this order

Process scheduling

Scheduling strategies

- **Round robin**

- Each job is given a maximum length of processor time (a time slice) after which the job is put at the back of the ready queue and the job at the front of the queue is given use of the processor

- If a job is completed before its time slice is used up, it leaves the system

Process scheduling

Scheduling strategies

- **Shortest remaining time**

- Jobs in the ready queue are sorted in ascending order of the remaining processing time the job is expected to need
- Good for short jobs
- There is a danger of long jobs being prevented from running because they never manage to get to the front of the queue

Process scheduling

Job status

- When entering the system, a job is placed in the ready queue by the high-level scheduler (HLS)

- Moving jobs in and out of the ready state is done by the low-level scheduler (LLS)

Process scheduling

- A process can be defined as **'a program being executed'**
- This definition is perhaps better slightly modified to include the state when the program first arrives in memory
- At this stage a **process control block (PCB)** can be created in memory ready to receive data when the process is executed
- Once in memory the state of the process can change

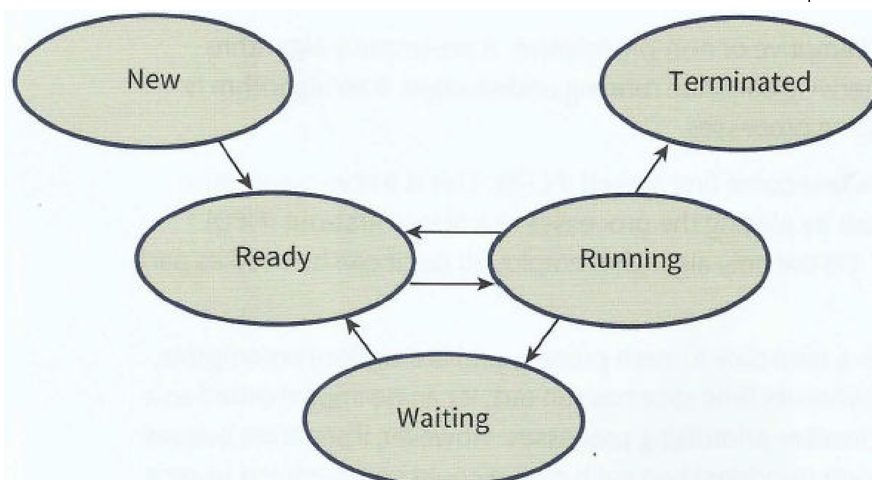
Process scheduling

- The transitions between the states can be described as follows:
- A new process arrives in memory and a PCB is created; **it changes to the ready state**
- A process in the ready state is given access to the CPU by the dispatcher; **it changes to the running state**
- A process in the running state is halted by an interrupt; **it returns to the ready state**

Process scheduling

- A process in the running state cannot progress until some event has occurred (I/O perhaps); **it changes to the waiting state** (sometimes called the 'suspended' or 'blocked' state)
  - A process in the waiting state is notified that an event is completed; **it returns to the ready state**
  - A process in the running state completes execution; **it changes to the terminated state**
- It is possible for a process to be separated into different parts for execution called **threads** where each thread is handled as though it were a process

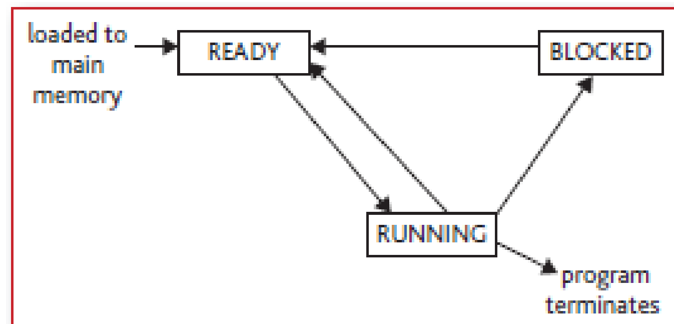
Process scheduling



Process scheduling

Job status

- ready - the job is waiting for use of the processor
- running - the job is currently using the processor
- blocked - the job is unable to use the processor at present



Process scheduling

Interrupts

- The fundamental process of the **Von Neumann computer architecture**
- Executes a sequence of program instructions stored in main memory
- The **fetch-execute cycle** (fetch-decode-execute)

Process scheduling

Interrupts

- Some interrupts are caused by errors that prematurely terminate a running process
- Otherwise there are two reasons for interrupts:
  - ***Processes consist of alternating periods of CPU usage and I/O usage***
  - ***I/O takes far too long for the CPU to remain idle waiting for it to complete***
  - The interrupt mechanism is used when a process in the running state makes a system call requiring an I/O operation and has to change to the waiting state

Process scheduling

Interrupts

  - ***The scheduler decides to halt the process for one of several reasons***
- Whatever the reason for an interrupt, the OS kernel must invoke an interrupt-handling routine
- This may have to decide on the priority of an interrupt
- One required action is that the current values stored in registers must be recorded in the process control block
- This allows the process to continue execution when it eventually returns to the running state

Process scheduling

Interrupts

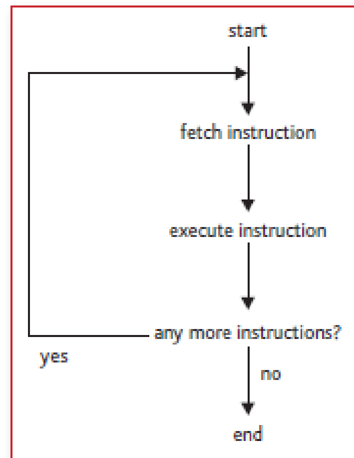


Figure 3.1.4 The fetch–execute cycle.

Process scheduling

Interrupts

- Satisfactory if processor understands (decodes correctly) the instructions
- Program maintains the processor control

- Sometimes this normal order of operation is interrupted in order for:
  - Processor time to be given to other programs loaded
  - Service an important routine

Process scheduling

Interrupts

- Interrupt Requests trigger these changes
- Interrupts are messages to the CPU
- They will be addressed by the CPU
- They will be serviced according to the severity of the request (e.g. program running is more “important” than the one requesting)

Process scheduling

Interrupts

- There are various types of Interrupt Requests
  - An I/O interrupt is generated by an I/O device to signal that a data transfer is complete or an error (e.g. printer out of paper) has occurred,
  - A timer interrupt is generated by an internal clock to indicate that the processor must attend to some time-critical activity,
  - A hardware interrupt is generated by any hardware problem, (e.g. a power failure which indicates that the OS must close down as safely as possible.

Process scheduling

Interrupts

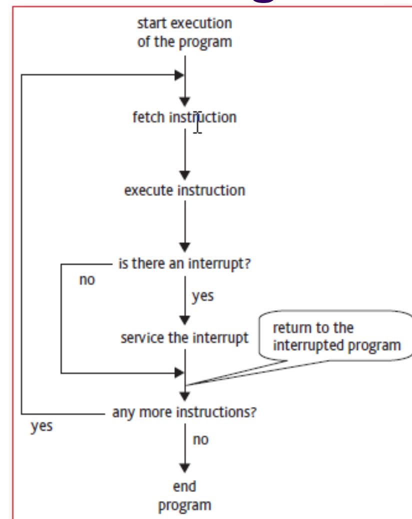


Figure 3.1.5 The fetch–execute cycle with an interrupt.

Process scheduling

Interrupts

- After the execution of an instruction
- Processor checks to see if an interrupt has occurred
- If so, the OS services the interrupt if it is more important than the task already being carried out
- This involves running a program called the **interrupt service routine** (ISR)
- Every interrupt signal, e.g. “printer out of paper”, has its own ISR that “services” the interrupt

Process scheduling

Interrupts

- Potential problem if too many interrupts requests ask to be serviced (current program may take too long to be completed)
- OS “remembers” the state of the program interrupted and the data involved
- Stores contents of registers (already filled with data from the interrupted program)
- Restores them back to continue processing

Process scheduling

Interrupts

- Another problem arises when an IRQ currently serviced is interrupted also
- Simplest solution is to create a queue of all IRQs
- **Queue** is called **Priority Queue**
- Order of programs in queue will be based on the priority level of each task

Process scheduling

Interrupts

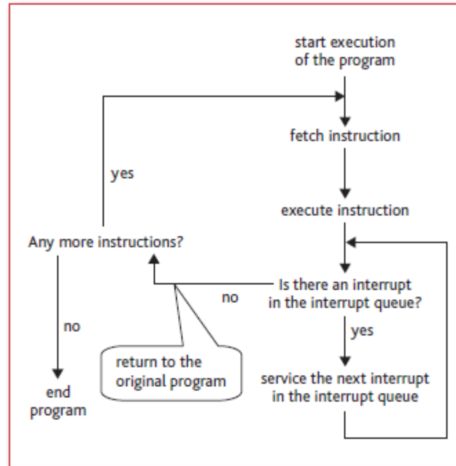


Figure 3.1.6 The fetch–execute cycle with multiple interrupts.

Process scheduling

Interrupts

Memory management

- The process of the operating system managing the use of the computer's primary memory
- The memory management module within the OS must:
  - keep track of memory used
  - keep track of memory which is available
  - make memory available when a program finishes execution
  - constantly check whether or not there is enough memory available to load a new program
  - ensure that two loaded programs do not attempt to use the same memory space

Memory management

Memory management includes:

- Provision of protected memory space for the OS kernel
- The loading of a program into memory requires defining the memory addresses for the program itself, for associated procedures and for the data required by the program

Memory management

Memory management includes:

- In a multiprogramming system, this might not be straight forward
- The storage of processes in main memory can get fragmented in the same way as happens for files stored on a hard disk
- There may be a need for the medium-term scheduler to move a process out of main memory to ease the problem

Memory management

- One memory management technique is to partition memory with the aim of loading the whole of a process into one partition
- Dynamic partitioning allows the partition size to match the process size
- An extension of this idea is to divide larger processes into segments, with each segment loaded into a dynamic partition

Memory management

- For a job to be able to use the processor, the job must be loaded into the computer's main memory
- If memory permits, several jobs can be loaded
- Programs and their data must be protected from the actions of other jobs being processed

Memory management

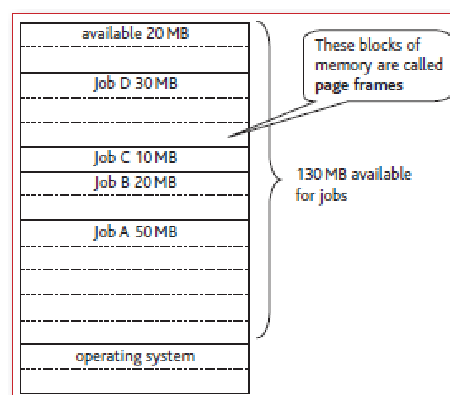


Figure 3.1.10 Jobs loaded into main memory.

Memory management

- When jobs finish and unloaded from memory, empty slots are created
- OS must fit programs/data in empty slots (**LOADER**)
- Easy solution to shift loaded program/data files to other memory locations to allow for space for new programs/data
  - Loads CPU with a great deal of processing time
- Second solution is to split new program/data and save in non-consecutive memory addresses

Memory management

- Relative addresses must be used
- Loader takes care of loading jobs in memory and adjusting addresses
- Addresses may be calculated in relation to the address calculated for the 1st instruction (Relative addressing)
- Program/data blocks in non-consecutive addresses must be linked
  - i.e. **LINKER**

Memory management

Paging

- Divide memory into equal-sized sections, called **pages**
- Pages are of equal size and each job is allocated a number of pages
- The allocated pages may be in order, or they may be scattered across the memory

Memory management

Paging

- Divisions are pre-determined
- Data and programs have to be loaded into the available memory to get the best fit possible
- The division of the available memory is into units called **page frames**
- A program is divided into equal-sized blocks called **pages**

Memory management

Segmentation

- **Divide memory space required into small sections called segments**
- Relies on the OS (the linker and loader) to store each part of the program in memory
- Too many memory “gaps” after repeated load/unload to/from memory operations

- Segmentation is far more complex to control than paging because of the different and unpredictable nature of the sizes of the segments

Memory management

Virtual memory

- Jobs are **loaded into memory when** they are **needed**, using a paging technique
- When a program is running, only those pages that contain required code need to be loaded

- If there is not enough space in memory to have all pages stored then the movement of pages into memory can take a disproportionate amount of time

Memory management

Virtual memory

- Led to the creation of small amounts of fast access storage between the drive and the memory
- OS now has part of the program in memory and part on the hard drive
- It needs to **predict which pages are most likely to be accessed next** and store them in virtual memory

Memory management

Virtual memory

- The use of **library (DLL)** files fits exactly with this strategy for managing the memory
- 1. Load the main program module
- 2. Load each DLL file(s) required for processing

Memory management

Advantages of Virtual Memory

- Very large programs can be run even though there is not an equally large amount of memory
- Only part of a program needs to be in memory at any one time
- E.g.: the index tables for a database could be permanently in memory but the full tables could be brought in only when required

Memory management

Disadvantages of Virtual Memory

- The increase of system overhead when running virtual memory
- The worst problem is **'disk thrashing'**, when part of a process on one page requires another page which is on disk
- When that page is loaded it almost immediately requires the original page again
- This can lead to almost perpetual loading and unloading of pages

Memory management

Spooling

- The process of **managing** that the **input and output** for different jobs do not become mixed up while they are waiting to be output

- Print jobs are sent to a spool queue on a fast access storage device, such as a disk
- They are queued on the disk, ready for the printer to deal with them in order

Memory management

Spooling

- What actually happens is that the jobs for the printer are **stored as files on the hard drive**
- The only thing that is stored in the spool queue is **a reference to where the print file is located**
- When the reference to that job gets to the top of the queue, the file is retrieved from storage and sent to the printer

Memory management

Spooling

- A spool queue is not necessarily a "first-in–first-out" queue
- Such a conventional queue would put new jobs (or references to them) at the end of the queue
- In a spool queue, the more important jobs can be inserted further up the queue
- The data structure then becomes a **priority queue**

Memory management

Spooling

- A spool queue of jobs it isn't really a queue of the jobs, only a queue of references to them
- It does not follow rules of traditional queues as it might allow "pushing in"
- Keeps output from different jobs separate
- Saves the user having to wait for the processor until the output is actually printed by a printer (a relatively slow device)
- Lets the processor get on with further processing while the jobs are queued up

Memory management

Memory management in Windows 7

- The Windows 7 Task Manager utility shows the large number of programs and processes which are running before the user loads any applications software
- The user then loads the Windows Media Player
- The Task Manager utility (bottom) reports the large increase in processor usage
- The Task Manager utility also shows:
  - the usage of the kernel part of the OS
  - the current usage of memory
  - that Windows uses paged memory allocation

Memory management

Spooling

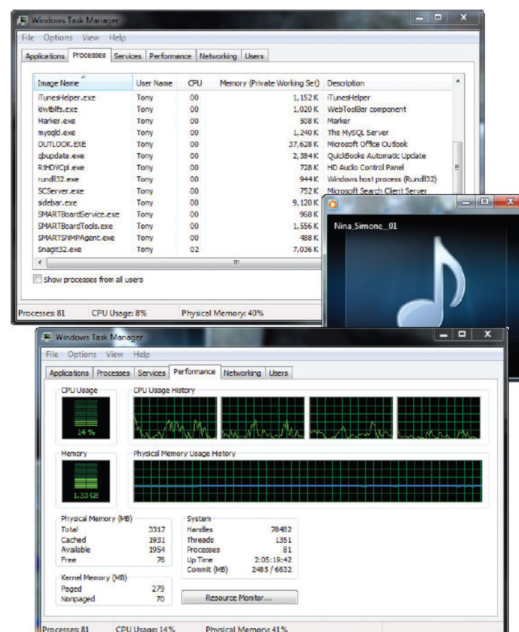
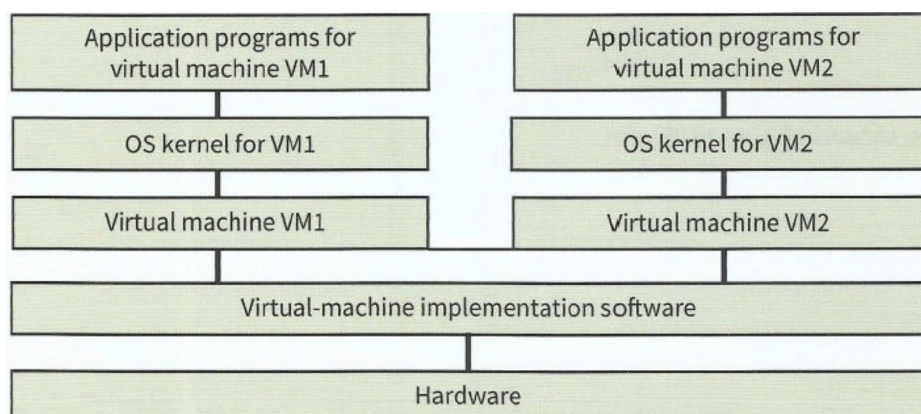


Figure 3.1.12 Processes running in Windows 7.

Virtual Machine

- The principle of a virtual machine is that a process interacts directly with a software interface provided by the operating system
- The kernel of the operating system handles all of the interactions with the actual hardware of the host system
- The software interface provided for the virtual machine provides an exact copy of the hardware

Virtual Machine



Virtual Machine

- The advantage of the virtual machine approach is that more than one different operating system can be made available on one computer system
- This is particularly valuable if an organisation has legacy systems and wishes to continue to use the software but does not wish to keep the aged hardware
- Alternatively, the same operating system can be made available many times
- Different companies can be offered their own virtual machine running as a server

Virtual Machine

- One drawback to using a virtual machine is the time and effort required for implementation
- Another is the fact that the implementation will not offer the same level of performance that would be obtained on a normal system

Stages of translation

- The translator must be able to "understand" what the program does
- Not the same understanding as by programmer
- "Understand" the structure of each line of code

- Each statement of a high-level language program undergoes through 3 (or 4?) stages of compilation

Stages of translation

- The front-end program performs analysis of the source code and produces an intermediate code that expresses completely the semantics (the meaning) of the source code
- The back-end program then takes this intermediate code as input and performs synthesis of object code
- Compiler does not stop at every option but accumulates them all
- Identifies all syntax errors and some logic errors
- Remaining logic errors require testing to be found
- Runtime errors are identified only via testing

Stages of translation

- Avg = (T1 + T2) / ~~3~~ 2
- (logic error NOT identified by translators)
- Avg = (T1 + T2) / 2

- IF Grade > 50 THEN
  - CONSOLE.WRITELINE(“~~FAIL~~ PASS”)

  - TESTING - Intensive and extensive testing
  - X=log(0) … ERROR (runtime or executable error)

Stages of translation

- The four stages of front-end analysis are:
  - lexical analysis
  - syntax analysis
  - semantic analysis
  - intermediate code generation

Lexical analysis

- Lexical analyser will:
- Convert the high-level language code into a stream of **tokens**
  - **E.g. Input → 1001000110100100**
- Replace single characters by their corresponding ASCII values
- **Create symbol table** by collecting/adding new identifiers (names /& data types) on the symbol table
- Remove comments/annotations and redundant (white) spaces, such as indentation

Lexical analysis

- Lexical analyser will:
- Report errors when found
  - E.g. INNPUT or INUT instead of INPUT
- Checks the rules for each identifier
  - E.g. Length of an identifier name used should be less than 64 bytes long
  - E.g. An identifier name cannot start using a number

Lexical analysis

- Lexical analyser will:
- Usually create a hash table to store two or more identifiers hashing to the same address/memory location
- Lexical analysis may take longer than the other stages of compilation

Syntax analysis

- Syntax analyser:
- Checks that the various rules of the language have been followed by every statement in the source code
- Gets the output code of the lexical analyser
- Parse the code to check that it is grammatically correct
  - i.e. ensures that it follows the rules defining each identifier and statement
  - These rules are defined by the **Backus-Naur Form (BNF) notation** or a **syntax diagram**

Syntax analysis

E.g.
<statement> := <variable><math_operator><variable>

Assume that the code
C:=A+B is received

Parser reads:
<variable><math_operator><variable>
But this is <statement>

Thus the syntax is correct ➔**Syntax analyser identifies a correct statement**, otherwise an error message is returned

Syntax analysis

Using:
<statement> := <variable><math_operator><variable>

Check the following lines of code

C:=(A+B) ☒ - brackets are not part of the definition

C:=A/B ☑

C:=A+B/A ☒ - only 1 operator and 1 variables are defined

More on parsing in Chapter 3.5

Syntax analysis

- Syntax analyser:
- Identifies any invalid identifiers or instructions not spotted during lexical analysis
  - E.g. invalid identifiers in a programming language that does not require declaration prior to their use
- Carries **semantic checks**
  - i.e. Checks the meaning of the statements; what they are meant to do including label checks, flow of control checks and declaration checks
  - E.g. check that specific subs/functions called, actually exist

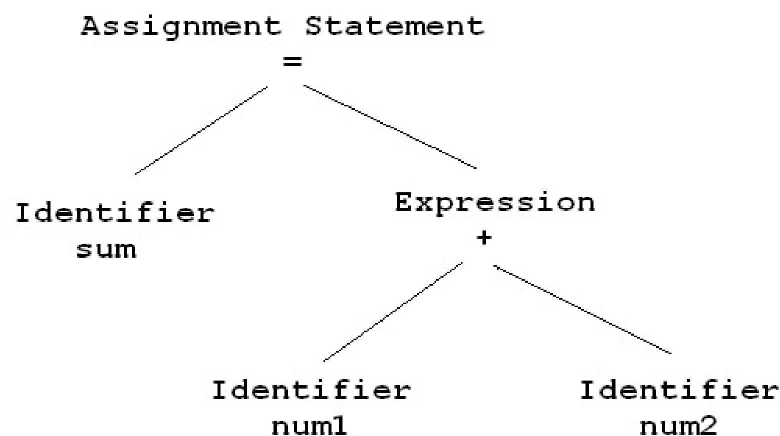
Syntax analysis

- Syntax analyser:
- Ensure that correct language constructs are used
  - E.g. Proper use of IF…ENDIF statements
  - E.g. Expected public/global identifiers declared locally
- Ensure that all identifiers have been declared and are used correctly, including data types
- This is the stage where data types, including their scope, are added to the symbol table
- Produce error messages pointing to any errors identified

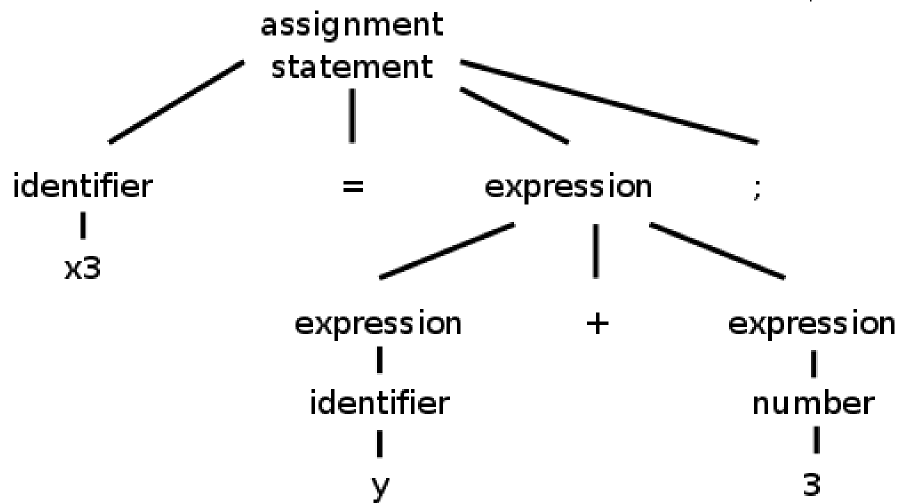
Semantic analysis

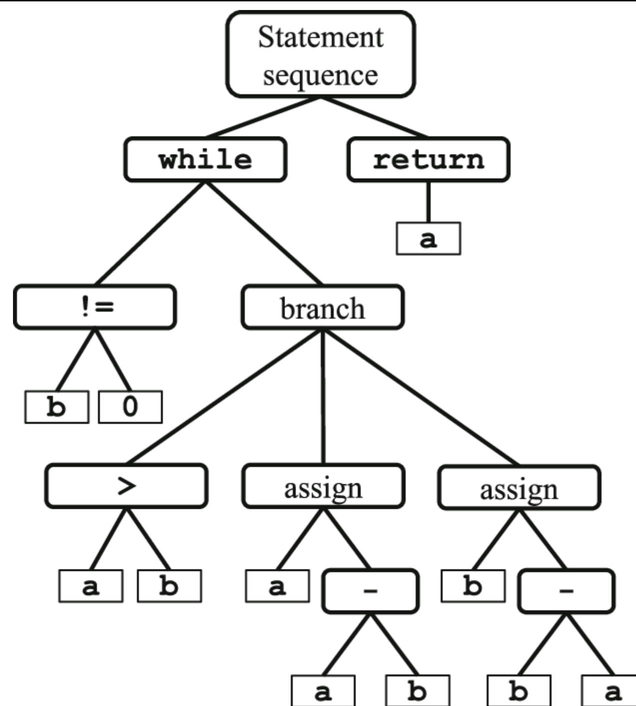
- Semantic analyser:
- Semantic analysis is about establishing the full meaning of the code
- An annotated abstract syntax tree is constructed to record this information
- For the identifiers in this tree an associated set of attributes is established including, for example, the data type
- These attributes are also recorded in the symbol table
- An often -used intermediate code created by the last stage of front-end analysis is a three address code
- As an example the following assignment statement has five identifiers requiring five addresses

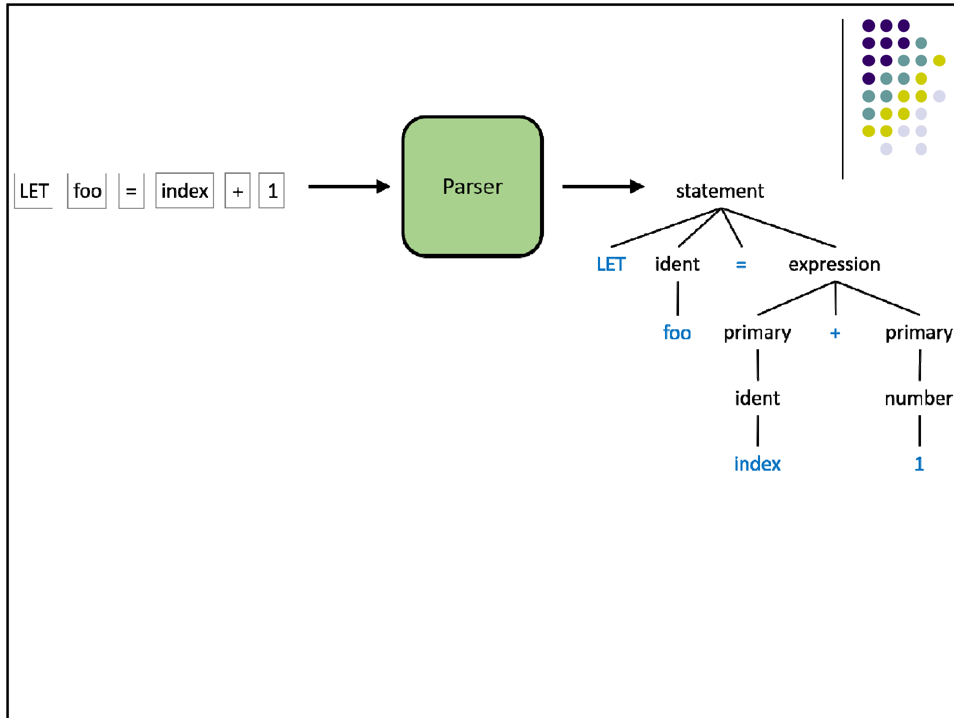
Semantic analysis

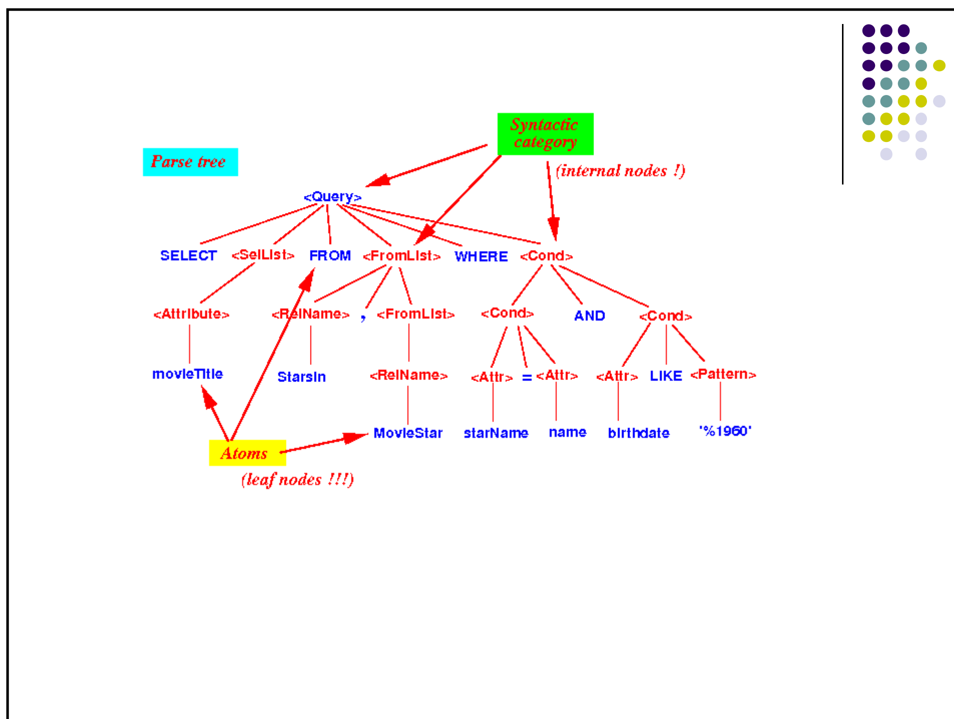


Semantic analysis






LET
foo
=
index
+
1
Parser
statement
LET
ident
=
expression
foo
primary
+
primary
ident
number
index
1

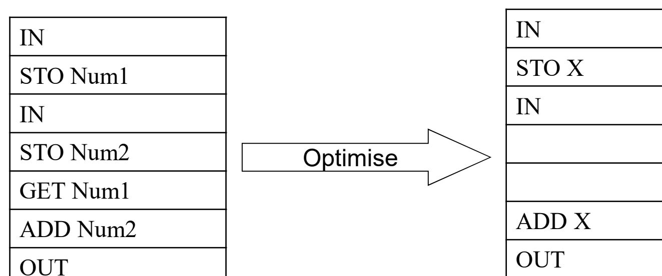

Syntactic category
Parse tree
(internal nodes !)
<Query>
SELECT
<SelList>
FROM
<FromList>
WHERE
<Cond>
<Attribute>
<RelName>
,
<FromList>
<Cond>
AND
<Cond>
movieTitle
StarsIn
<RelName>
<Attr>
=
<Attr>
<Attr>
LIKE
<Pattern>
MovieStar
starName
name
birthdate
'%1960'
Atoms
(leaf nodes !!!)

Code generation and optimisation

- Upon reaching this stage, it implies that all errors have been corrected
- Thus the source code must be converted to a form understood by the CPU → create object code required by the CPU that:
  - Will be processed as fast as possible
  - Be as short as possible

Code generation and optimisation

- Will be processed as fast as possible
- Be as short as possible



Linkers and Loaders

- Programs are usually built up from small, self-contained blocks of code called subprograms, procedures or **modules**, which can be reused whenever needed
- Each module can be compiled separately
- Each reusable object code can be stored in a “library” of routines
- Advantages of reusable object code:
- Already tested / works
- Saves time, not having to write and/or compile again

Linkers and Loaders

- Variable names and memory addresses are different from one function of the library routine to another
- Can be resolved by using two utility programs
- The **Loader** and the **Linker**
- A loader loads all the modules into memory and sorts out difficulties such as changes of addresses for the variables
- A linker links the modules together by making sure that references from one module to another are correct
- If one module calls another, it is important that the correct module is called and that the correct data are sent to the called module

Compilation errors

- During lexical analysis:
- Keywords in the program code are identified and turned into tokens of binary numbers ready for the next phase
- Lexical analyser expects to find the keyword in a **lookup table** that contains the appropriate replacement token
- If there is an error in a keyword, the analyser does not find a reference to it in the lookup table

Compilation errors

- If the analyser decides that the incorrect keyword is a variable, it inserts the name into the symbol table and does not report an error
- However, if the language requires that variable names should be declared, the name is not recognised and a diagnostic error is output

Compilation errors

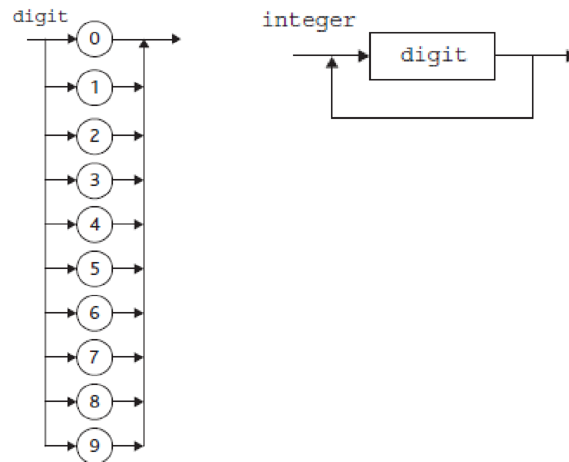
- At the syntax analysis stage, it is recognised that keywords are missing and errors are reported
- The task of the syntax analyser is to ensure that statements in the code follow the rules that are laid down for particular keywords
- Errors are reported if the remainder of the statement does not match the “pattern” laid down for the specific keyword

Compilation errors

- Compiler does not stop at every option but accumulates them all
- Identifies all syntax errors and some logic errors
- Remaining logic errors require testing to be found
- Runtime errors are identified only via testing

Backus-Naur form and syntax diagrams

- Using syntax diagrams:



Backus-Naur form and syntax diagrams

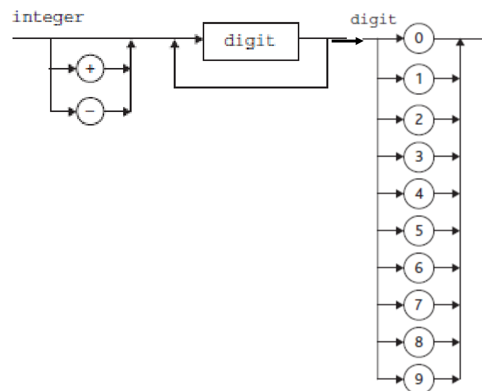
Using the derived definition of the integer, define a positive or negative integer

- <signed_integer> ::=<sign><integer>
- <integer> ::= <digit>|<digit><integer>
- <digit> ::= 0|1|2|3|4|5|6|7|8|9
- <sign> ::= +|-

5 Is an integer and acceptable signed integer
-34 is a signed integer
+2-4 is not a signed integer (-4 is not an integer)

Backus-Naur form and syntax diagrams

- Define a positive or negative integer using syntax diagrams:



Backus-Naur form and syntax diagrams

- Now try to define a variable for a programming language you are creating

- Must start with an upper case letter
- A-F are the only acceptable letters
- 1-5 are the only acceptable numbers
- Can contain any number of acceptable letters or numbers

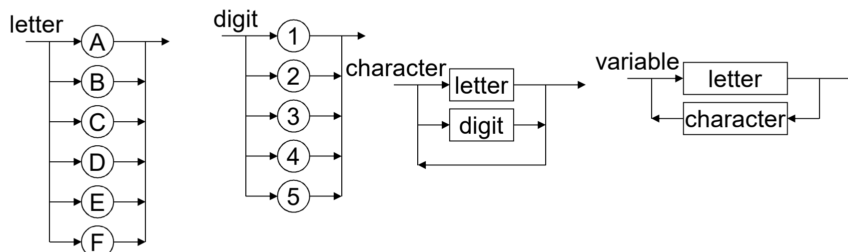
Backus-Naur form and syntax diagrams

- 1. <variable> ::= <letter>|<letter><character>
- 2. <character> ::= <letter>|<letter><character>|<digit>|<digit><character>
- 3. <letter> ::= A|B|C|D|E|F
- 4. <digit> ::= 1|2|3|4|5

- Test your definition
- A √ Rules 1,3
- A2 √ Rules 1,3,2,4
- 3A X Rules 1,3 Fails
- A1BC2 √ Rules 1,3,2,4,2,3,2,3,2,4
- ABS X Rules 1,3,2,3,2,3 Fails
- AB7 X Rules 1,3,2,3,2,4 Fails

Backus-Naur form and syntax diagrams

- Syntax diagrams for the variable defined for your programming language



Backus-Naur form and syntax diagrams

- Now try to define a variable
- Think of the rules defining a variable
  - Must start with a letter
  - Letters can be upper or lower case
  - Can contain any number of alphanumeric characters and certain symbols

- Finally, TEST the definition

Backus-Naur form and syntax diagrams

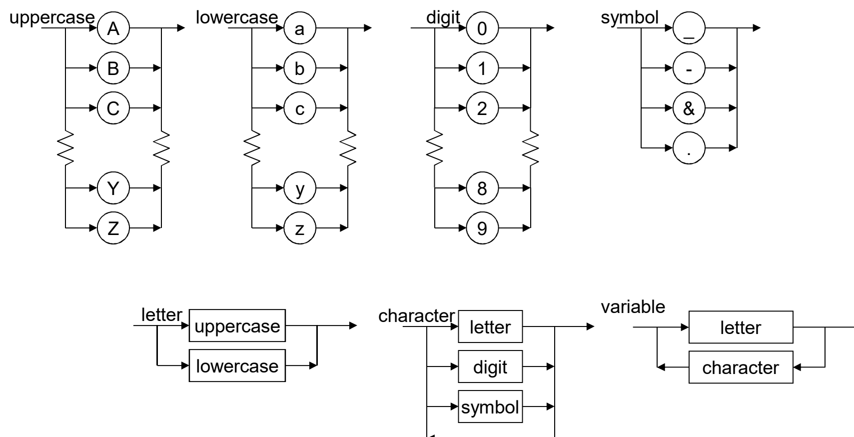
- <variable> ::= <letter>|<letter><character>
- <character> ::= <letter>|<letter><character>|
<digit>|<digit><character>|
<symbol>|<symbol><character>
- <letter> ::= <uppercase>|<lowercase>
- <uppercase> ::= A|B|C|D|E|F|G|H|I|J|K|L|M|
N|O|P|Q|R|S|T|U|V|W|X|Y|Z
- <lowercase> ::= a|b|c|d|e|f|g|h|i|j|k|l|m|
n|o|p|q|r|s|t|u|v|w|x|y|z
- <digit> ::= 0|1|2|3|4|5|6|7|8|9
- <symbol> ::= _|-|&|.

Backus-Naur form and syntax diagrams

- Test your definition
- A √
- A2 √
- 8A X
- A1B.C2 √
- &ABS X
- AB_7 √

Backus-Naur form and syntax diagrams

- Syntax diagram for defining the variable



Reverse Polish notation

- An expression is written (and read) as:
- A + B
- The arithmetic operation is in the middle of the two operands A, B
- This is called **INFIX** notation

- If the arithmetic operation is placed at the beginning:
- + A B ⇒ **PREFIX (also known as Polish notation)**

- If the arithmetic operation is placed at the end:
- A B + ⇒ **POSTFIX (also known as Reverse Polish notation)**

Reverse Polish notation

Reverse Polish is important in computing because:

- expressions written in postfix notation are unambiguous
- expressions do not need brackets
- expressions can be evaluated using a stack

5 – 3 + 2 = 4 --- Inorder
In RPN: 5 3 – 2 +

5 – (3 + 2) = 0 --- Inorder
In RPN: 5 3 2 + -

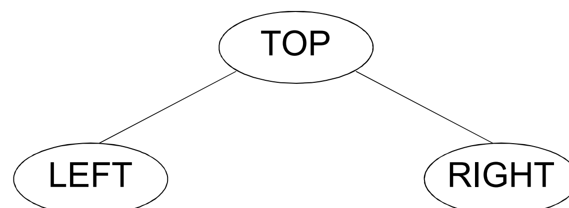
Reverse Polish notation

- E.g. Expression: A - B / C
- Infix: A - B / C
- Prefix: - A / B C
- Reverse Polish Notation (Postfix): A B C / -
- Note that the order of operands NEVER changes

- E.g. Expression: (A - B) / C
- Infix: (A - B) / C
- Prefix: - A B / C
- Reverse Polish Notation (Postfix): A B - C /

Converting between reverse Polish notation and the infix form of algebraic expressions

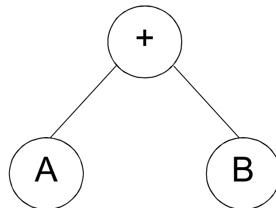
- **Traversing binary trees**



Always read from left to right. The **TOP** is read according to the traversal methodology:

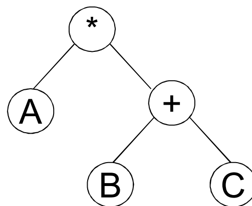
Polish notation		Reverse Polish notation
Prefix	Infix	Postfix
(Preorder)	(Inorder)	(Postorder)
Top Left Right	Left **Top** Right	Left Right **Top**

Converting between reverse Polish notation and the infix form of algebraic expressions



Polish notation		Reverse Polish notation
Prefix	Infix	Postfix
(Preorder)	(Inorder)	(Postorder)
Top Left Right	Left Top Right	Left Right Top
+ A B	**A + B**	**A B +**

Converting between reverse Polish notation and the infix form of algebraic expressions



Polish notation	Infix	Reverse Polish notation
Top Left Right	Left Top Right	Left Right Top
*** A + B C**	**A * (B + C)**	**A B C + ***

Converting between reverse Polish notation and the infix form of algebraic expressions

Express the following in Polish and Reverse Polish Notation:

a. (A + B) / (C - D) RPN: AB+CD-/

b. A + B / C - D

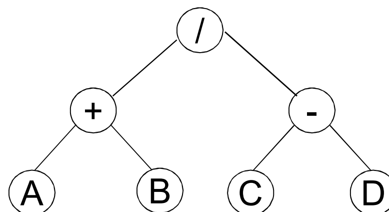
c. A * (B - (C + D)) RPN: ABCD+-*

d. A * B - C + D

e. A - B * (C + D)

Converting between reverse Polish notation and the infix form of algebraic expressions

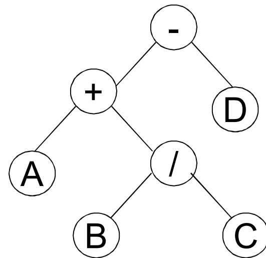
a. (A + B) / (C - D)
Using Infix notation construct the binary tree



Polish notation	Infix	Reverse Polish notation
/ + A B - C D	**(A + B) / (C - D)**	**A B + C D - /**

Converting between reverse Polish notation and the infix form of algebraic expressions

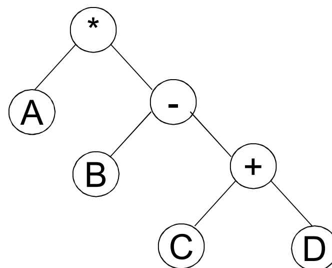
b. A + B / C - D
Binary tree (Using Infix notation)



Polish notation	Infix	Reverse Polish notation
- + A / B C D	**A + B / C - D**	**A B C / + D -**

Converting between reverse Polish notation and the infix form of algebraic expressions

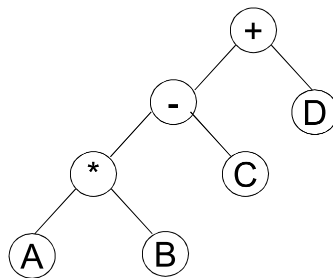
c. A * (B - (C + D))
Binary tree



Polish notation	Infix	Reverse Polish notation
*** A - B + C D**	**A * (B - (C + D))**	**A B C D + - ***

Converting between reverse Polish notation and the infix form of algebraic expressions

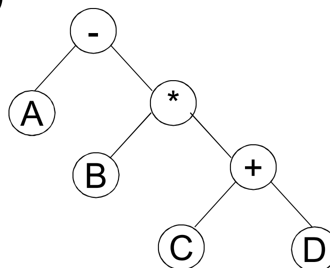
d. A * B - C + D
Binary tree



Polish notation	Infix	Reverse Polish notation
+ - * A B C D	**A * B - C + D**	**A B * C - D +**

Converting between reverse Polish notation and the infix form of algebraic expressions

e. A - B * (C + D)
Binary tree



Polish notation	Infix	Reverse Polish notation
- A * B + C D	**A - B * (C + D)**	**A B C D + * -**

Converting between reverse Polish notation and the infix form of algebraic expressions

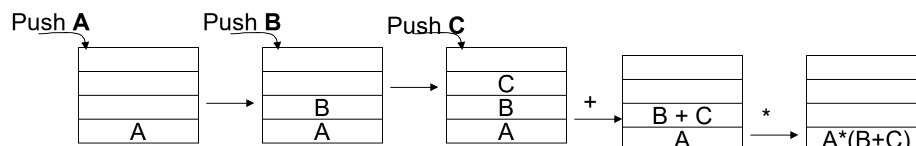
Using a stack

- To convert a reverse Polish expression to infix notation or to evaluate the expression, a stack can be used

1. Each operand is placed (pushed) onto the stack in turn until it meets an operator
2. Read the operator
3. The two top values in the stack are read (popped)
4. The operator is carried out on them
5. The result is placed on the stack
6. Repeat until the final infix expression is the only item on the stack or the expression has been evaluated

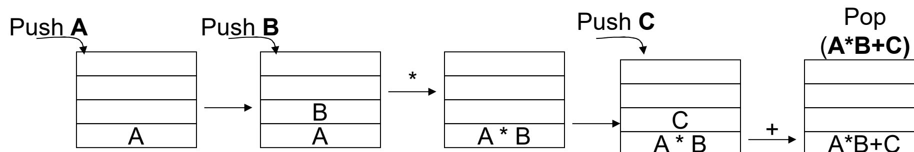
Converting between reverse Polish notation and the infix form of algebraic expressions

- E.g. A * (B + C)
- Reverse Polish notation: A B C + *



Converting between reverse Polish notation and the infix form of algebraic expressions

- E.g. A * B + C
- Reverse Polish notation: A B * C +



Converting between reverse Polish notation and the infix form of algebraic expressions

- E.g. 5 * 8 + 3
- Reverse Polish notation: 5 8 * 3 +