Introduction to Compilation

Dr. S.P.Khandait

Compiler construction is a broad field. In the early days, much of the efforts in compilation was focused on how to implement high-level constructs. Then, for a long time, the major emphasis was on improving the efficiency of generated code. These topics remain important even today, but many new technologies have caused them to become more specialized. Following sections introduces about the process of compilation,a detailed discussion on the phases of compilation and describes the tools that can the be used for compilation along with suitable examples.

Compiler Basics

The process by which software accepts a text in a certain language as input and produces a text in another language, while preserving the meaning of that text, is called translation. While the input language is called the source language, the output language is called the target or object language. A translator is a generalized form of compiler. The language in which the translator is written is called the implementation language. When the object language is a low level language, such a translator is called a compiler. This conversion process is essential for the hardware to interpret and perform the input program.

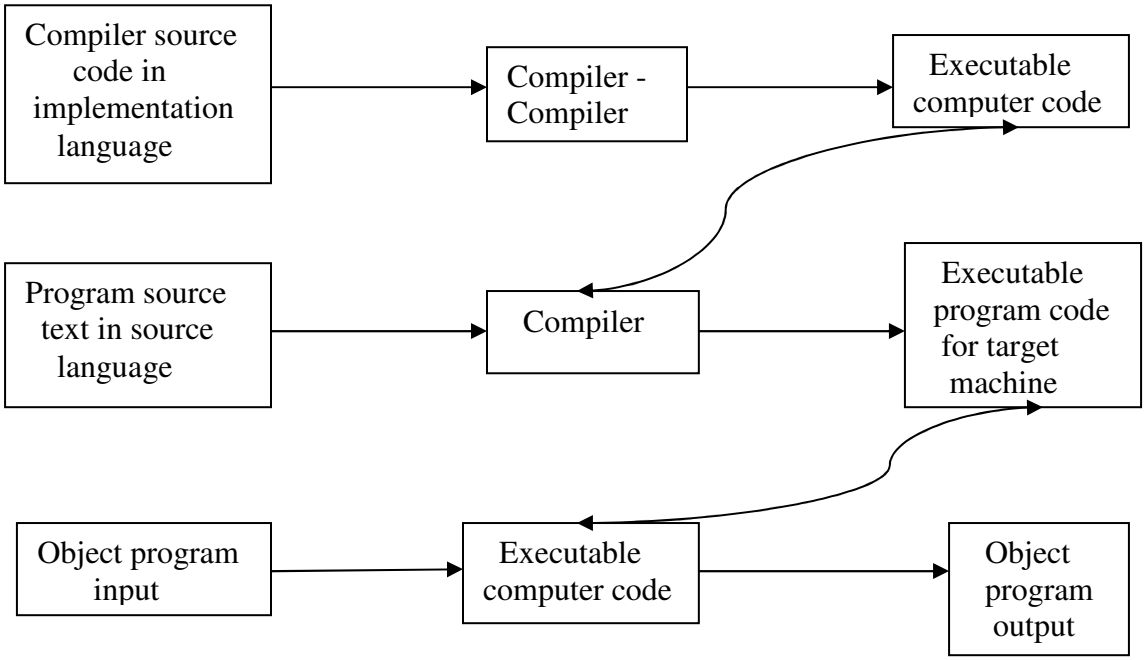


Fig. 1 Compiling and running a compiler

To obtain the compiler , its source text serves as an input to another compiler which produces an executable code for the source text. This is called as the executable file of the compiler. Then the source program input by the user is compiled by this compiler to produce object code, which is loaded into the memory and executed. This process of compiling and running a compiler is depicted in Fig.1 When the source language is also the implementation

language and source text to be compiled is actually a new version of the complier itself it process is called bootstrapping

Thus the compiler is just a giant file conversion program. To carry out its translation functionality, the semantics and the grammar of the source and target languages should be specified.

Study of design of compilers is essential due to the following reasons:

1. Compiler construction is a very successful branch of computer science. Some of the reason for this are listed below:
   - When the problems are properly structured, the compilers can obtain input, construct a semantic representation, and synthesize their output from it. Thus the analysis-synthesis paradigm of compilers is very powerful and widely applicable. For example, a program for tallying word length in a text consists of a front end which analyzes the text and constructs a table of (length, frequency) pairs, and a back end which prints this table.
   - For some parts of compiler construction excellent **standardized formal-isms** have been developed, which greatly reduce the effort to produce these parts. The best examples are regular expressions and context free grammars, used in lexical and syntactic analysis.
   - Once the methods are formalized, automatic program generator tools can be used to **generate** parts of the programs that are formally represented. For example lexical analyzers can be generated (syntax descriptions) and code generators can be generated from machine descriptions. All these are generally reliable, easier to debug and are at a higher level of abstraction than their handwritten counterparts.
2. Compiler construction has a **wide applicability**. It can be applied to reading structured data, file conversion problem and software testing.
3. Compiler can also aid in the **study** of useful algorithms and data structures that are used in their construction. It includes hashing methods, tables, stacks, garbage collection schemes, dynamic programming methodologies and graph algorithms.
4. The interfaces to software packages involve the use of a **command language**. It can be interpreted or compiled.
5. **Automated tools** need to be developed when software is migrated from old language to the new programming languages. For this, software processers should be written to perform analyses of the source program and to translate it into the target programming languages
6. Automated tools need to be developed to rewrite the software developed using old design methodologies to current methodologies. This process of **re-engineering** enhances the maintainability of the software. A compiler can be used for this purpose.

Issues in Compilation

The following complex issues have to be considered in the design due to the nature of a higher level programming language (HLL):

1. As the HLL has complex expressions, **hierarchy of operations** has to be maintained to determine the correct order of evaluation of the expressions.
2. Maintaining data type integrity **with automatic** type conversions is a must. This is because each part of a complex expression can be made up of different types.

3. User defined types like struct, enum, union are new types defined and used in a source program. The compiler has no prior knowledge about the nature of such types. To handle these **user defined data types,** their definitions must be 'remembered' by the compiler.
4. The compiler must develop appropriate **storage mappings** for the data structures used in a program by utilizing the knowledge about the allocation of memory for the data.
5. A block structured source program consists of different **name spaces.** The compiler must resolve the occurrence of each variable name in a program to determine the name space to which a referenced variable belongs to. For this purpose, a compiler should construct separate symbol tables and should be searched to resolve a variable name.
6. Compiler should have facilities to handle different **control structure** like 'if-then-else', 'for', 'while' etc. For example, to handle a 'while' statement, the compiler should have facilities to increment the loop variable and terminate the loop. Further, a check should be made prevent the flow of control from outside to inside the loop.
7. **Optimization** is performed by re-arranging the computations in a program such that the transformed program is semantically equivalent to the original program, but executes faster and/or occupies less space. Some of the optimization methods are elimination of common sub expression, elimination of loop invariants etc.

Analysis Synthesis Model

While designing language translator, the unit of the source program on which the translation model is applied has to be considered. If the unit chosen is an expression, then it is completely analyzed before generating a code for it. This unit is usually called a pass. A compiler pass is one complete scan of the source program or its equivalent representation. Fig. 2 depicts passes i and i+1 in the compilation process. The code generated by pass i is used as input for pass i + 1. The process of compilation may involve single or multiple passes.

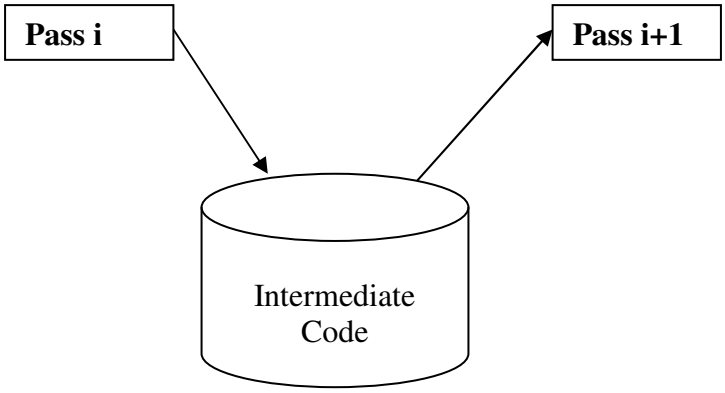


Fig.2 Passes in Compilation Process

The following are the advantages of having multiple passes in the compiler.

1. **Forward reference :** Consider a forward jump using an unconditional goto statement (eg. Goto 100. Let the current address be 10). The goto 100 statement cannot be compiling the current statement. This problems posed by forward references can be solved by performing storage allocation in the first pass and generating code in the next pass.

2. **Storage limitation :** A pass generates an intermediate code which reflects the effects of its processing. This code is stored in a disk file. The next pass reads this code from the disk for processing it. Since the program parts of different passes overlay each other, memory requirements are reduced.
3. **Optimization :** The intermediate code generated by the source program can be used for optimization by constructing control and data flow graphs. It can be used to produce a smaller and more efficient target code.

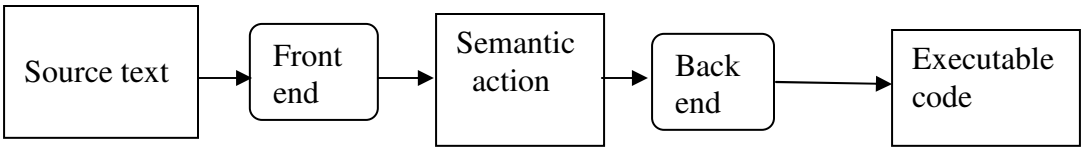


Fig. 3 Conceptual structure of a compiler

Thus in the translation model, the phases of compilation connected with code analysis form one pass and the remaining portions form the next pass of the compiler. The fundamental translation model called analysis-synthesis model, consists of a two step processing of a source program as shown in Fig.3. They are:

1. **Analysis of source program:** This process is carried out by the front end of the compiler. It determines the meaning of the source string. The actions of the front end are thus defined by the grammar of the given language. The front end is isolated from the back end and can work independently due to the separation of the compilation process into phases.
2. **Synthesis of target program:** This process is carried out by the back end of the compiler. It constructs an equivalent target string from the source string using the semantic actions produced by the front end of the compiler. Its actions thus depends on the pragmatic aspects (aspects concerning the execution environment) of the compiler which includes the features of target machine and operating system.

Each pass is divided into various sub-steps. These sub-steps are termed as the **phases** of the compiler. A phase is defined as a logically cohesive unit that takes as input one representation of the source program and produces as output another representation of the source program. The various sub-steps in analysis pass are:

1. Determination of the lexical constituents in a source string. It is also called as the **lexical analysis** phases of the compiler.
2. Determination of the structure of the source string. It is also called as the **syntax analysis** phases of the compiler.
3. Determination of the meaning of the source string. This is also called as the **semantic analysis** phases of the compiler.

The various sub-steps in the synthesis of the target program are:

1. Optimization of code.
2. Allocation of memory.

3. Generation of code.

Some examples of tools used in analysis part of compilation are listed below:

1. **Structure editors:** These editors can take up additional functionalities when compared to ordinary editors. It accepts an input sequence of commands to build the source program. It analyzes the program text and represents it as an hierarchical structure. It also checks for keywords and parenthesis matching.
2. **Pretty printers :** It helps in analyzing and restructuring the program. It realigns the structure of the program, changes its font and prints it with indentations.
3. **Static Checker:** It reads and analyses the program and attempts to discover potential bugs in it. For example, it can identify dead portions of a code.
4. **Interpreter:** It performs the operations given by the source program after scanning. The process of synthesis traverses the tree and executes the operation at each node (rather than generating a code for it) .
5. **Text formatters:** It takes as input a stream of characters, which have to be type set along with the commands to indicate the paragraphs, figures and mathematical structures.
6. **Silicon Compiler :** It has source language similar to the conventional programming language. Here the variables represent the logical signals (0 or 1) in a switching circuit.
7. **Query Interpreter:** It translates a predicate language having relational and boolean operations into commands to search the database for records satisfying the predicate.

Cousins of the compiler

In this section we discuss the software which aids the compilers. The macro processors. Produce input to the compiler and the assemblers process the output of the compiler. The compilation tool chain is given below:

[preprocessor] ⟶ [compiler] ⟶ [assembler] ⟶ [linker] ⟶ [loader] ⟶

A description of the cousins of the compiler is provided in this section

1. **Preprocessors :** Preprocessors have the capability to augment the source language by providing the ability to include files and expand macros. Often the "preprocessor" inserts procedure calls to implement the extensions at runtime. They perform the following functions:

a. **Macro processing:** Macros are short hands for longer constructs. A macro processor has to deal with two types of statements –macro definition and macros expansion. Marco definitions have the name of the macro and a body defining the macro. They contain formal parameters During macro expansion, these formal parameters are substituted for the actual parameters.

b**. File inclusion:** These preprocessors can also include contents of a file in a program text.

c. Rational **Preprocessors:** They augment older languages with modern flow –of control and data structuring facilities

d**. Language extensions:** These preprocessors attempt to add capabilities to the language. For example adding database query facilities to the language.

2. **Assemblers:** Assembly code is a mnemonic version of machine code in which names, rather than binary values, are used for machine instructions and memory addresses. An assembler needs

to assign memory locations to symbols (called identifiers) and use the numeric location addresses in the target machine language produced. The assembler should take care that the same address must be used for all occurrences of a given identifier and two different identifiers must be assigned two different locations. Conceptually the simplest way to accomplish this is to make two passes over the input. During the first pass, each time a new identifier is encountered, an address is assigned and the pair (identifier, address) is stored in a symbol table, during the second pass, whenever an identifier is encountered, its address is looked up in the symbol table and this value is used in the generated machine instruction.

Consider the following trivial C function that computes and returns the xor of the characters in a string.

```
int  xor (char  s[ ])
{
     int  ans  = 0 ;
     int  i  = 0 ;
    while  (s [ i ]   !=  0)
    {
        ans  =  ans   ^  s [ i ] ;
         i   =   i  +  1 ;
    }
     return ans ;
}
```

The corresponding assembly language code is listed below:

```
.globl   xor
    xor :
         subl       $8,    &esp
         movl       $0,    4 (&esp)
         movl       $0,       (&esp)
    L2 :
         movl      ( &esp ) ,      &eax
         addl      12 (&esp ) ,    &eax
         cmpb      $0 ,     (&eax)
         je    L3
         movl      ( &esp ) ,    &eax
         addl      12 (&esp ) ,    &eax
         movsbl      ( &eax ) ,     &edx
         leal         4( &esp ) ,    &eax
         xorl          &edx ,         ( &eax )
         movl          &esp ,     &eax
```

```
        incl        ( &eax )
        jmp L2

L3 :
    movl    4 ( &esp ) ,  &eax
    addl    $8 ,    esp
    ret
```

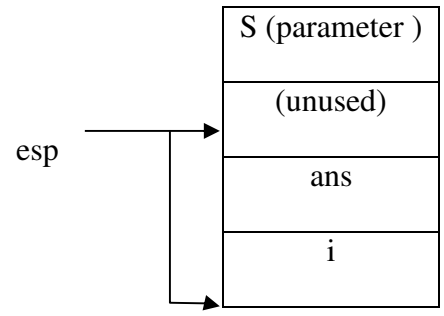


Fig.4 Stack contents

The operation of the program shown in Fig. 4 is given which includes :

1. The stack pointer (esp) originally points to the (unused) frame pointer. Just above it is the parameter s (Fig.4)
2. Allocate (by moving the stack pointer) and initialize the local variables.
3. Add 's' (an address) to 'i', thus giving the address of s [i].
4. Compare the contents of the above address (i.e., s[i] to 0) and break out if appropriate.
5. Calculate s [i] xor ans. The code actually performs the calculation in the memory location containing ans.
6. Increment 'i' and loop.
7. When the loop ends store the return value (in eax), restore the sack pointer and return.

3. Linkers : Linkers or linkage combine the output of the assembler (object code) for several different

compilations. The linker has also another input – the libraries. To the linker the libraries look like other

programs compiled and assembled. The two primary tasks of the linker are :

1**. Relocating relative addresses :** The assembler processes one file at a time. Thus the symbol table produced while processing file A is independent of the symbol defined in file B, and conversely. Thus it is likely that the same address will be used for different symbols in each program. Thus when the (local) addresses in the symbol table for file A are relative to file A, they must be relocated by the linker. This is accomplished by adding the starting address of file A(which in turn is the sum of the lengths of all files processed previously in this run) to the relative address.

2. Resolving external references : Assume procedure f in file A and procedure g in file B, are compiled (and assembled) separately. Assume also that f invokes g Since the compiler and assembler do not see g when processing f, it appears impossible for procedure f to know where is the memory to find g. For this purpose, the output of the file a compilation indicates that the address of g is needed. This is called use of g. This is called the definition of g. The assembler

passes this information to the linker. The simplest linker technique is to make two passes. During the first pass, the linker records in its "external symbol table" (a table of external symbol) all the definitions encountered. During the second pass, every use can be resolved by access to the table.

4. Loaders : After the linker has done its work, the resulting "executable file" can be loaded into the central memory by the operating system. The process of loading takes a relocatable code, alters the relocatable addresses and places the altered instructions and data in memory at proper locations. This process is dependent on the intricacies of the operating system. In the early single-user operating systems, all programs would be loaded into a fixed address and the loader simply copies this file to memory. In the modern operating systems, this process is much more complicated since (parts of) many programs reside in memory at the same time.

5. Debuggers : A debugger is a program that is used to determine execution errors in a compiled program. It is often packaged with a compiler in an Integrated Development Environment (IDE). It keeps track of all the source code information like line numbers and names of variables and procedures. It can also halt an execution at prespecified locations called breakpoints.

6. **Profiler :** A profiler is a program that collects statistics on the behavior of the object program during execution. It includes the number of times a procedure is called and its execution time. It helps to improve the efficiency of the object code generated.

7. **Project Managers :** Modern software projects are coded in a team. So the source code produced by different members of the team should be properly coordinated and merged. For this purpose project managers are used. Examples of these programs are source code control system (sccs) and revision control system (rcs).

Phases of compilation

A compiler takes as input a source program and produces as output an equivalent sequence of machine instructions. As this process of compilation is highly complex, it is split into a series of sub processes called phases. A phase is a logically cohesive operation that takes as input one representation of the source program and produces as output another representation. Broadly, the activities of compilation are split into analysis part and synthesis part as discussed previously. In compilation, analysis is split into **linear analysis, hierarchical analysis and semantic analysis.** They are represented by the lexical analysis (scanning),syntax analysis (parsing) and semantic analysis phases of the compiler. The analysis phase generates an intermediate code to describe the sequence of steps needed to implement the program. It also generates a table of information, describing the symbols in the source code. Further, the analysis part also detects lexical, syntax and semantic errors in the source program. The details of the various phases in analysis portion of the compiler are described below:

1. Lexical analysis: This phase performs the linear analysis on the source program. It reads a stream of characters making up the source program from left to right and groups them into tokens. A tokens is defined as a sequence of characters that have a collective meaning. For each token identified as a sequence of characters that have a collective meaning. For each token identified, this phase also determines the category of the token as identifier, constant or reserved words and its attribute that identifies the symbol's position in the symbol table. For example consider the following statement.

a = (b+c) * (b+c) * 2

The various tokens and their attributes are listed in the table 1

Table 1 Tokens for the expression a = (b+c) * (b+c) * 2

Symbol	Category	Attribute
a	Identifier	#1
=	Operator	Assignment (1)
b	Identifier	#2
+	Operator	Arithmetic
c	Identifier	#3
*	Operator	Arithmetic
(	Operator	Open parenthesis
)	Operator	Closed parenthesis
2	Constant	#4

Here, #2 etc. represent the position of the symbols (variables and constants) in the symbol table. When '+','-',=' etc are used, its category is operator with code 1.

2. **Syntax Analysis :** This phase performs hierarchical analysis on the source program. Here, the tokens are grouped into hierarchically nested collections with collective meaning called **expressions** or **statements.** It determines the structure of source language. This hierarchical structure is defined as using recursive definitions. These recursive definitions are represented by using Context Free Grammars (CFG). These represent the grammar / syntax of the language. These grammatical phrases are represented in the form of a parse tree. The parse tree describes the syntactic structure of the input. The terminal nodes of the parse tree represent the tokens and the interior nodes represent the expressions. The parse tree for the above expression is given in Fig 5.

   Syntactic structures can also be represented by using syntax trees. A syntax tree is a compressed representation of the parse tree, where the operation appears as interior nodes and the operands for this operator appear as their children. The syntax tree for above expression is shown in Fig 5. The difference between parse **tree and syntax tree** is listed as follows:

   - A syntax tree is a compressed representation of a parse tree.
   - The interior nodes in a syntax tree represent an operator, whereas the interior nodes in a parse tree represent an expression.
   - The leaf nodes of a syntax tree represent the operand, whereas the leaf nodes in parse tree represent the tokens.

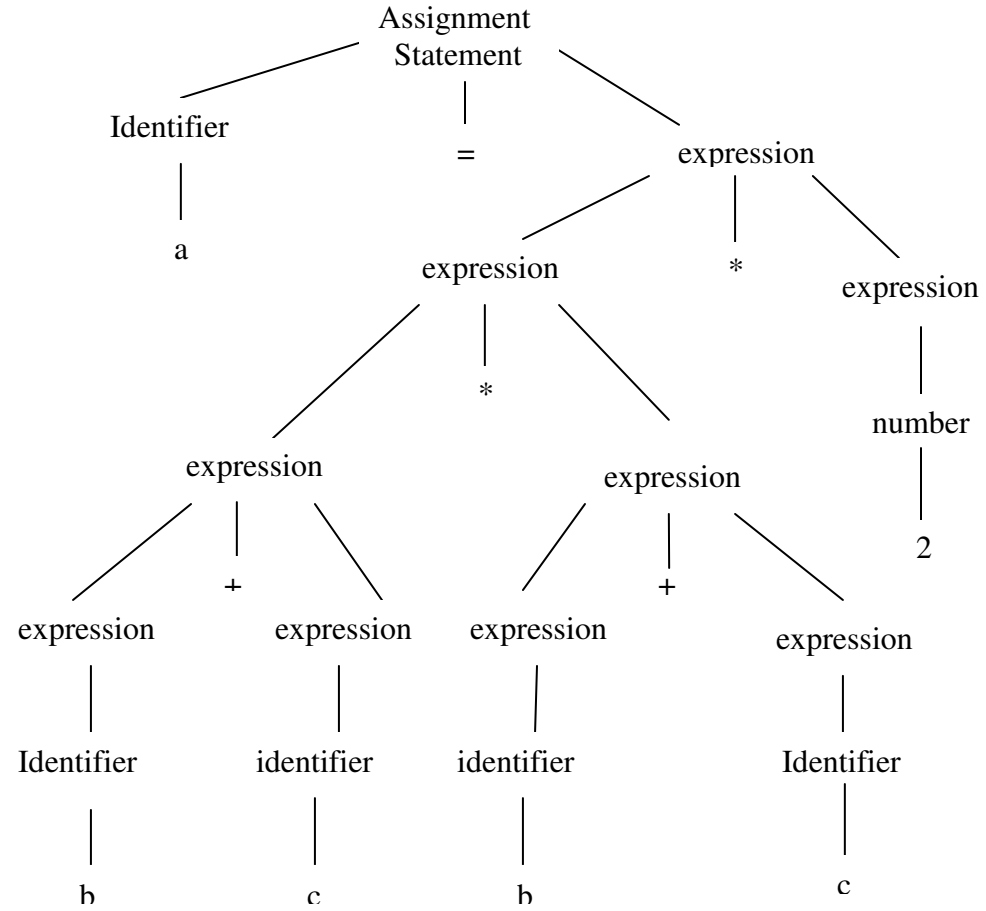


Fig. 5 Parse tree for expression a = (a+c) * (b+c) * 2

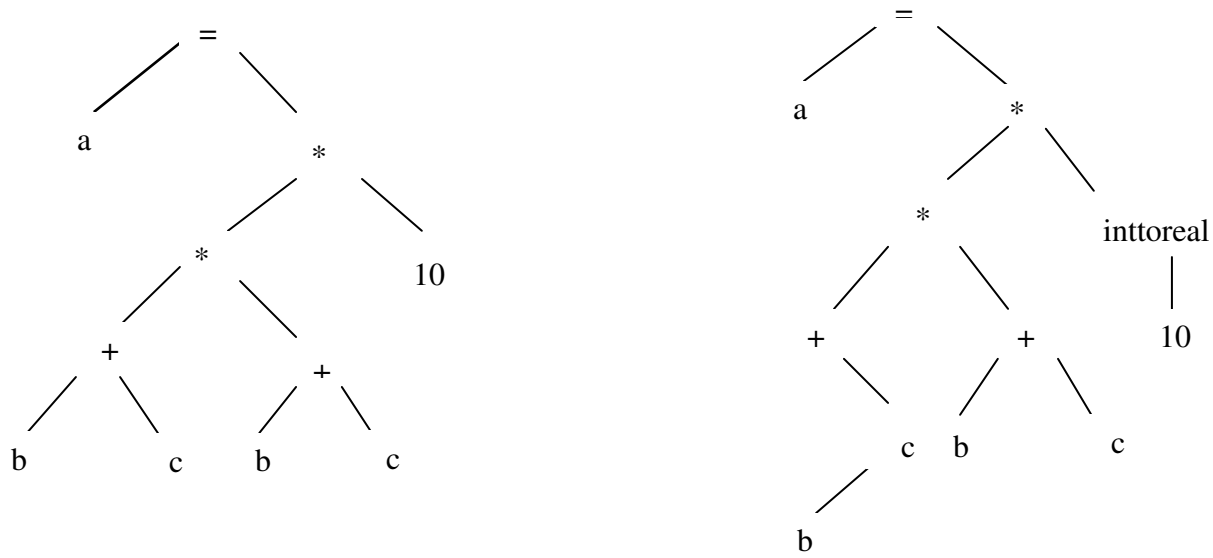


Fig.6a) Syntax tree for expression a = (b+c) * (b+c) * 2 **Fig.6b) Its Annotated tree**

3. **Semantic Analysis :** The goal of semantic analysis is to determine the **meaning** of a source string. It checks the source program for semantic errors and gathers the type of

information that can be used in subsequent phases of compilation. Type checking for operation is also performed during this phase. For example consider the expression 1. Let the types of b and c be real. So the constant 2 has to be converted into real and then multiplication operation is performed. This syntax tree after semantic analysis is called as **annotated tree** and is represented in Fig 6b. This process of semantic analysis of an assignment statement can be expressed by the following algorithm;

1. Analyze the syntax tree in a bottom manner
   a. Consider an elemental tree
   b. Append attributes; e.g. type, to operands in the tree
   c. Analyze for correctness of usage
2. Determine the 'meaning' in the sense of actions involved in evaluating the elemental tree.
3. Transform the tree to reflect the effects of the actions. The transformed tree indicates the remaining actions in the meaning of the tree.
4. Repeat steps 1-3 until the entire tree is analyzed.

4. Intermediate code Generation: It is a part of the synthesis process of the compiler. The intermediate code is the representation for an abstract machine. Using the intermediate code, optimization and code generation can be performed. An intermediate code should possess the following properties :

- Intermediate code should be easily generated from the semantic representation of the source program.
- It should be easy to translate the intermediate code to target language.
- It should be capable of holding the values computed during translation.
- It should maintain precedence ordering of the source language.
- It should be capable of holding the correct number of operands of the instruction.

There are different forms of intermediate codes like quadruples, triples etc. Example of three address code (TAC) for the syntax tree in Fig. 6 b is given below:

```
T1  =  inttoreal (2)
T2  =  b + c
T3  =  b + c
T4  =  T2  *  T3
T5  =  T4  *  T1
 a  =  T5
```

5. Code Optimisation : The main aim this phase is to improve on the intermediate code to generate a code to generate a code that runs faster /or occupies less space. It is used to establish a trade off between compilation speed and execution speed. For example the output of the previous phase can be modified to calculate (b+c) only once and to reuse its values as shown below:

```
T1  =  inttoreal (2)
T2  =  b + c
T3  =  T2  *  T2
T4  =  T3  *  T1
 a  =  T4
```

6. Code Generation : The main aim of this phase is to allocate storage and generate relocatable machine /assembly code. Memory locations and registers are allocated for

variables. The instructions in intermediate code format are converted into machine instructions. Hence it involves the pragmatics or machine dependent aspects The target code generated for the expression 1 when the first operand is for source/destination and the second operand is only source operand (for operations like add, Sub etc.) and Mov a, b means a b is given below:

```
MOV      b,    R2
ADD      R2,   c
MUL      R2,   R2
MUL      R2,   #2 . 0
MOV      R2,   a
```

7. Target Code Optimisation : The compiler also attempts to improve the target code generated by the code generator by choosing proper addressing modes to improve the performance, replacing slow instruction by fast once and eliminating redundant instructions. For example, "MUL R2, # 2.0"instruction can be replaced by SHL (shift left instruction) as shown below :

```
MOV      b,    R2
ADD      R2,   c
MUL      R2,   R2
SHL      R2
MOV      R2,   a
```

Other modules involved in compilation are :

1. **Symbol Table Management :** A symbol table is a data structure that contains a record each is identifier with field for the attributes of the identifier. This data structure has facilities to manipulate (add / delete) the elements in it. The type information about the identifier is detected during lexical analysis phase and is entered into the symbol table. This information is used during the intermediate code generation phases of the compiler to verify type information. A sample structure of the symbol table is given in Table 2a:

Table 2a : symbol Table

Address	symbol	attribute	Memory Location
1	a	Id, real	1000
2	b	Id, real	1100
3	c	Id, real	1110

2. Literal Table Management : A literal table maintains the details of constants and strings used in the program. It reduces the size of a program in memory by allowing reuse of constants and strings. It is also needed by the code generator to construct symbolic addresses for literals.

Table 2b : Literal Table

Address	symbol	attribute	Memory Location
4	2	Const, int	1200

3. Error Detection and Reporting : Each phase encounters errors. After detecting the errors, this phase must deal with the errors to continue with the process of compilation. A list of errors encountered by various phases is given below:

1. Lexical analyser : Misspelled tokens.
2. Syntax analyser : Syntax errors like missing parenthesis.
3. Intermediate Code Generator: Incompatible operands for an operator.
4. Code Optimizer: Unreachable statements.
5. Code Generator: Memory restrictions to store a variable. For example, when the value of an interger variable exceeds its size.
6. Symbol Tables : Multiply declared identifiers.

Example 1.1 : Show the output of all the phases of the compiler for the following line of code:

A[index] = 4 + 2 + index ;

1. **Output of lexical analyzer** are the tokens and their types :

A	-	identifier	4	-	number
[	-	left bracket	2	-	number
Index	-	identifier	+	-	plus sign
]	-	right bracket			
=	-	assignment			

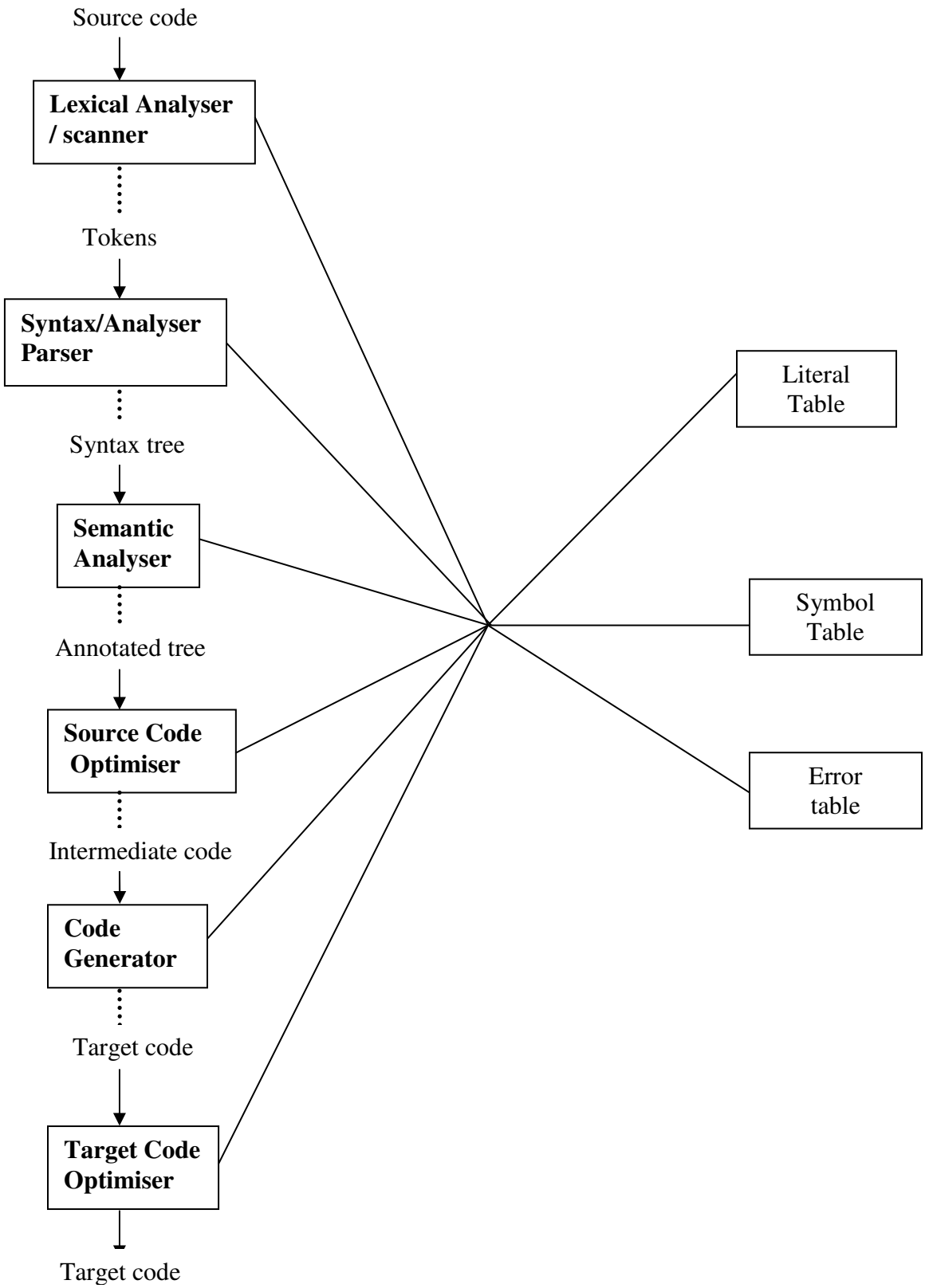
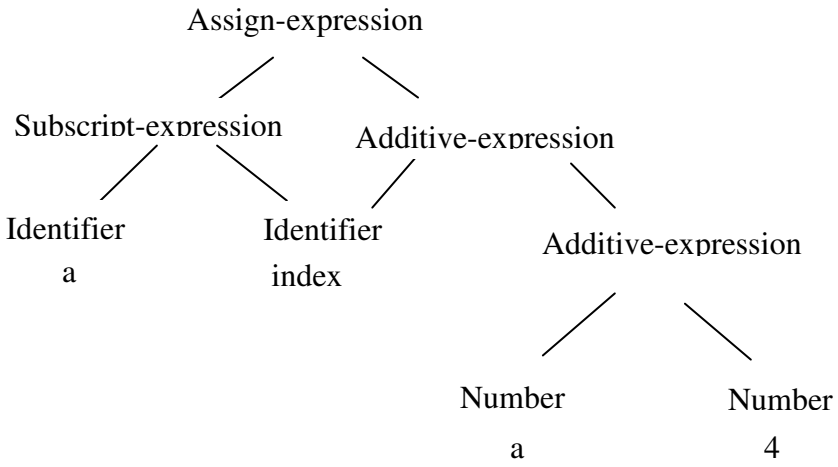
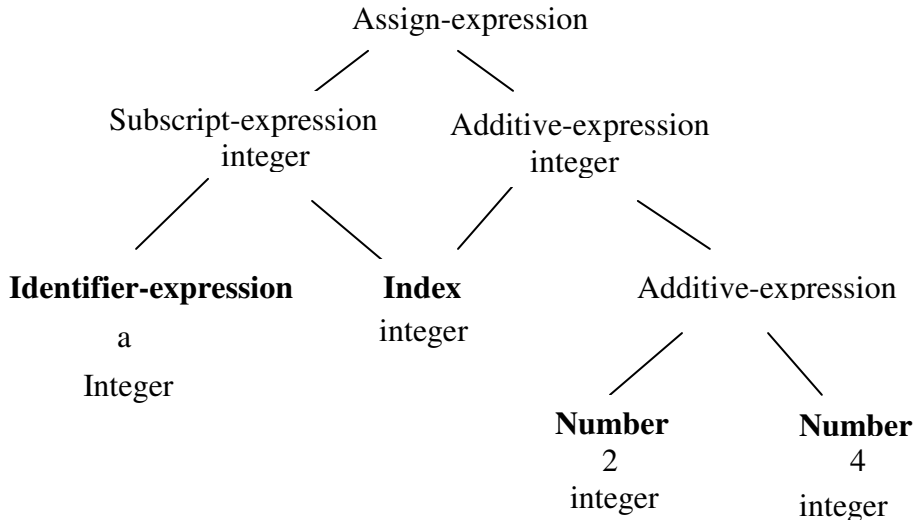


Fig. Phases of compilation

2. **Output of syntax analyzer** is the parse tree/ syntax tree as shown below:



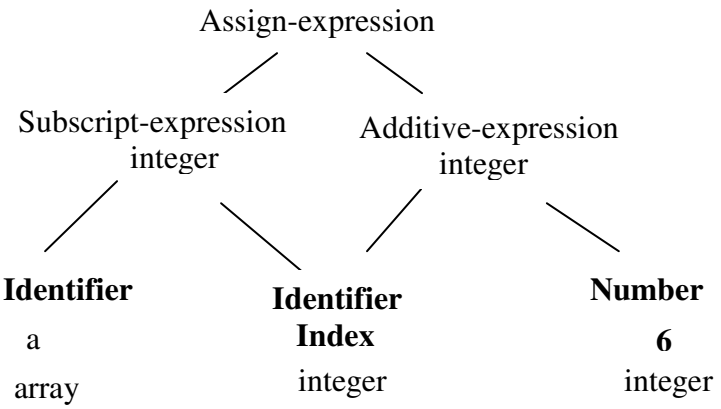
3. **Output of the semantic analyzer** is the annotated tree as shown below:



4. **Intermediate code generated** is

```
t1          =       4   +   2
t2          =       t1  +   index
a (index)   =       t2
```

5. **Code optimization :** Here the annotated parse tree can be represented as follows by combining the constants 2 and 4.



The intermediate code produced is as follows:

t1 = 6 + index
a (index) = t1

6. Target Code and optimized optimized target code are generated as follows:

```
MOV   index,    R0          //  index value is moved into  R0
MOV    &a,  R1        //  address   of    'a' is moved into  R1
ADD    R0,  R1        //  calculate the position of the array element
MOV    #6,   R2       //   store constant   6  into   R2
ADD    R0,   R2       //   find    6  +  index
MOV    R2,   7R1      //   move the value in R2 register to location
                      //   specified by   R1.
```

7. Contents of the symbol table are as follows

Address	**symbol**	**attribute**	**Memory Location**
1	a	array	1000

2	index	id, real	1100

8. Contents of the literal table are as follows

Address	symbol	attribute	Memory Location
1	2	integer	1002
2	4	integer	1104

Grouping of the phases

A number of phases can be grouped into a pass, so that their activities can be interleaved together in a pass, Usually the front end consists of phases that depend upon the source language and are independent of the target language. It includes lexical analysis, syntactic analysis, creation of symbol tables, semantic analysis and generation of intermediate code. Some amount of code optimization and error handling can also be included in this phase. The back end includes the phases of compilation that depends on the target machine. It includes code optimization, symbol table operation error handling and code generation

When lexical, syntax and intermediate code generation phases are grouped into a pass, the tokens stream obtained after scanning can be translated into intermediate code. The syntax analyzer calls the scanner to get the tokens as and when needed to fix up the syntactic structure of the language.

1. When the number of passes is reduced, the time taken to read and write intermediate files to or
   from the disk can be reduced.
2. On reducing the number of passes, the entire information of the pass has to be stored in the
   temporary memory. This increases the memory space needed to store the information.
3. When phases have more coupling among them, it is advantageous to group them together. For
   example, lexical analysis and syntax analysis phases are grouped together as the lexical analyzer
   can fetch the token as and when requested by the syntax analyzer.
4. Related phases should only be grouped together :

a) Code generation cannot be performed until an intermediate representation is available for the source code. So it cannot be grouped with the syntax analysis phases.

b) Intermediate and target code generation can be merged into a single pass by using technique of backpatching. Here the addresss of the branch instructions can be left blank and can be filled in when the information is available (when the instruction is available).

Execution of a program

There are two popular models for program execution. They are:

1. Translation, linking and loading of a program.
2. Interpretation of a program

This sub-section provides a discussion on the features of compilation and interpretation and differentiates

Translation, linking and loading of a program

Fig 1.7 shows the model of translation, linking and execution of a program. In this model, the translator processes the source program to generate an object module which is in a relocatable machine language form. The loader processes the object module generated for the source program along with library files to produce an executable form of the program. This form is called the binary program. The loader performs two important function. They are relocation of the object modules and linking of the object modules to resolve mutual references.

An absolute loader loads the binary form of the program in the computer's memory and organizes its execution. The different forms of the program, namely the source program, the object module and the binary program, can be stored in the libraries of the computer system for repeated use. Two widely used forms of translators are compilers and assemblers which translate the programs written in higher level language and assembly language respectively. An important advantage of this model is that the machine language program generated by the translator executes very efficiently. Drawbacks of this model are that the machine language program is not portable to other system, and that a source program has to be recompiled for every modification.

The interpreters are superior to compilers in a program development environment, which involves testing, debugging and editing a program. For example, consider a program with 300 statement being executed on a test data such that only 75 statements are visited during the test run. Total CPU time in compilation followed by execution of program is 300* tc75* te =307.5 *tc , while total CPU time in interpretation of the program is 75* ti = 75*tc. Hence, interpretation of the programs would be cheaper for program development environments when compared to compilation.

Self-modifying programs are programs that modify themselves or evolve during executing. For such programs interpretation is more convenient when compared to compilation.

To conclude, the following are the advantages of an interpreter:

1. It is written in high level language and hence it is portable.
2. It is easier to write.
3. It has better error checking facilities.
4. It is more secure.

A comparison of the capabilities of the compiler and interpreter is shown in table 1.3. The code generating back end is then replaced by an interpreting back end and the whole program is called an interpreter. It should be pointed out that there is no difference between using a compiler and an interpreter. In both cases the program text is processed into an intermediate form, which is then interpreted by some mechanism. In compilation, the program processing is considerable, the resulting intermediate form and machine specific binary executed code is low level, the interpreting mechanism is a hardware (CPU) and the program executes efficiently. But, in interpretation, the program processing is minimal to moderate, the resulting intermediate form, is high to medium level, the interpreting mechanism is software (a program) and the program execution is relatively slow.

Table : Comparison of interpreters and compilers

S.No.	Compiler	Interpreter
1	It has more processing	It involves processing
2	It is faster	It is slower
3	Writing a compiler is very difficult	Interpreters are easier to write
4	Interpreting mechanism is CPU	Interpreting mechanism is a software in HLL
5	Less secure	More secure
6	Intermediate code is low level	Intermediate code is HLL
7	Compilers are not portable.	Interpreters are portable.
8	Compiled code is not portable	Interpreted code is portable.

Compiler Construction Tool

Originally, compiler were written “from scratch”, but now the situation is quit different . A number of tools are available to ease the burden of construction of compilers. Compiler construction tools are the tools that have been created for automatic design of specific compiler components. These tools use specialized language and algorithms. They produce the components at a higher degree of abstraction. The following is a list of compiler construction tools:

1. **Parser Generators:** They produce syntax analyzers from Context Free Grammars (CFG). As syntax analysis phase is highly complex and consumes manual and

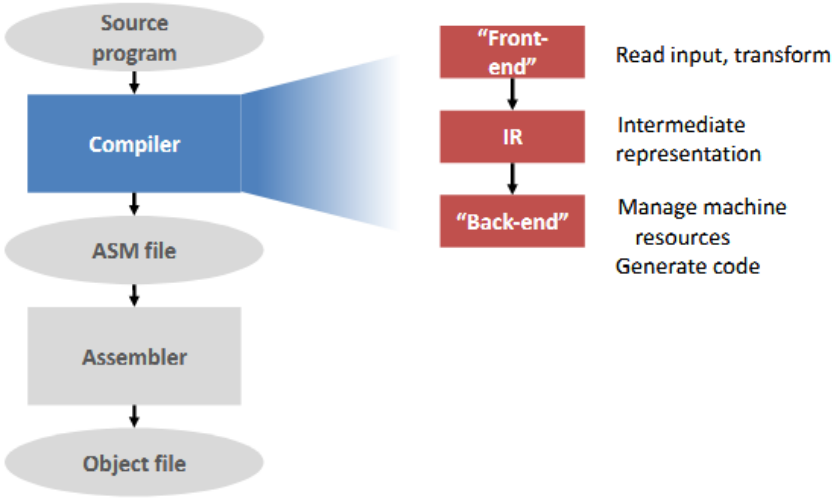
compilation time, these parser generation are highly useful. An example for this tools is yacc.

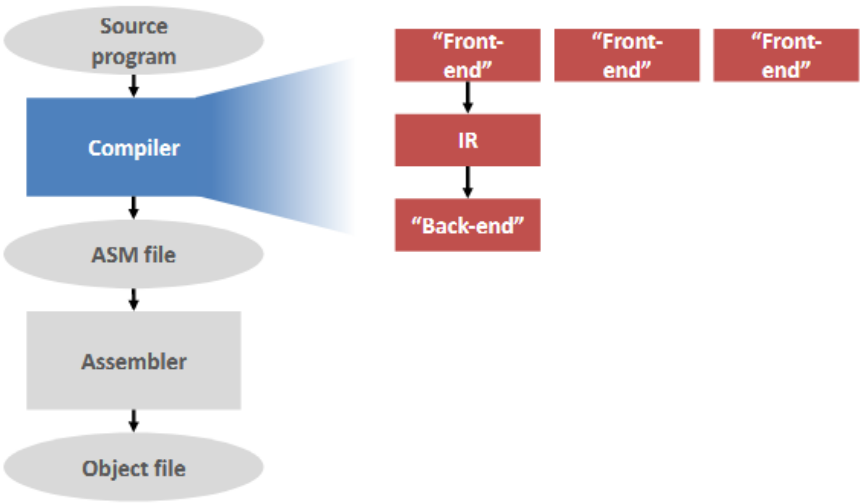
2. **Scanner Generators:** They generate lexical analyzers automatically from the language specifications written using regular expressions. It generates a finite automaton to recognize the regular expression. An example of this tool is lex.
3. **Syntax directed translation engines:** These engines have routines to traverse the parse tree and produce intermediate code. The basic idea is that one or more translations are associated.
4. **Automatic Code Generations:** These tools convert the intermediate language into machine language for the target machine using a collection of rules. Template matching process is used. An intermediate language statement is replaced by its equivalent machine language statement.
5. **Data Flow Engines :** It is used in code optimization. These tools perform good code optimization using "data-flow analysis" which gathers information that flows from one part of the program to another.

Structure of Realistic Compiler

Compiler model

- **Compilation** ***prior to execution***
  - AOT "Ahead of (Execution) Time" compilation
  - Commonly used for languages without language-specific execution environments (e.g., C, C++)
  - Available in Java as well (IBM J9, Oracle HotSpot)
- **Other model:** ***Continuous compilation***
  - JIT "Just in Time" compilation
  - Usually: optimization of methods that are frequently invoked (hot)
  - Commonly used with language virtual machines (e.g., Java VM)
    - E.g., HotSpot JVM has two JIT compilers (C1 and C2)





IR – Intermediate representation

- **Compiler-internal representation**
  - E.g., compiler must distinguish between names in different scopes
  - E.g., many programs work with variables, computers work with locations
- **Must express all language constructs/concepts**
- **Code generator maps IR to assembly code**
  - Machine code another option
- **No "best" IR – all are compromises**