# Overview of Language Processors

**Language Processors** A Language Processor is a software which bridges a specification or execution gap. Program to input to a LP is referred as a Source Program and output as Target Program. A language translator bridges an execution gap to the machine language of a computer system.

# Language Processing System

Any computer system is made of hardware and software. The hardware understands a language, which humans cannot understand. So we write programs in high-level language, which is easier for us to understand and remember. These programs are then fed into a series of tools and OS components to get the desired code that can be used by the machine. This is known as Language Processing System.

The high-level language is converted into binary language in various phases. A compiler is a program that converts high-level language to assembly language. Similarly, an assembler is a

program that converts the assembly language to machine-level language.

User writes a program in C language (high-level language).

The C compiler, compiles the program and translates it to assembly program (low-level language).

An assembler then translates the assembly program into machine code (object).

A linker tool is used to link all the parts of the program together for execution (executable machine code).

A loader loads all of them into memory and then the program is executed.

# Preprocessor

A preprocessor, generally considered as a part of compiler, is a tool that produces input for compilers. It deals with macro-processing, file inclusion, language extension, etc.

# Interpreter

An interpreter, like a compiler, translates high-level language into low-level machine language. The difference lies in the way they read the source code or input. A compiler reads the whole source code at once, creates tokens, checks semantics, generates intermediate code, executes the whole program and may involve many passes. In contrast, an interpreter reads a statement from the input, converts it to an intermediate code, executes it, then takes the next statement in sequence. If an error occurs, an interpreter stops execution and reports it. whereas a compiler reads the whole program even if it encounters several errors.

# Assembler

An assembler translates assembly language programs into machine code. The output of an assembler is called an object file, which contains a combination of machine instructions as well as the data required to place these instructions in memory.

# Linker

Linker is a computer program that links and merges various object files together in order to make an executable file. All these files might have been compiled by separate assemblers. The major task of a linker is to search and locate referenced module/routines in a program and to determine the memory location where these codes will be loaded, making the program instruction to have absolute references.

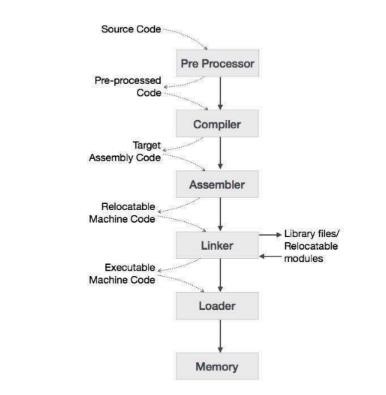
# Loader

Loader is a part of operating system and is responsible for loading executable files into memory and execute them. It calculates the size of a program (instructions and data) and creates memory space for it. It initializes various registers to initiate execution.

# Cross-compiler

A compiler that runs on platform (A) and is capable of generating executable code for platform (B) is called a cross-compiler.

Source-to-source Compiler A compiler that takes the source code of one programming language and translates it into the source code of another programming language is called a source-to-source compiler

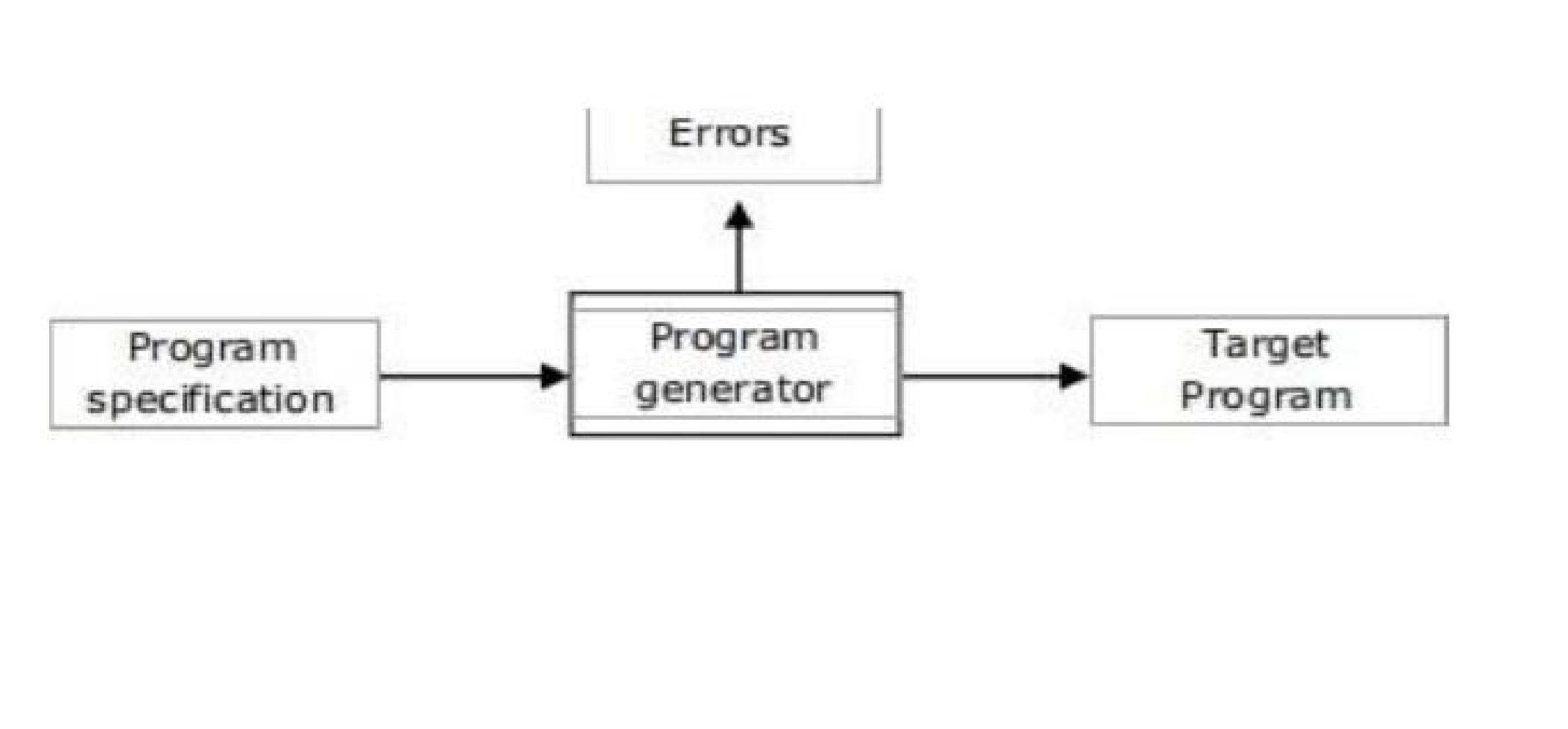


# Language processing activity

There are mainly two types of language processing activity which bridges the semantic gap between source language and target language.

# 1. Program generation activities

A program generation activity aims an automatic generation of a program.

Program generator is software, which aspects source program and generates a program in target language. Program generator introduces a new domain between the application and programming language domain is called program generator domain

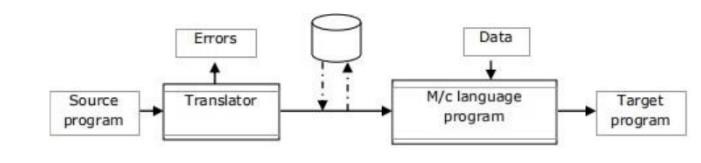
**2. Program Execution**

Two popular models for program execution are translation and interpretation.

# A. Translation

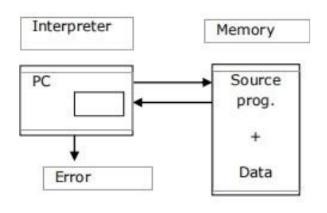
The program translation model bridges the execution gap by translating a program written in PL

called source program, into an equivalent program in machine or assembly language of the computer system, called target program.



# B. Interpretation

The interpreter reads the source program and stores it in its memory. The CPU uses the program counter (PC) to note the address of the next instruction executed.



The statement would be subjected to the interpretation cycle, which could consist following steps:

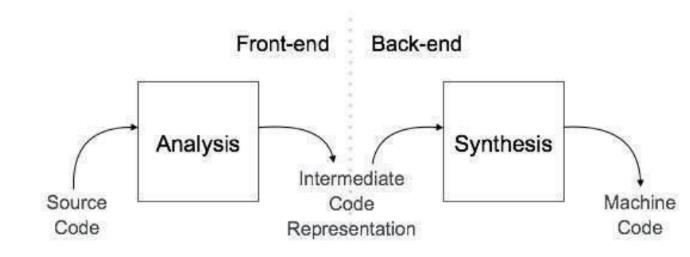
1. Fetch the instruction
2. Analyze the statement and determine its meaning, the computation to be performed and its operand.
3. Execute the meaning of the statement.

**Fundamental of Language Processing**  **Phases & Passes of Compiler**

A compiler can be broadly divided into two phases based on the way they compile.

# Analysis Phase

Known as the front-end of the compiler, the analysis phase of the compiler reads the source program, divides it into core parts and then checks for lexical, grammar and syntax errors. The analysis phase generates an intermediate representation of the source program and symbol table, which should be fed to the Synthesis phase as input.



# Synthesis Phase

Known as the back-end of the compiler, the synthesis phase generates the target program with the help of intermediate source code representation and symbol table.

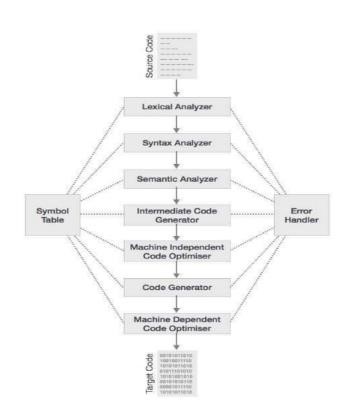
A compiler can have many phases and passes.

**Pass** : A pass refers to the traversal of a compiler through the entire program.

**Phase** : A phase of a compiler is a distinguishable stage, which takes input from the previous stage, processes and yields output that can be used as input for the next stage.

A pass can have more than one phase.

The compilation process is a sequence of various phases. Each phase takes input from its previous stage, has its own representation of source program, and feeds its output to the next phase of the compiler. Let us understand the phases of a compiler.



# Analysis phase Lexical Analysis

The first phase of scanner works as a text scanner. This phase scans the source code as a stream of characters and converts it into meaningful lexemes. Lexical analyzer represents these lexemes in the form of tokens as:

<token-name, attribute-value>

Lexical analysis builds a descriptor, called a token. We represent token as entry Code #no

where Code can be Id or Op for identifier or operator respectively and no indicates the entry for the identifier or operator in symbol or operator table. Consider following code i: integer; a, b: real; a= b + i;

The statement a= b+i is represented as a string of token a = b + i

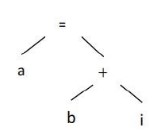
Id#1 Op#1 Id#2 Op#2 Id#3

# Syntax Analysis

The next phase is called the syntax analysis or parsing. It takes the token produced by lexical analysis as input and generates a parse tree (or syntax tree). In this phase, token arrangements are checked against the source code grammar, i.e. the parser checks if the expression made by the tokens is syntactically correct.

Syntax analysis processes the string of token to determine its grammatical structure builds an intermediate code that represents the structure.

The tree structure is used to represent the intermediate code. Consider the statement a = b + i can be represented in tree form as



# Semantic Analysis

Semantic analysis checks whether the parse tree constructed follows the rules of language.

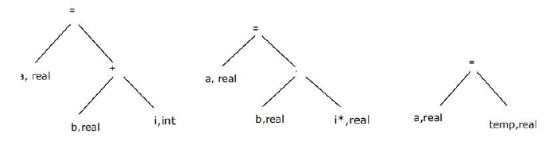
For example, assignment of values is between compatible data types, and adding string to an integer. Also, the semantic analyzer keeps track of identifiers, their types and expressions; whether identifiers are declared before use or not etc. The semantic analyzer produces an annotated syntax tree as an output. Semantic analysis determines the meaning of a statement by applying the semantic rules to the structure of the statement.

While processing a declaration statement, it adds information concerning the type, length and dimensionality of a symbol to the symbol table.

While processing an imperative statement, it determines the sequence of actions that would have to be performed for implementing the meaning of the statement and represents them in the intermediate code.

Considering the tree structure for the statement a = b + i

If node is operand, then type of operand is added in the description field of operand. While evaluating the expression the type of b is real and i is int so type of i is converted to real i\*.



# Intermediate Code Generation

After semantic analysis the compiler generates an intermediate code of the source code for the target machine. It represents a program for some abstract machine. It is in between the high-level language and the machine language. This intermediate code should be generated in such a way that it makes it easier to be translated into the target machine code.

**IR contains intermediate code and table.**

Symbol table

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Symbol | Type | Length | Address |
| 1 | I | Int |  |  |
| 2 | A | Real |  |  |
| 3 | B | Real |  |  |
| 4 | I\* | Real |  |  |
| 5 | Temp | Real |  |  |

# Intermediate code

1.Convert(id1#1) to real, giving (id#4)

2.Add(id#4) to (id#3), giving (id#5)

3.Store (id#5) in (id#2)

**Synthesis phase**

# Memory allocation

The memory requirement of an identifier is computed from its type, length and dimensionality and memory is allocated to it.

The address of the memory area is entered in the symbol table

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Symbol | Type | Length | Address |
| 1 | I | Int |  | 2000 |
| 2 | A | Real |  | 2001 |
| 3 | B | Real |  | 2002 |

# Code Optimization

The next phase does code optimization of the intermediate code. Optimization can be assumed as something that removes unnecessary code lines, and arranges the sequence of statements in order to speed up the program execution without wasting resources (CPU, memory).

# Code Generation

In this phase, the code generator takes the optimized representation of the intermediate code and maps it to the target machine language. The code generator

translates the intermediate code into a sequence of (generally) re-locatable machine code. Sequence of instructions of machine code performs the task as the intermediate code would do.

The synthesis phase may decide to hold the value of i\* and temp in machine registers may generate the assembly code

CONV\_R AREG, I  CONV\_R TEMP, I

ADD\_R REG, B ADD\_R B, TEMP

MOV\_R A, B

A source code can directly be translated into its target machine code, then why at all we need to translate the source code into an intermediate code which is then translated to its target code? Let us see the reasons why we need an intermediate code.

If a compiler translates the source language to its target machine language without having the option for generating intermediate code, then for each new machine, a full native compiler is required.

Intermediate code eliminates the need of a new full compiler for every unique machine by keeping the analysis portion same for all the compilers.

The second part of compiler, synthesis, is changed according to the target machine.

It becomes easier to apply the source code modifications to improve code performance by applying code optimization techniques on the intermediate code.

# Intermediate Representation

Intermediate codes can be represented in a variety of ways and they have their own benefits.

# High Level IR

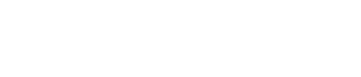
- High-level intermediate code representation is very close to the

source language itself. They can be easily generated from the source code and we can easily apply code modifications to enhance performance. But for target machine optimization, it is less preferred.

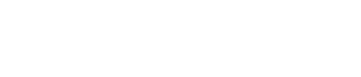
# Low Level IR

- This one is close to the target machine, which makes it suitable for register and memory allocation, instruction set selection, etc. It is good for machine-independent optimization.

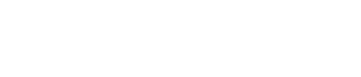
Toy Compiler



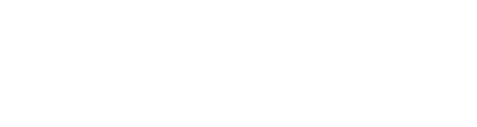
Semantic



Syntax

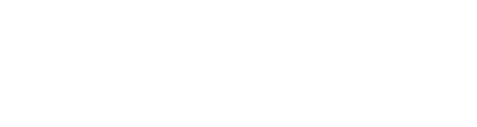


Lexical



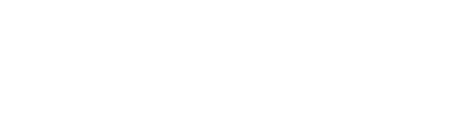
Back

End



Front

End



Toy

Compiler

**Front End**

1. It includes 3 analysis:
   1. Lexical
   2. Syntax
   3. Semantic
2. Functions of Front End:
   1. Determine validity of source statement from view point of analysis.
   2. Determine ‘content’ of source statement.
   3. Construct a suitable representation of source statement for the use of subsequent analysis function or synthesis phase of a language processor.
3. Result is stored in the following forms:
   1. Table of Information
   2. IC: Description of source statement.
4. Output of Front End:
   1. IR : Intermediate Representation. It has two things with it. 1. Table of Information. 2. Intermediate Code (IC).

Source

Statement

Lexical

class

of

lexical

units

information

Syntax

analysis,

structure

constructs

IC

Semantic

Action

Construct a

representation

of

statement

IR

a. Table of Content

* Contains information obtained during different analysis of source program.
* Eg. Symbol Table: Has information containing all the identifiers used in source program.
* Semantic analysis adds more information to symbol table.
* So, Symbol table have information about lexical analysis + new names designated by temporary results.
* Eg: IR produced by analysis phase of program: i : integer; a, b : real; a := b + i;

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Sr. No.** | **Symbol** | **Type** | **Length** | **Address** |
| 1 | I | Integer | - | - |
| 2 | A | Real | - | - |
| 3 | B | Real | - | - |
| 4 | i\* | Real | - | - |
| 5 | Temp | Real | - | - |

b. IC: Intermediate Code

* Has sequence of IC units.
* What does IC do?
* Each IC unit represent the meaning of each individual actions performed based on source program.
* IC unit may contain reference to information in various tables.
* Actions:
  + - Convert (id,#1) to real, giving (id,#4)
    - Add (id,#4) to (id,#3), giving (id,#5)
    - Store (id,#5) in (id,#2)

Lexical Analysis of Front End

* Also called Scanning.
* Identifies lexical units and then classifies these units further into lexical classes like id’s, constants, identifiers, reserved keywords and then enter them into different tables.
* Output: Token, Is the description of analysis. It has two parts. 1. Class Code, 2. Number.
* Eg: OP Table
* Class code identifies class to which a lexical unit belongs.
* Number in class is the entry number in lexical unit in relevant table.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | String of token:  a := b + i;  Id#2 op#5 id#3 op#3 id#1 op#1 | | |  |  | | --- | --- | | **Number** | **Class Code: OP** | | 1 | ; | | 2 | - | | 3 | + | | 4 | \* | | 5 | := | |

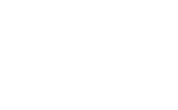
Syntax Analysis of Front End

* Also called Parsing.
* Syntax analysis process is the string of tokens built by lexical analyzer to determine the statement class.
* i.e Assignment Statement Class,

For Loop Class,

Switch Case, etc

* It builds IC representing the structure of statements in tree form.
* Why tree is chosen to represent IC?
* Answer: Tree can represent hierarchical structure of PL statement appropriately.
* IC is then passed to semantic analysis to determine meaning of the statement.
* Eg: IC



a,b

:

real

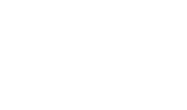
Real



a



b



a

:=

b

+

I;

:=



a



+

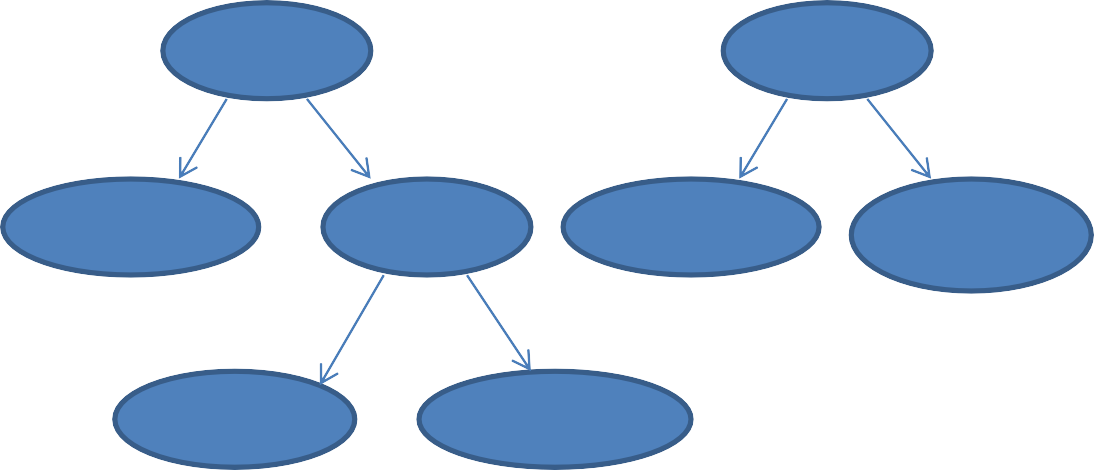


b

i

Semantic Analysis of Front End

* When semantic analysis determine the meaning of sub-tree in IC, it adds
  + - * Declaration (added to symbol table)
      * Imperative Statement Analysis (determines the action sequence.



:=



:=



a,

real

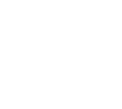


+



a,

real



temp,

real



b,

real



i\*,

real



:=



a,

real



+



b,

real



i,

int

Steps:

1.Information concerning the type of operands is added to IC tree. See f first tree.

2.Rules of meaning governing assignment statement on right hand side should be evaluated first. Hence focus on right sub-tree.

3.Rules of addition indicate type conversion of i.

i.e convert i to real, giving i\*. See 2nd sub tree.

4.Rules of addition indicate feasibility of addition which leads to “add i\* to b, giving temp”.

5.The assignment is performed leads to action “Store temp in a”.

Source

Program

Lexical

Errors

Syntax

Errors

Semantic

Errors



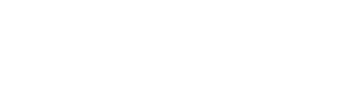
Tokens



Trees

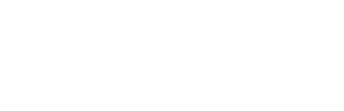


IC

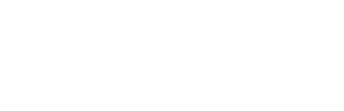


Semantic

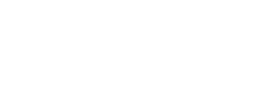
Analysis



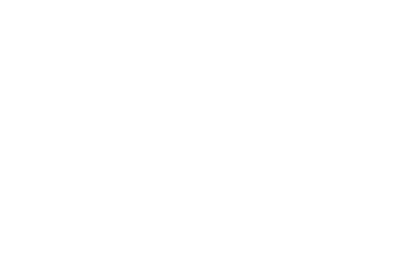
Parsing



Scanning



IR



Symbol Table

Constants

Table

Other

Tables

* Back End
  + - 1. Memory Allocation
      2. Code Generation

1. Memory Allocation:

– A memory allocation requirement of an identifier is computed from its

* Size,
* Length,
* Type,
* Dimension
  + - * And then memory is allocated to it.
      * The address of memory area is then entered in symbol table. See next figure.
      * i\* & temp are not shown because memory allocation is not required now for this id.
      * i\* and temp should be allocated memory but this decision are taken in preparatory steps of code generation.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Sr. No.** | **Symbol** | **Type** | **Length** | **Address** |
| 1 | I | Integer | - | 2000 |
| 2 | A | Real | - | 2016 |
| 3 | B | Real | - | 2020 |

2. Code Generation:

Uses knowledge of target architecture.

* 1. Knowledge of instruction
* 2. Addressing mode in target computer.

Important issues effecting code generation:

* Determine the place where the intermediate results should be kept. i.e in memory or register?
* Determine which instruction should be used for type conversion operation.
* Determine which addressing mode should be used for accessing variable.

Eg: for sequence of actions for assignment statement a:=b+c;

1. Convert I to real, giving i\*; 2. Add i\* to b, giving temp;

3. Store temp in a. The synthesis phase:

1. decide to hold the values of i\* & temp in machine register.
2. generate following assembly code.

CONV\_R AREG, I

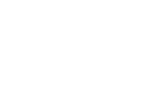
ADD\_R AREG,B

MOVE\_M AREG,A

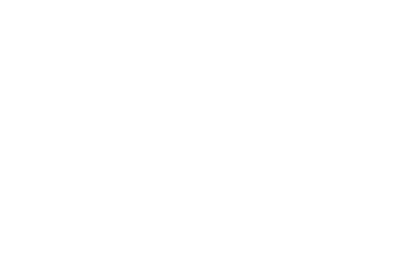
See next figure of Back End.

Target

Program



IR



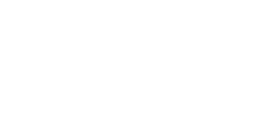
Symbol Table

Constants

Table

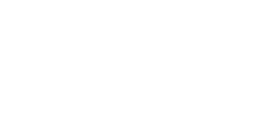
Other

Tables



Code

Generation



Memory

Allocation

# Symbol table

Symbol table is an important data structure created and maintained by compilers in

order to store information about the occurrence of various entities such as variable names, function names, objects, classes, interfaces, etc. Symbol table is used by both the analysis and the synthesis parts of a compiler.

A language processor uses the symbol table to maintain the information about attributes of symbols used in a source program.

It performs the following four kinds of operations on the symbol table:

1.Add a symbol and its attributes: Make a new entry in the symbol table.

2.Locate a symbol’s entry: Find a symbol’s entry in the symbol table.

3.Delete a symbol’s entry: Remove the symbol’s information from the table.

4.Access a symbol’s entry: Access the entry and set, modify or copy its attribute information.

The symbol table consists of a set of entries organized in memory.

**A symbol table may serve the following purposes depending upon the language in hand**:

1.To store the names of all entities in a structured form at one place.

2.To verify if a variable has been declared.

3.To implement type checking, by verifying assignments and expressions in the source code are semantically correct.

4.To determine the scope of a name (scope resolution).

**A symbol table is simply a table which can be either linear or a hash table. It maintains an entry for each name in the following format:**

<symbol name, type, attribute>

For example, if a symbol table has to store information about the following variable declaration:

static int interest; then it should store the entry such as:

<interest, int, static>

The attribute clause contains the entries related to the name.

# Implementation

If a compiler is to handle a small amount of data, then the symbol table can be

implemented as an unordered list, which is easy to code, but it is only suitable for

small tables only. A symbol table can be implemented in one of the following ways:

* Linear (sorted or unsorted) list
* Binary Search Tree
* Hash table

Among all, symbol tables are mostly implemented as hash tables, where the source

code symbol itself is treated as a key for the hash function and the return value is the information about the symbol.

# Operations

A symbol table, either linear or hash, should provide the following operations.

# 1. insert()

This operation is more frequently used by analysis phase, i.e., the first half of the

compiler where tokens are identified and names are stored in the table. This operation

is used to add information in the symbol table about unique names occurring in the

source code. The format or structure in which the names are stored depends upon the

compiler in hand.

An attribute for a symbol in the source code is the information associated with that

symbol. This information contains the value, state, scope, and type about the symbol.

The insert() function takes the symbol and its attributes as arguments and stores the information in the symbol table.

For example:

int a; should be processed by the compiler as: insert(a, int);

# 2. lookup()

lookup() operation is used to search a name in the symbol table to determine: if the symbol exists in the table.

if it is declared before it is being used. if the name is used in the scope. if the symbol is initialized. if the symbol declared multiple times.

The format of lookup() function varies according to the programming language. The basic format should match the following: lookup(symbol)

This method returns 0 (zero) if the symbol does not exist in the symbol table.

If the symbol exists in the symbol table, it returns its attributes stored in the table.

# Data Structures for Language Processing

Two kinds of data structures can be used for organizing its entries:

O Linear data structure: Entries in the symbol table occupy adjoining areas of memory.

This property is used to facilitate search.

O Nonlinear data structure: Entries in the symbol table do not occupy contiguous areas of memory. The entries are searched and accessed using pointers.

# Symbol table entry formats

Each entry in the symbol table is comprised of fields that accommodate the attributes of one symbol. The symbol field of fields stores the symbol to which entry pertains.

The symbol field is key field which forms the basis for a search in the table.

The following entry formats can be used for accommodating the attributes:

O **Fixed length entries**: Each entry in the symbol table has fields for all attributes specified in the programming language.

O **Variable-length entries**: The entry occupied by a symbol has fields only for the attributes specified for symbols of its class.

O **Hybrid entries:** A hybrid entry has fixed-length part and a variable-length part.

# Search Data Structures

Search data structures (Search structure) is used to create and organize various tables of information and mainly used during the analysis of the program.

# Features of Search data structures

The important features of search data structures include the following:

An entry in search data structure is essentially a set of fields referred to as a record.

Every entry in search structure contains two parts: fixed and variable. The value in fixed part determines the information to be stored in the variable part of the entry

# Operations on Search Structures

Search structures are characterized by following operations:

**Insert Operation:**

To add the entry of a newly found symbol during language processing.

**Search Operation:**

To enable and support search and locate activity for the entry of symbol

**Delete Operation:** To delete the entry of a symbol especially when identified by processor as redundant declarations.

# Sequential Search Organization

In sequential search organization, during the search for a symbol, probability of all active entries being accessed in the table is same.

For an unsuccessful search, the symbol can be entered using an ‘add’ operation into table.

# Binary Search Organization

Tables using binary search organization have their entries assumed to satisfy an ordering relation. It should be considered that for table containing ‘f’ occupied entries, the probability of successful search is log2f and unsuccessful search is log2f. The binary search organization requires that entry number of a symbol table should not change after ‘add’ operation. This may become limiting factor for addition and deletion during language processing.

# Hash Table Organization

* A hash table, also known as a hash map is a data structure that has the ability to map keys to the values using a hash function.
* Hash table organization is an efficient implementing associative arrays and symbol tables that outperform other data structures with its capability of performing 'm' accesses on 'n' names.
* It has the following two parts:
* A hash table that contains a fixed array of 'm' pointers to storage table entries.
* Storage table entries organized into separate linked lists called buckets.

•Hash function is used for the mapping of a key value and the slot where that value belongs to the hash table.

•The hash function takes any key value from the collection and computes an integer value from it in the range of slot names, between 0 and m - 1.

# Linked List and Tree Structure Organizations Linear List - simple list

•Linear list organization is the simplest and easiest way to implement the symbol tables.

• It can be constructed using single array or equivalently several arrays that store names and their associated information.

•During insertion of a new name, we must scan the list to ensure whether it is a new entry or not.

•If an entry is found during the scan, it may update the associated information but no new entries are made.

If the symbol table has 'n' names, the insertion of new name will take effort proportional to 'n' and to insert 'n' names with 'm' information, the total effort is 'cn (n+m)', where 'c' is a constant representing the time necessary for a few machine operations.

•The advantage of using list is that it takes minimum possible space. On the other hand, it may suffer for performance for larger values of 'n' and 'm’.

# Self-Organizing List

* Searching in symbol table takes most of the time during symbol table management process.
* The pointer field called 'LINK' is added to each record, and the search is controlled by the order indicated by the ‘LINK’.
* A pointer called 'FIRST' can be used to designate the position of the first record on the linked list, and each 'LINK' field indicates the next record on the list.
* Self-organizing list is advantageous over simple list implementation in the sense that frequently referenced name variables will likely to be at the top of the list.

•If the access is random, the self-organizing list will cost time and space.

# Search Trees

• Symbol tables can also be organized as binary tree organization with two pointer fields, namely,

'LEFT' and 'RIGHT' in each record that points to the left and right sub trees respectively.

•The left sub tree of the record contains only records with names less than the current records name.

•The right sub tree of the node will contain only records with name variables greater than the current name.

•The advantage of using search tree organization is that it proves efficient in searching operations, which are the most performed operations over the symbol tables.

•A binary search tree gives performance compared to list organization at some difficulty in implementation.

# Allocation Data Structures

Allocation strategy is an important factor in efficient utilization of memory for objects, defining their scope and lives using either static, stack, or heap allocations.

# Stack Allocation

* Stack is a linear data structure that satisfies last-in first-out (LIFO) policy for its allocation and de allocation.
* This makes only last element of the stack accessible at any time.

•Implementing stack data structure requires use of Stack Base (SB) that points to first entry of stack, and a Top of Stack (TOS) pointer to point to last entry allocated to stack.

# Stack Allocation for Activation Records

* The stack allocation is based on the principles of control stack in which entries are Activation Records (ARs) of some procedure.
* ARs are pushed and popped on each call (activations) and return (the end of procedure) respectively.
* On each activation request, the memory is allocated on the TOS and pointer is incremented by the size of allocation.

•On execution of return from the procedure, AR is deallocated and TOS is decremented by the same size.

# Heap Allocation

• Heaps are a kind of non-linear data structure that permits allocation and deallocation of list of entities in any (random) order as needed.

•Heap data structure returns a pointer to allocated area in heap for an allocation request.

•Hence, an allocated entity must maintain a pointer to the memory area allocated to it.

•Space allocation strategy using heaps is optimized for performance by maintaining free areas and implementing policies such as first-fit and best-fit for new object allocation