COMPILER DESIGN

Motivation

- Why we study Compiler Design?
- What is the importance of Compiler?
- What is the objective of this course?

Reasons:

- Compilers provide an essential interface between applications and architectures.
- Compilers embody a wide range of theoretical techniques.
- Compiler construction teaches programming and software engineering skills.
- It teaches you how real world applications are designed.
- It brings you closer to the language to exploit it.

Reasons:

- Compiler (a sophisticated program) bridges a gap between the language chosen & a computer architecture.
- Compiler improves software productivity by hiding low level details while delivering performance.
- Compiler provides techniques for developing other programming tools, like error detection tools.
- Program translation can be used to solve other problems, like Binary translation

Isn’t it a solved problem?

- Machines have continued to change since they have been invented.

Changes in Architecture→ Changes in Compilers

* New features present new problems
* Changing costs lead to different concerns
* Must re-engineer well known solutions

Compiler construction shows us a microcosmic view of computer science.

artificial intelligence	greedy algorithms learning algorithms
algorithms	graph algorithms union-find network flows dynamic programming
theory	*dfa*'s for scanning parser generators lattice theory for analysis
systems	allocation and naming locality synchronization
architecture	pipeline management memory hierarchy management instruction set use

Inside a compiler, all these things come together.

What qualities do you want in a compiler?

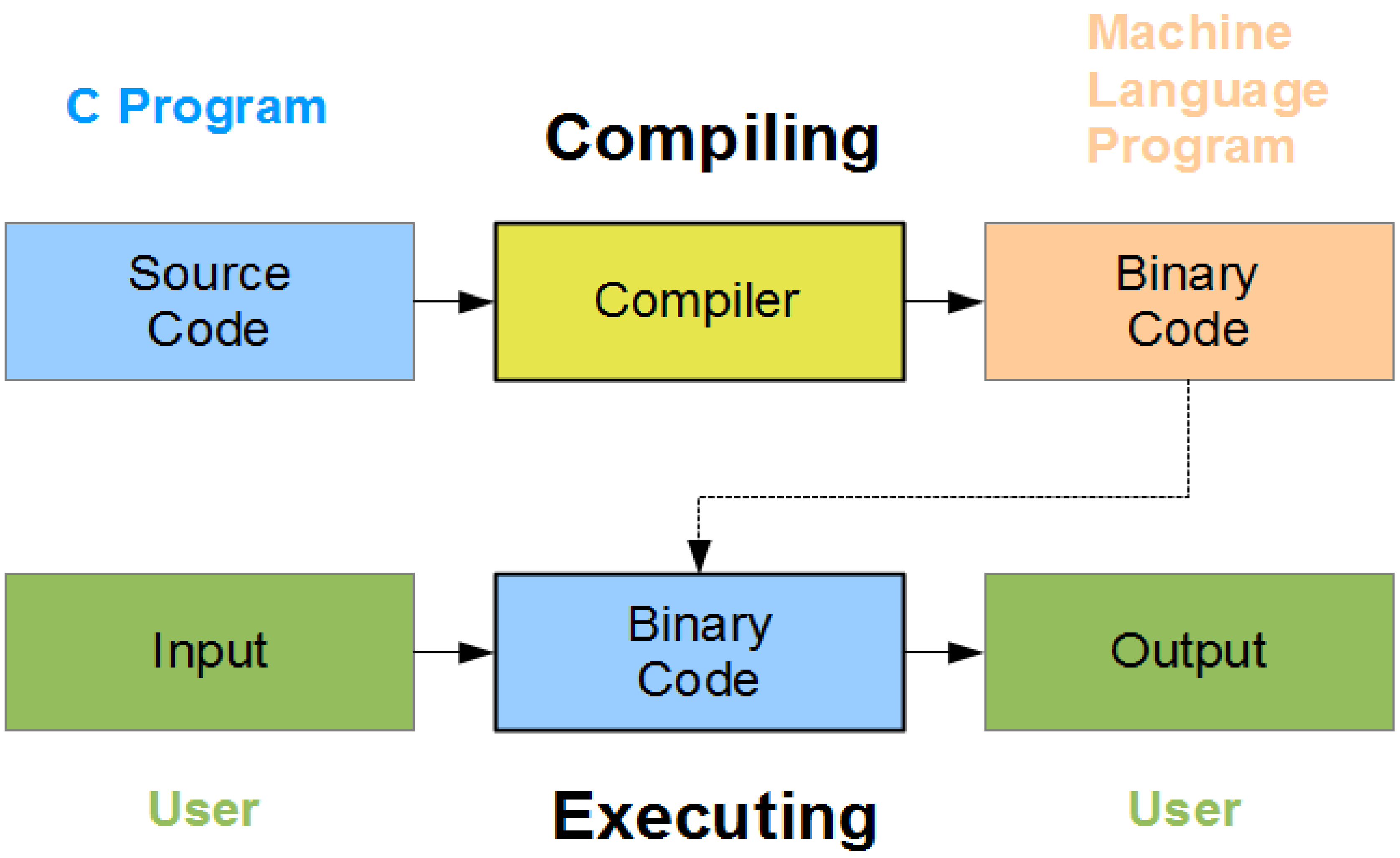
Here is a list:

1. Correct code
2. Output runs fast
3. Compiler runs fast
4. Compile time proportional to program size
5. Support for separate compilation
6. Good diagnostics for syntax errors
7. Works well with the debugger
8. Good diagnostics for flow anomalies
9. Cross language calls
10. Consistent, predictable optimization

Program Execution

- Execution of a program written in HLL is basically a 2-step process

1. The source program is compiled first i.e. translated into the object program.
2. The resulting object program is loaded into memory and executed.

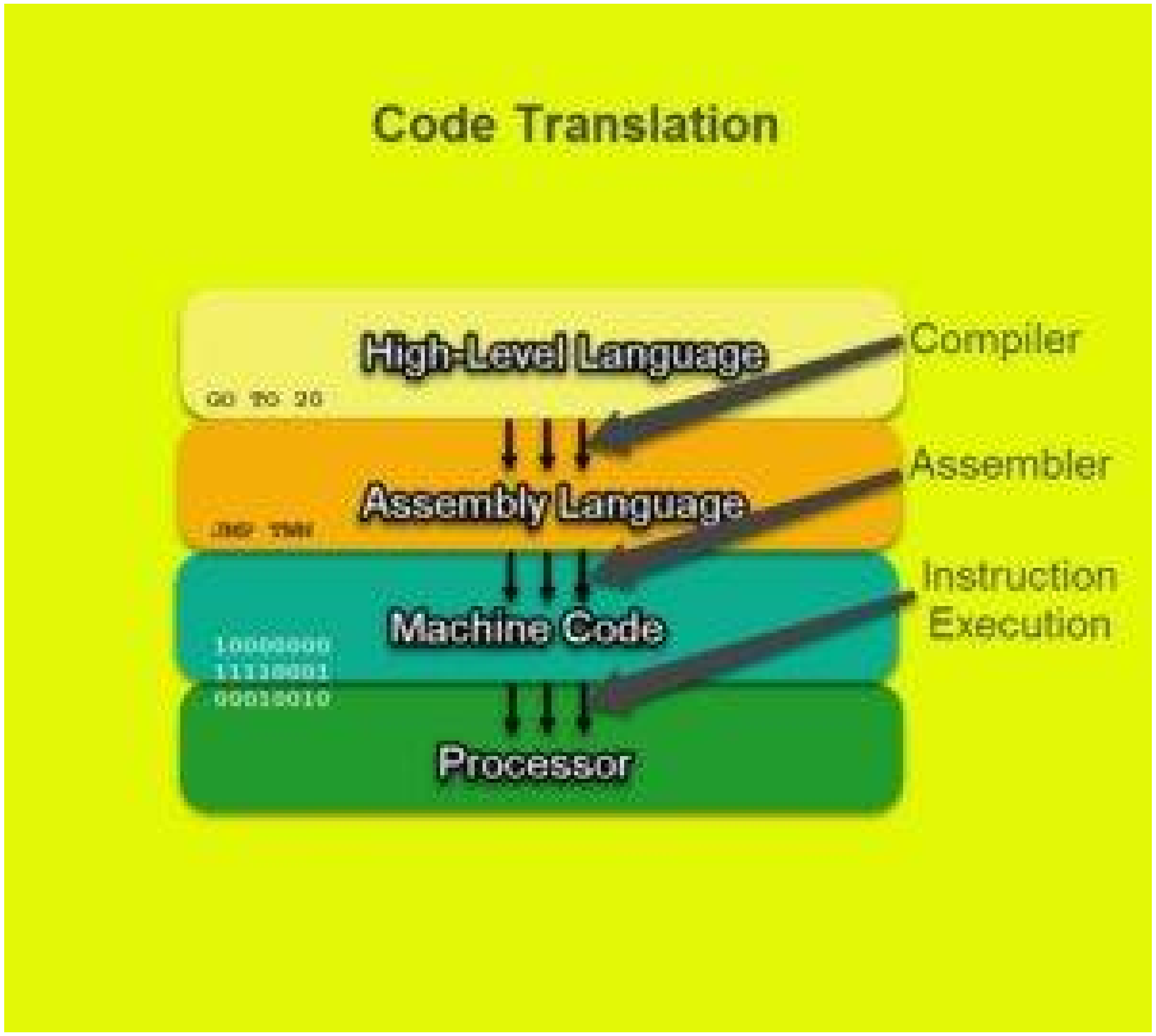

C Program
Compiling
Machine Language Program
Source Code
Compiler
Binary Code
Input
Binary Code
Output
User
Executing
User

Translator (computing)

- A **translator** or **programming language processor** is a computer program that performs the translation of a program written in a given programming language into a **functionally equivalent** program in another computer language (the target language), without losing the functional or logical structure of the original code (the "essence" of each program).

Translators

- Compilers, Interpreters and Assemblers are all software programming tools that convert code into another type of code, but each term has specific meaning.
- All of the above work in some way towards getting a high-level programming language translated into machine code that the central processing unit (CPU) can understand.
- It’s important to note that all translators, compilers, interpreters and assemblers are programs themselves.

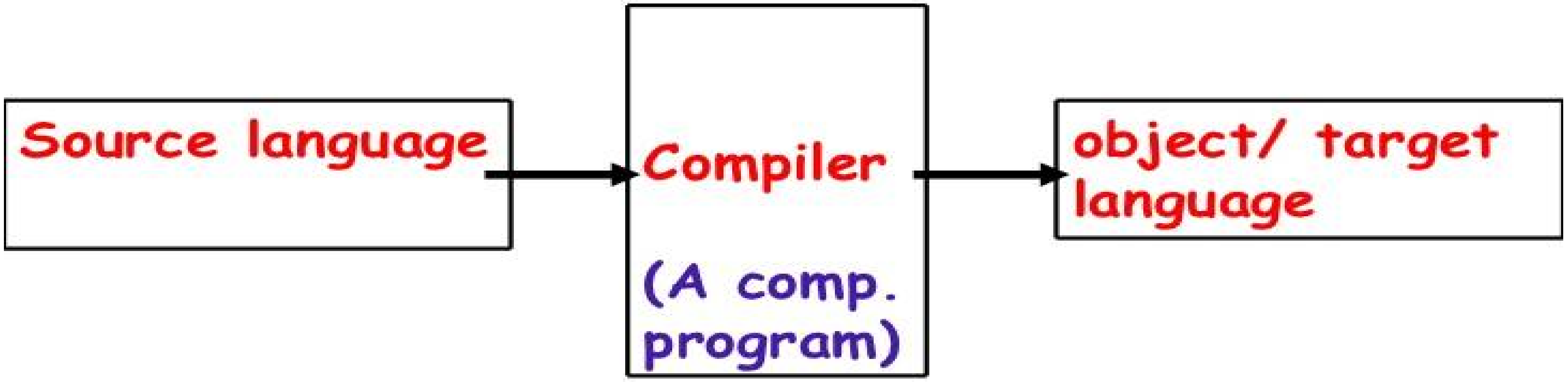

Code Translation
High-Level Language
GO TO 20
Compiler
Assembly Language
JMP TSN
Assembler
Machine Code
10000000
11110001
00010010
Instruction
Execution
Processor

Compilers

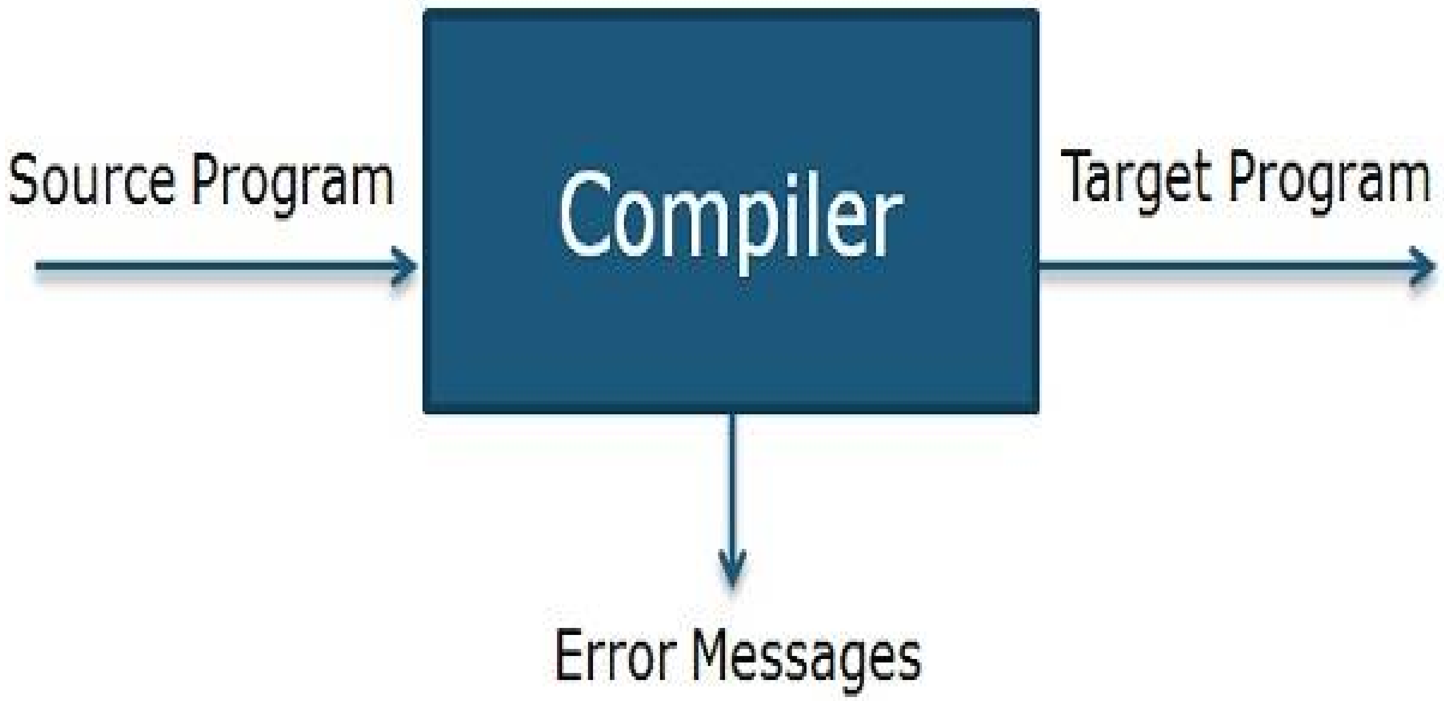
- Compilers convert high-level language code to machine (object) code in **one session**.
- Compilers will **report errors** after compiling has finished.
- Compilers can take a while, because they have to translate high-level code to lower-level machine language all at once and then save the executable object code to memory.
- A compiler creates machine code that runs on a processor with a specific Instruction Set Architecture (ISA), which is processor-dependent.
- Compilers are also platform-dependent.
- A cross-compiler, however, can generate code for a platform other than the one it runs on itself.
- Choosing a compiler , means that first you need to know the ISA, operating system, and the programming language that you plan to use.

What actually a compiler is?

- A compiler is a computer program that translates a computer program written in one computer language (called the *source language*) into an equivalent program written in another computer language (called the output, object, or *target language*).



Translation from source to object


Source Program
Compiler
Target Program
Error Messages

Interpreters

- Another way to get code to run on your processor is to use an interpreter, which is not the same as a compiler.
- An interpreter translates code like a compiler but reads the code and immediately executes on that code, and therefore is initially faster than a compiler.
- Thus, interpreters are often **used** in software development tools **as debugging tools**, as they can execute a single in of code at a time.

Compiler v/s Interpreter

- Compilers translate code all at once and the processor then executes upon the machine language that the compiler produced. If changes are made to the code after compilation, the changed code will need to be compiled and added to the compiled code (or perhaps the entire program will need to be re-compiled.)
- But an interpreter, although skipping the step of compilation of the entire program to start, is much slower to execute than the same program that's been completely compiled.
- Interpreters, however, have usefulness in areas where speed doesn't matter (e.g., debugging and training)

Compiler v/s Interpreter

- Compilers often come as a package with other tools, and each processor manufacturer will have at least one compiler or a package of software development tools (that includes a compiler).
- There are several types of interpreters: the syntax-directed interpreter (i.e., the Abstract Syntax Tree (AST) interpreter), bytecode interpreter, and threaded interpreter Just-in-Time (a kind of hybrid interpreter/compiler), and a few others.
- Some examples of programming languages that use interpreters are Python, Ruby, Perl, and PHP.

Compiler v/s Interpreter

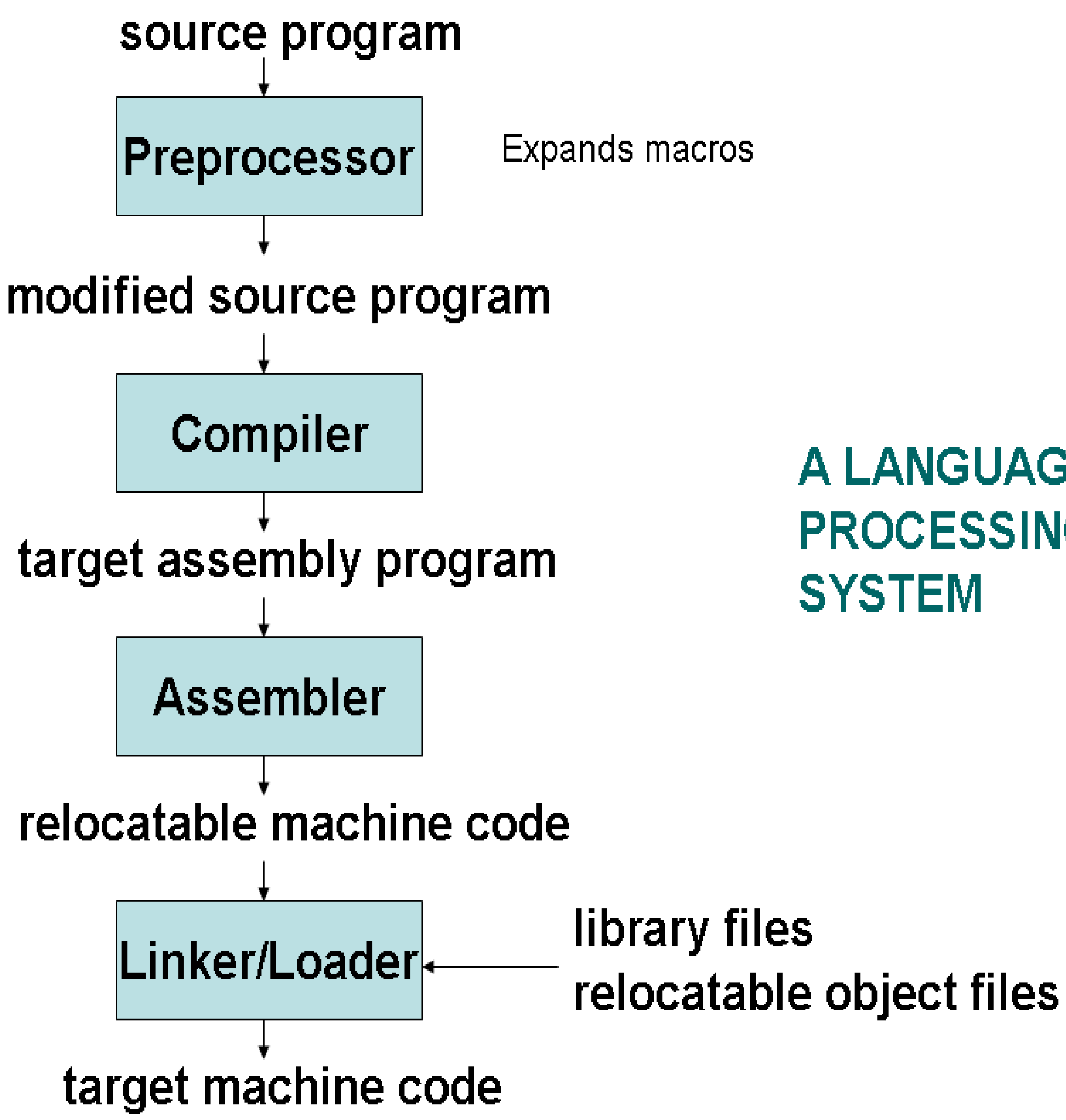
- An interpreter, like a compiler, translates high-level language into low-level machine language. The **difference lies in the way they read the source code** or input.
- A compiler reads the whole source code at once, creates tokens, checks semantics, generates intermediate code, executes the whole 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.

Cousins of Compilers

- Preprocessors
- Assemblers
  - Compiler may produce assembly code instead of generating relocatable machine code directly.
- Loaders and Linkers
  - Loader copies code and data into memory, allocates storage, setting protection bits, mapping virtual addresses, .. Etc
  - Linker handles relocation and resolves symbol references.
- Debugger

Assemblers

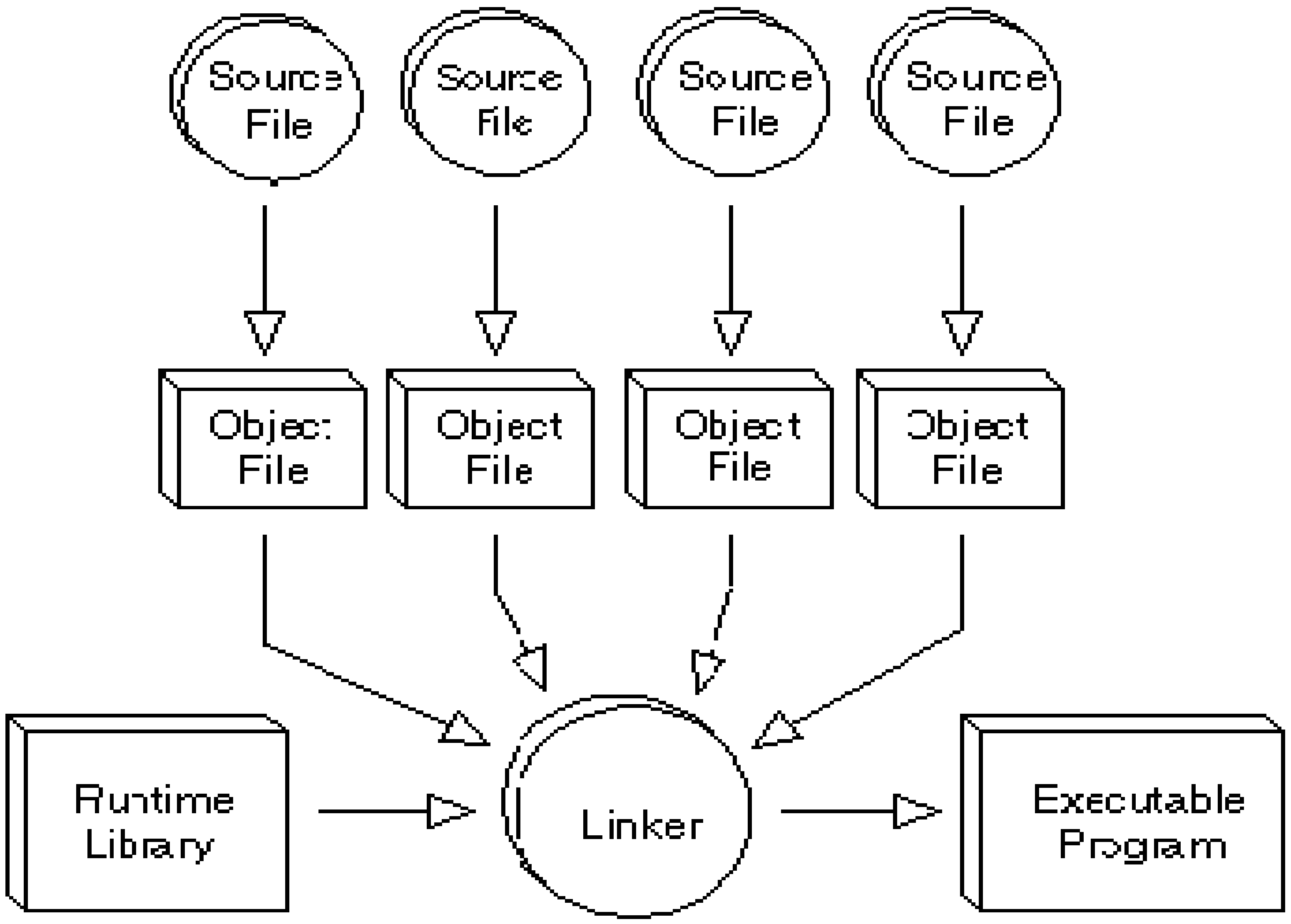
- An assembler translates a program written in assembly language into machine language and is effectively a compiler for the assembly language.
- Assembly language is a low-level programming language. An assembler converts assembly language code into machine code (also known as object code), an even lower-level language that the processor can directly understand.
- Assembly language is the next level up from machine code, and is quite useful in extreme cases of debugging code to determine exactly what’s going on in a problematic execution. For instance, Sometimes compilers will “optimize” code in unforeseen ways that affect outcomes such that it’s necessary to carefully follow the step-by-step action of the processor in assembly code,

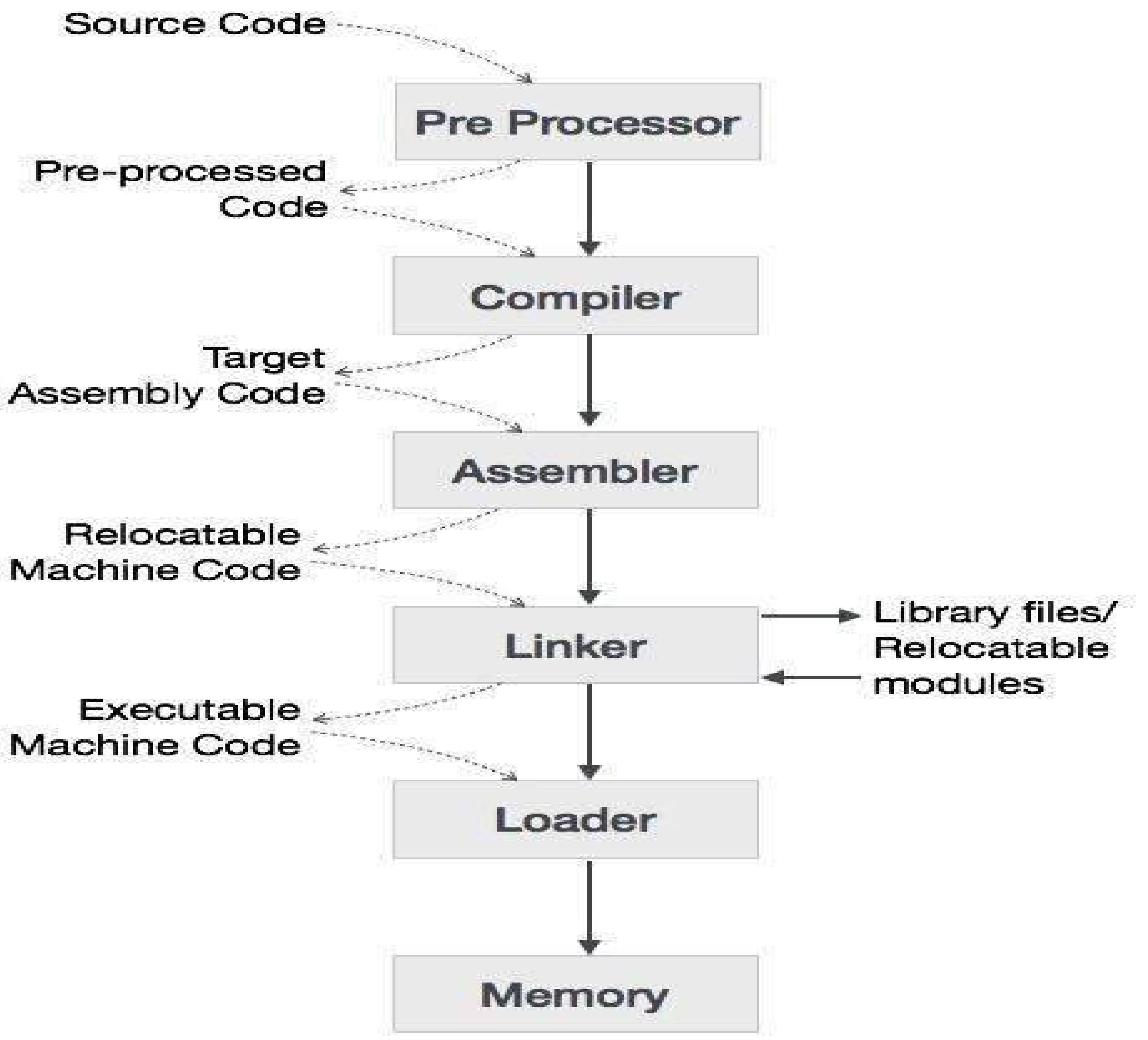


A LANGUAGE PROCESSING SYSTEM

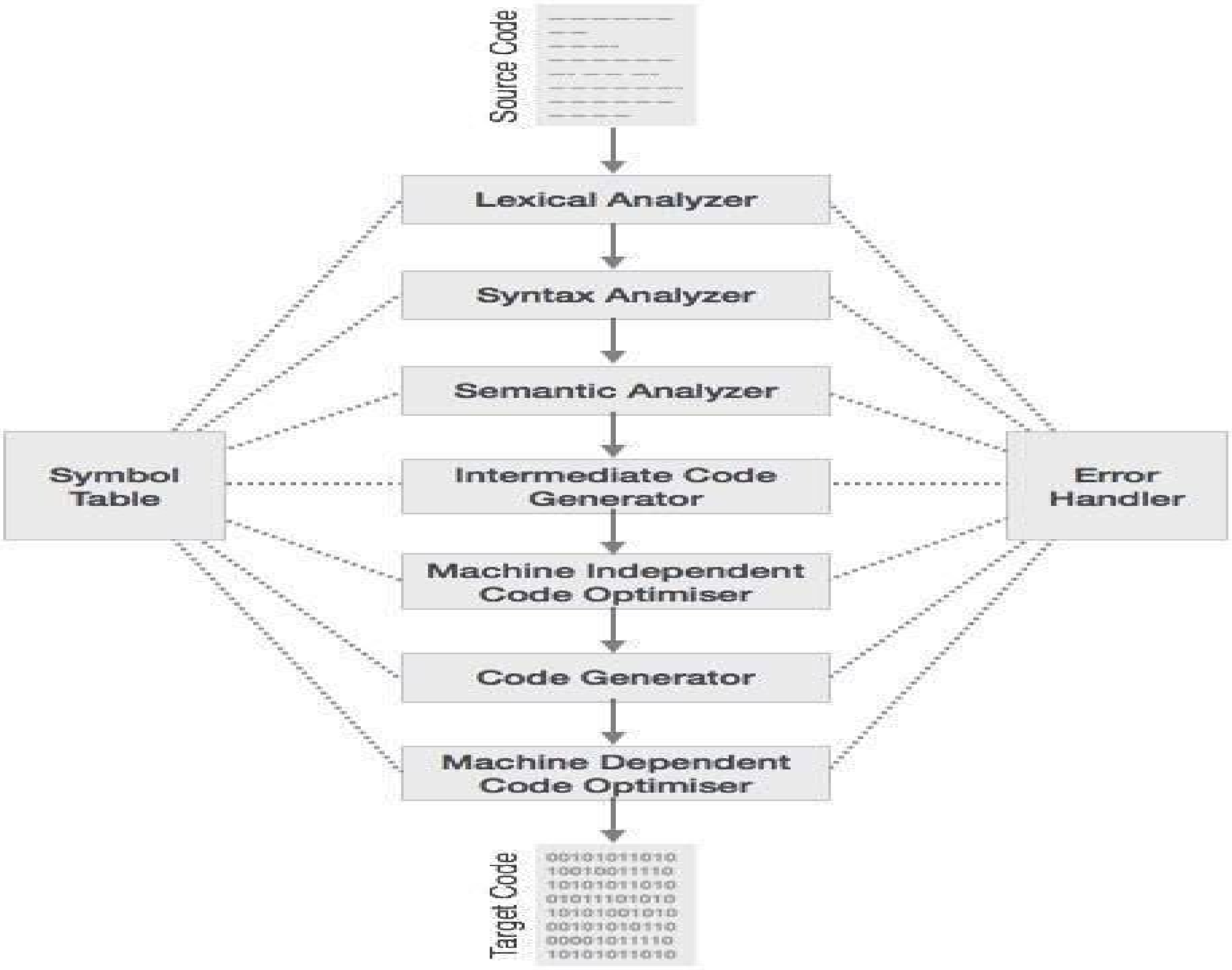
- A **preprocessor**, generally considered as a part of compiler, is a tool that produces input for compilers. It deals with macro-processing(macro expansion), file inclusion, etc.
- A **compiler** is a program that converts high-level language to assembly language.
- An **assembler** translates assembly 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(relocatable).

- **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** 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(absolute) for it. It initializes various registers to initiate execution.


Source File
Source File
Source File
Source File
Object File
Object File
Object File
Object File
Runtime Library
Linker
Executable Program


Source Code
Pre Processor
Pre-processed Code
Compiler
Target Assembly Code
Assembler
Relocatable Machine Code
Linker
Library files/ Relocatable modules
Executable Machine Code
Loader
Memory

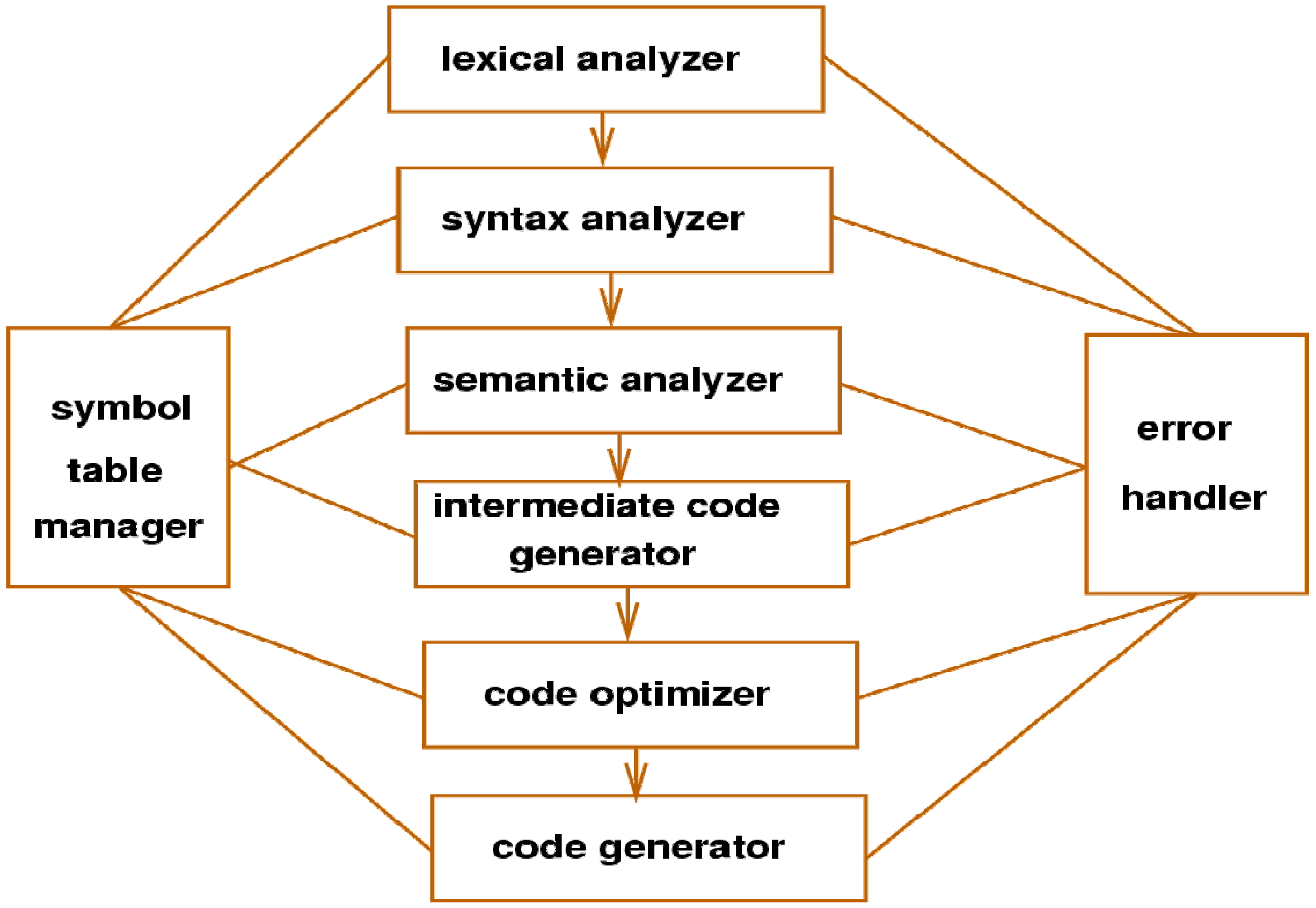
A detailed look at the internals of a compiler


Source Code
Lexical Analyzer
Syntax Analyzer
Semantic Analyzer
Intermediate Code Generator
Machine Independent Code Optimiser
Code Generator
Machine Dependent Code Optimiser
Target Code
Symbol Table
Error Handler

Phases of Compiler

- 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.

Phases of Compiler



Lexical Analysis (Scanner)

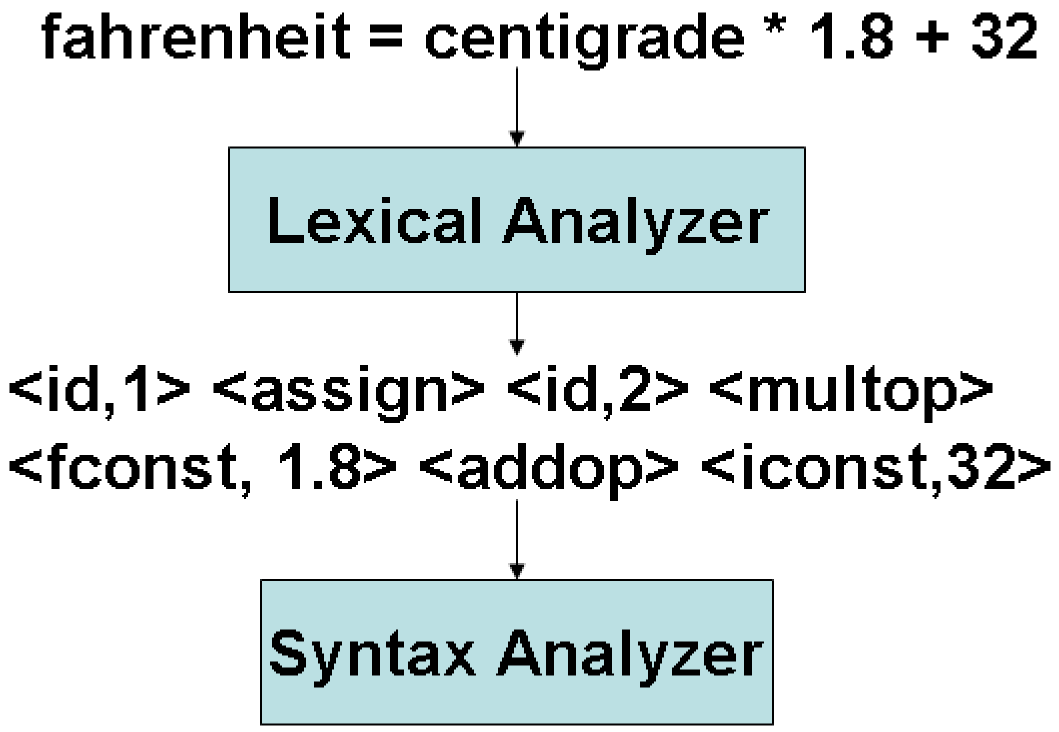
- The first phase of scanner works as a text scanner. This phase scans the source code as a stream of characters and converts it into meaningful lexemes. Lexical analyzer represents these lexemes in the form of tokens as:

  <token-name, attribute-value>

Lexical Analysis

- LA can be generated automatically from regular expression specifications
  - LEX and Flex are two such tools
- LA is a deterministic finite state automaton
- Why is LA separate from parsing?
  - Simplification of design - software engineering reason
  - I/O issues are limited LA alone
  - LA based on finite automata are more efficient to implement than pushdown automata used for parsing (due to stack)

Translation Overview - Lexical Analysis


fahrenheit = centigrade * 1.8 + 32
Lexical Analyzer
<id,1> <assign> <id,2> <multop>
<fconst, 1.8> <addop> <iconst,32>
Syntax Analyzer

Why LA is called Scanner?

- It is also called as scanner because in this phase the complete source code is scanned and a source program is broken into groups of (strings) sequence of characters. These groups are called Tokens.
- It reads the source program one character at a time.
- Each token represents a sequence of characters that can be treated as a single logical entity called **Lexemes.**
- **e.g. const pi = 3.14** (Here pi is a lexeme for token id)
- A token describes a sequence of characters having some meaning in the source program.
- Tokens can be: Identifiers, keywords, operators, punctuation symbols, constants

Tasks performed by LA:

- Eliminates the comments & white spaces
- Correlates error messages with the source program
- Replacing lexemes by tokens

i.e. Generate token stream

Token – example 1

- Input text

  if(x >= y) y = 10;

- Token Stream

  IF LP ID(x) GEQ ID(y) RP

  ID(y) Assign INT(10) SEMI

Syntax Analysis (Hierarchical Analysis)

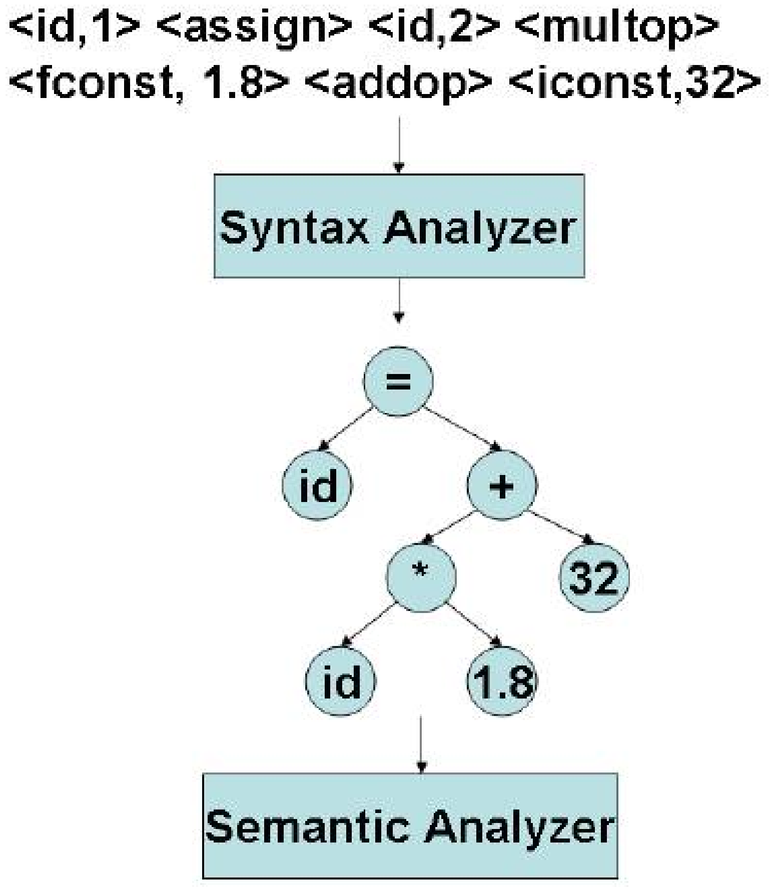
- The next phase is called the syntax analysis or **parsing**. It takes the token produced by lexical analysis as input and generates a parse tree (or syntax tree). In this phase, token arrangements are checked against the source code grammar, i.e. the parser checks if the expression made by the tokens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

The Parser has two functions:

1. It checks that the token appearing in its input occurs in a pattern permitted by the specification of the source language.

2. It make explicitly the hierarchical structure of the incoming token stream.(Parse tree or syntax tree)

Translation Overview - Syntax Analysis


<id,1> <assign> <id,2> <multop>
<fconst, 1.8> <addop> <iconst,32>
Syntax Analyzer
=
id
+
*
32
id
1.8
Semantic Analyzer

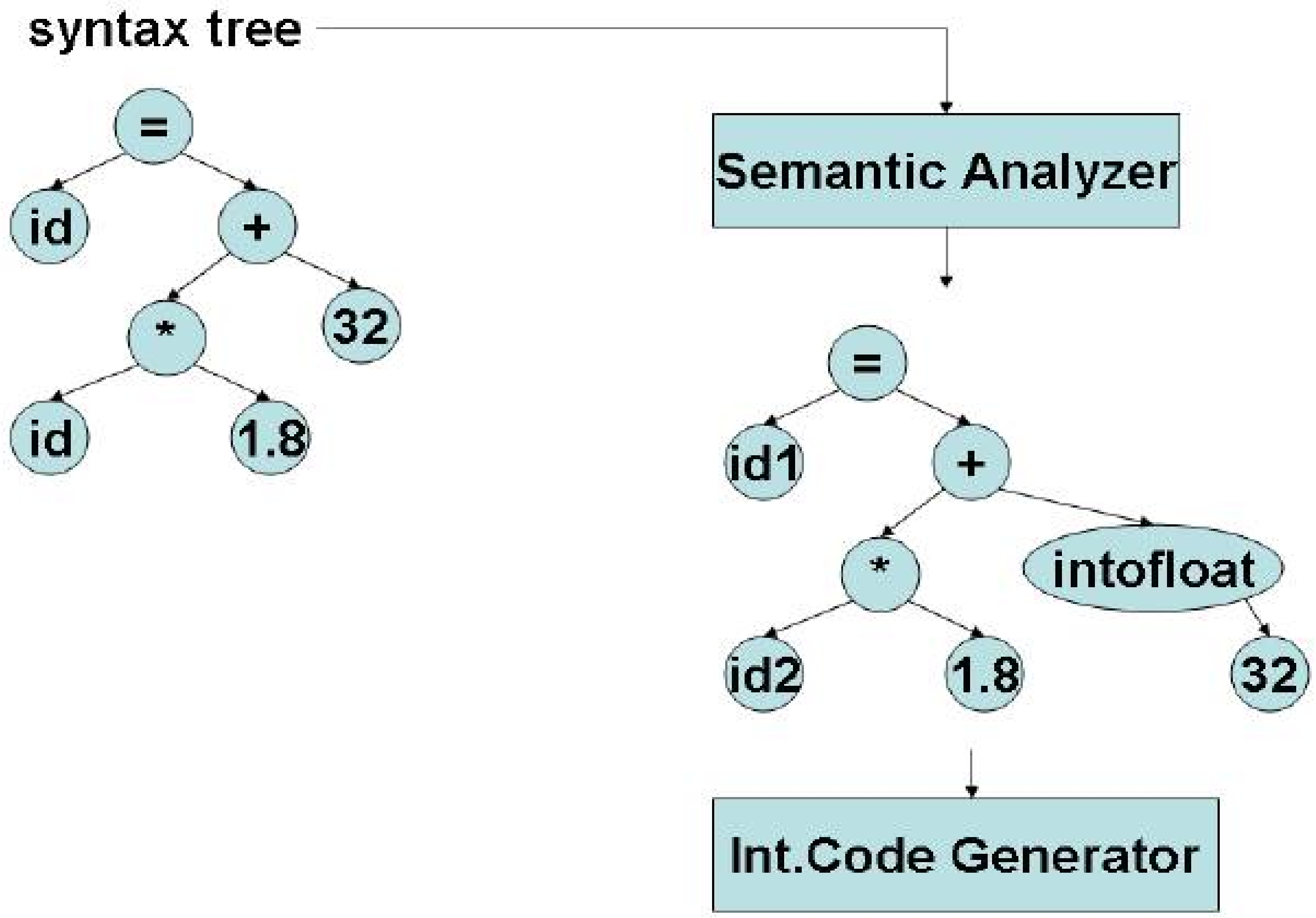
Parsing(Syntax Analysis)

- Syntax analyzers (parsers) can be generated automatically from several variants of context-free grammar specifications
    - LL(1), and LALR(1) are the most popular ones
    - ANTLR (for LL(1)), YACC and Bison (for LALR(1)) are such tools
- Parsers are deterministic push-down automata
- Parsers cannot handle context-sensitive features of programming languages; e.g.,
    - Variables are declared before use
    - Types match on both sides of assignments
    - Parameter types and number match in declaration and use

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.

Translation Overview - Semantic Analysis



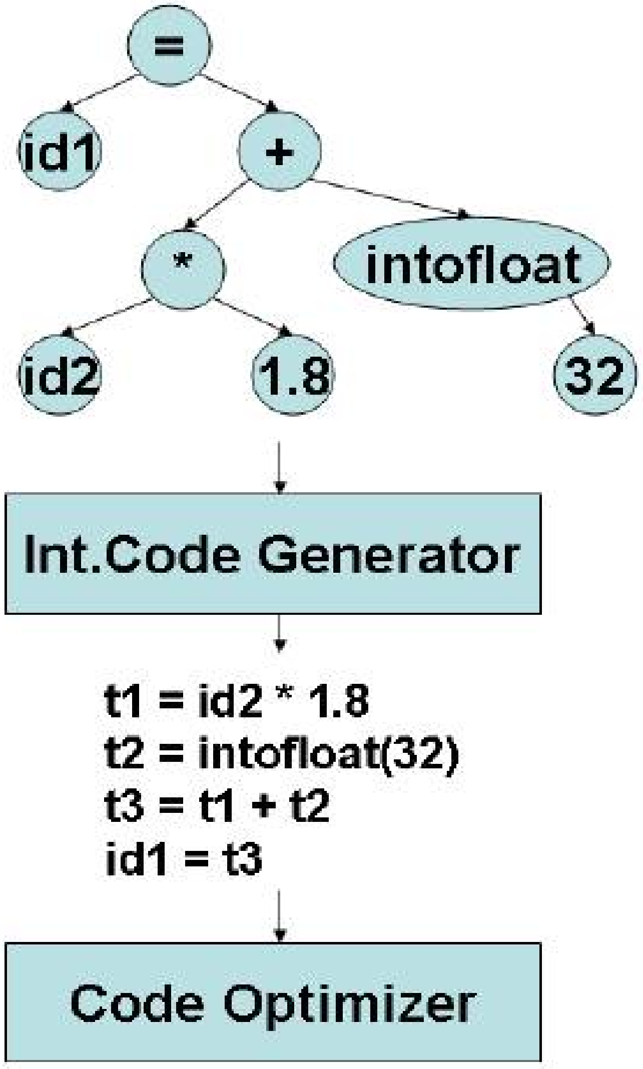
Semantic Analysis

- Semantic consistency that cannot be handled at the parsing stage is handled here
- Type checking of various programming language constructs is one of the most important tasks
- Stores type information in the symbol table or the syntax tree
  - Types of variables, function parameters, array dimensions, etc.
  - Used not only for semantic validation but also for subsequent phases of compilation
- Static semantics of programming languages can be specified using attribute grammars

Intermediate Code Generation

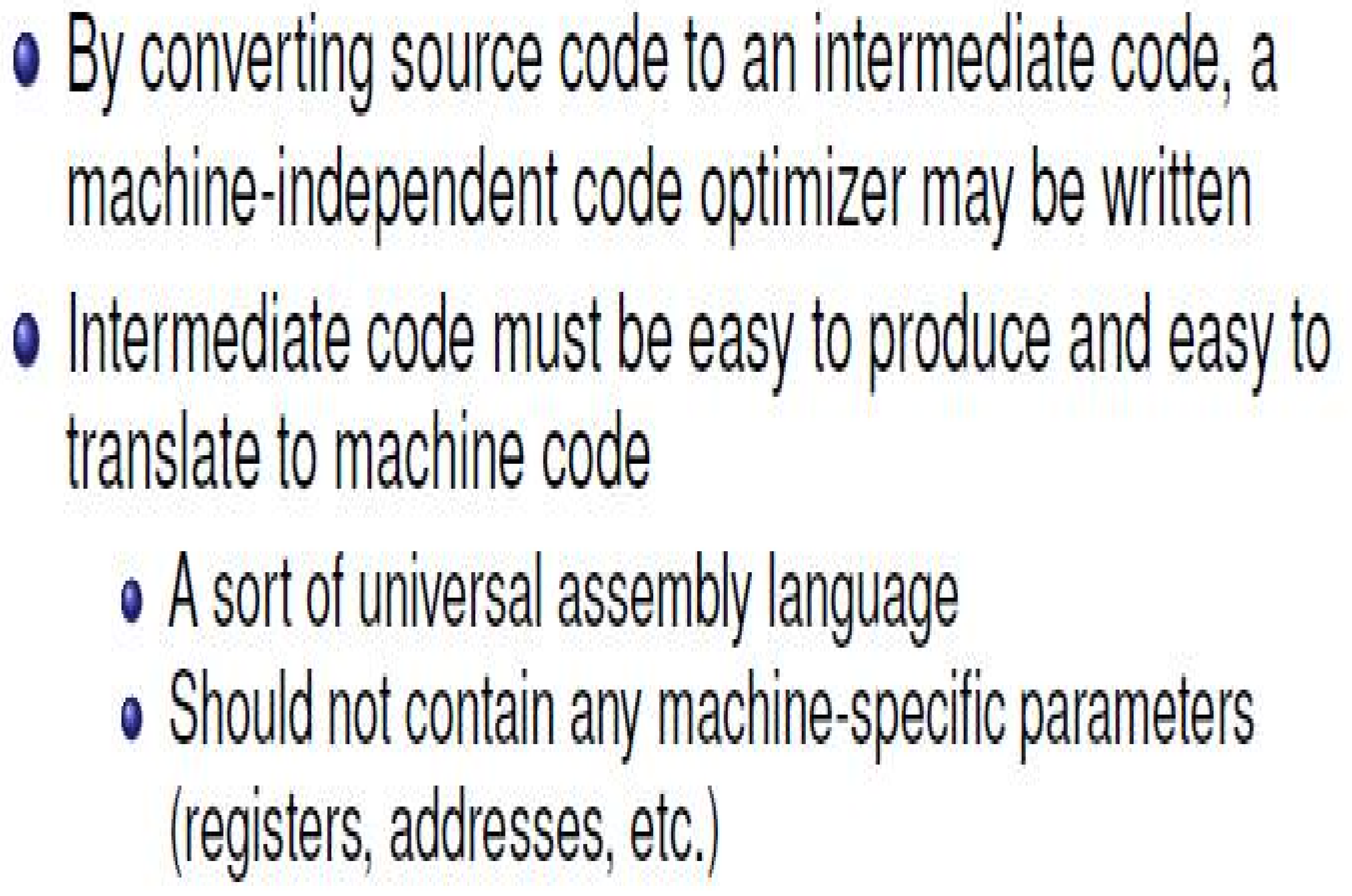
- After semantic analysis the compiler generates an intermediate code of the source code for the target machine.
- It is in between the high-level language and the machine language.
- This intermediate code should be generated in such a way that it makes it easier to be translated into the target machine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

Translation Overview - Intermediate Code Generation



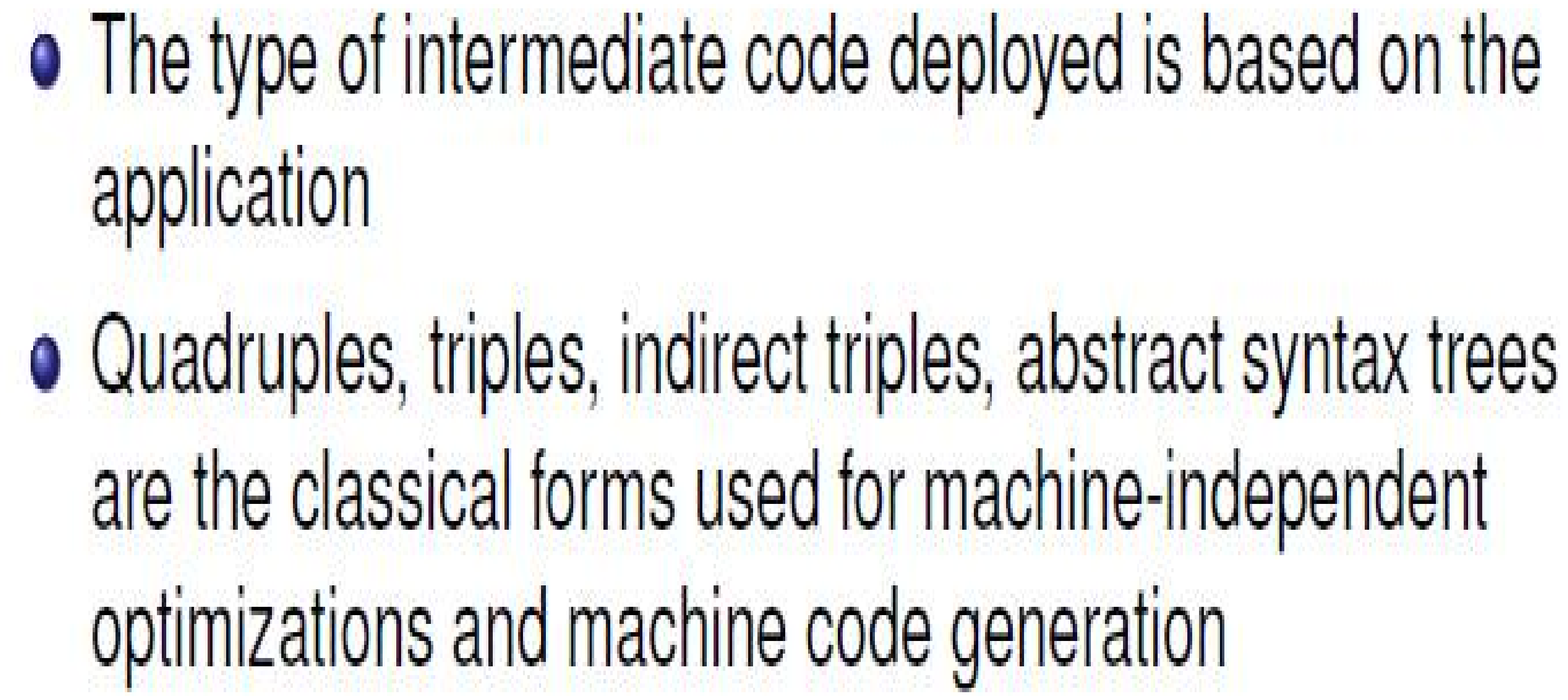
```
t1 = id2 * 1.8
t2 = intofloat(32)
t3 = t1 + t2
id1 = t3
```

Intermediate Code Generation



- By converting source code to an intermediate code, a machine-independent code optimizer may be written
- Intermediate code must be easy to produce and easy to translate to machine code
  - A sort of universal assembly language
  - Should not contain any machine-specific parameters (registers, addresses, etc.)

Different Types of Intermediate Code

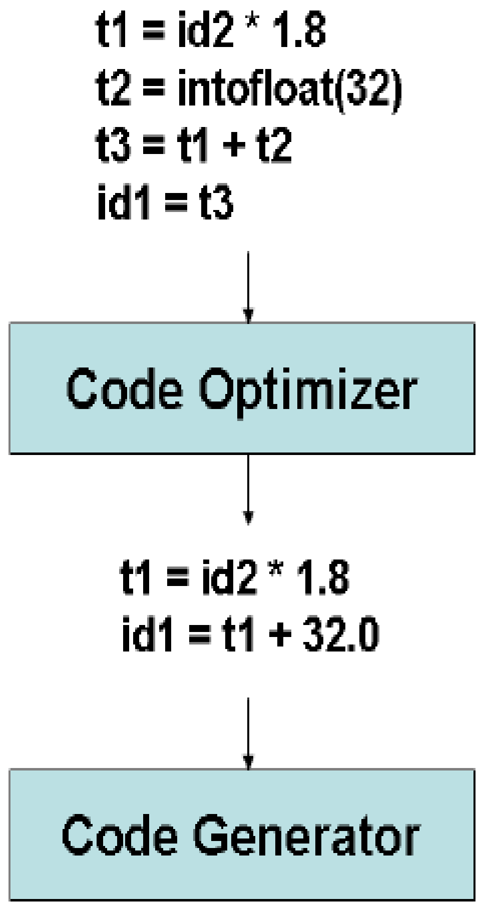


- The type of intermediate code deployed is based on the application
- Quadruples, triples, indirect triples, abstract syntax trees are the classical forms used for machine-independent optimizations and machine code generation

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).

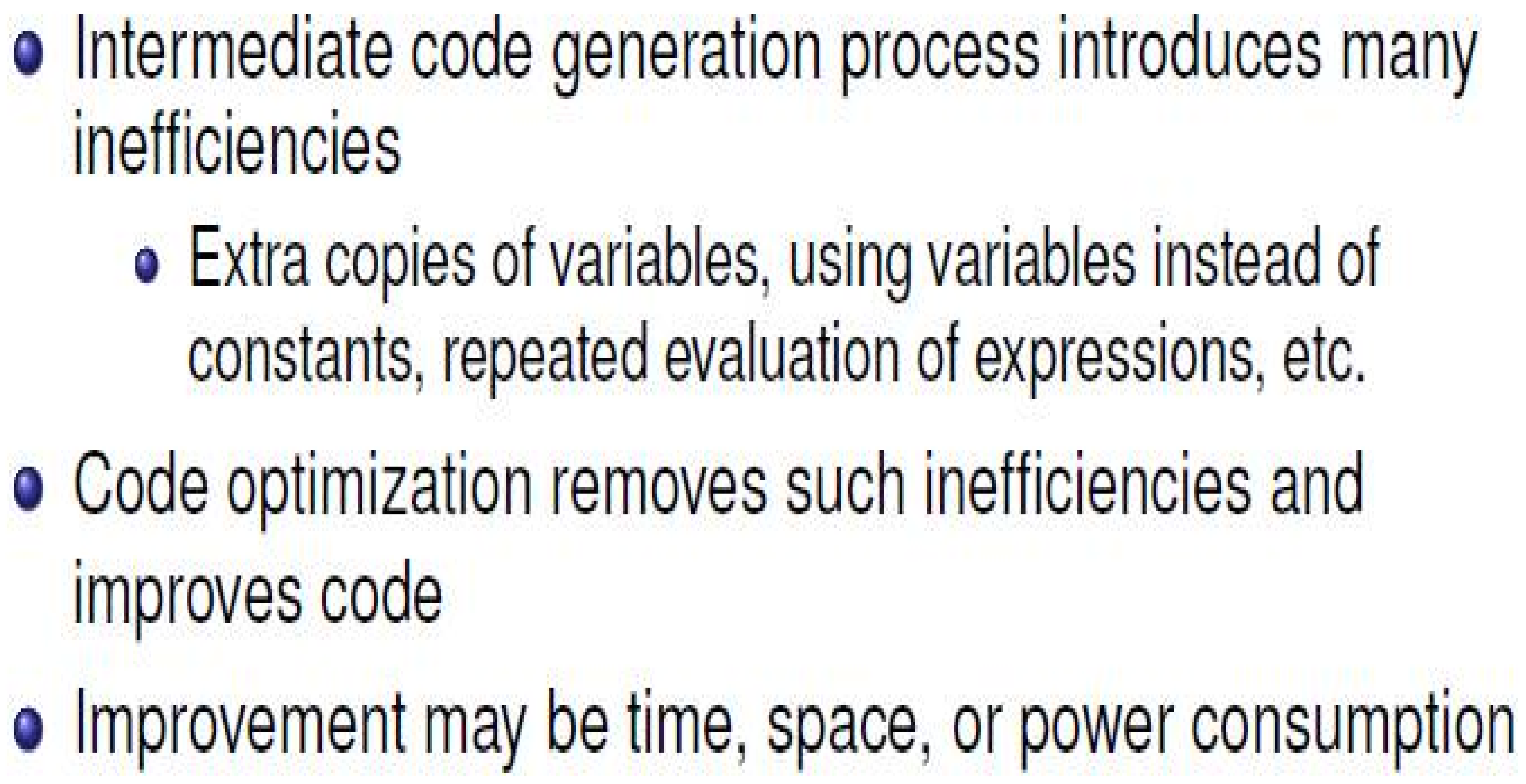
Translation Overview- Code Optimization



```
t1 = id2 * 1.8
t2 = intofloat(32)
t3 = t1 + t2
id1 = t3
```

```
t1 = id2 * 1.8
id1 = t1 + 32.0
```

Machine Independent Code Optimization



- Intermediate code generation process introduces many inefficiencies
  - Extra copies of variables, using variables instead of constants, repeated evaluation of expressions, etc.
- Code optimization removes such inefficiencies and improves code
- Improvement may be time, space, or power consumption

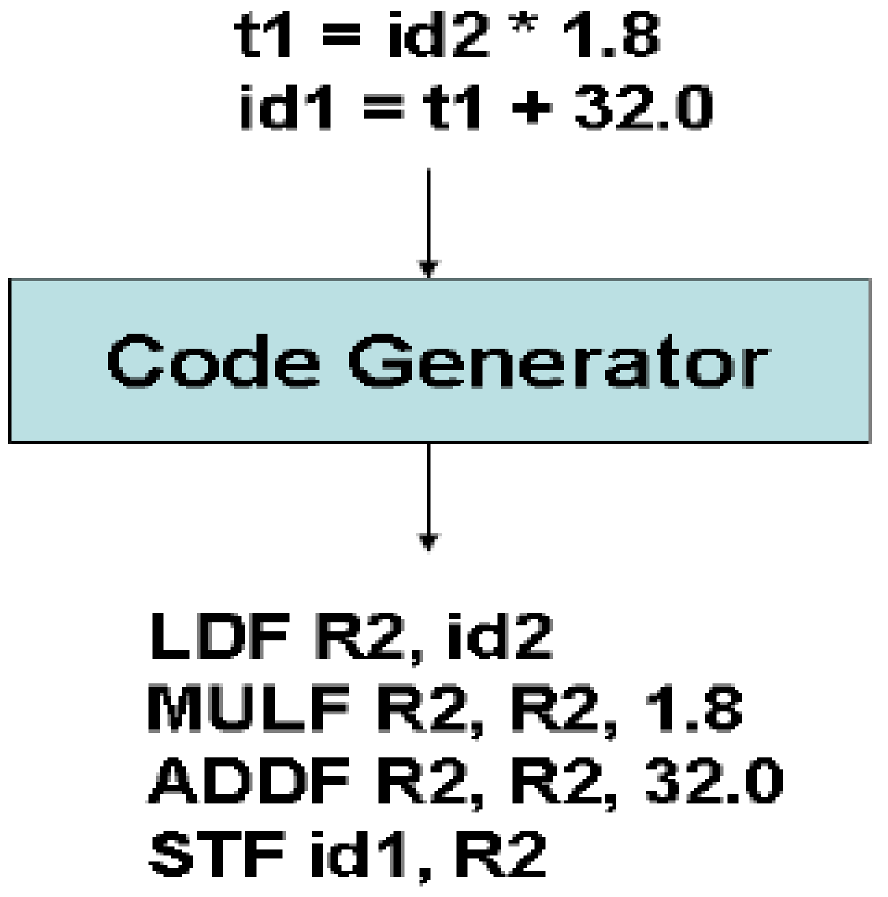
Examples of Machine Independent Optimization

- Common sub-expression elimination
- Copy propagation
- Loop invariant code motion
- Partial redundancy elimination
- Induction variable elimination and strength reduction
- Code opimization needs information about the program
  - which expressions are being recomputed in a function?
  - which definitions reach a point?
- All such information is gathered through data-flow analysis

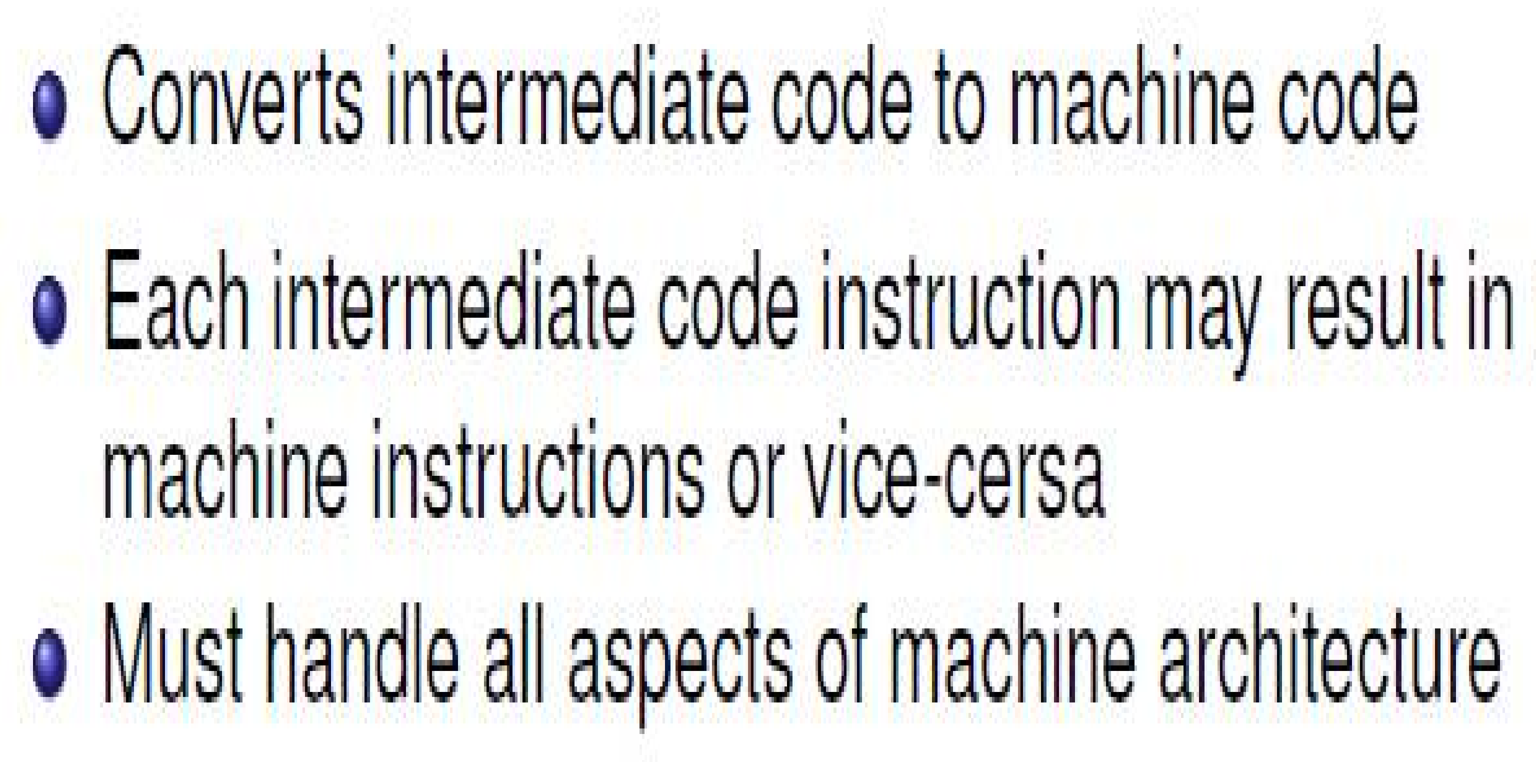
Code Generation

- 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.

Translation Overview- Code Generation



Code Generation (Machine Dependent)



- Converts intermediate code to machine code
- Each intermediate code instruction may result in many machine instructions or vice-cersa
- Must handle all aspects of machine architecture

Symbol Table

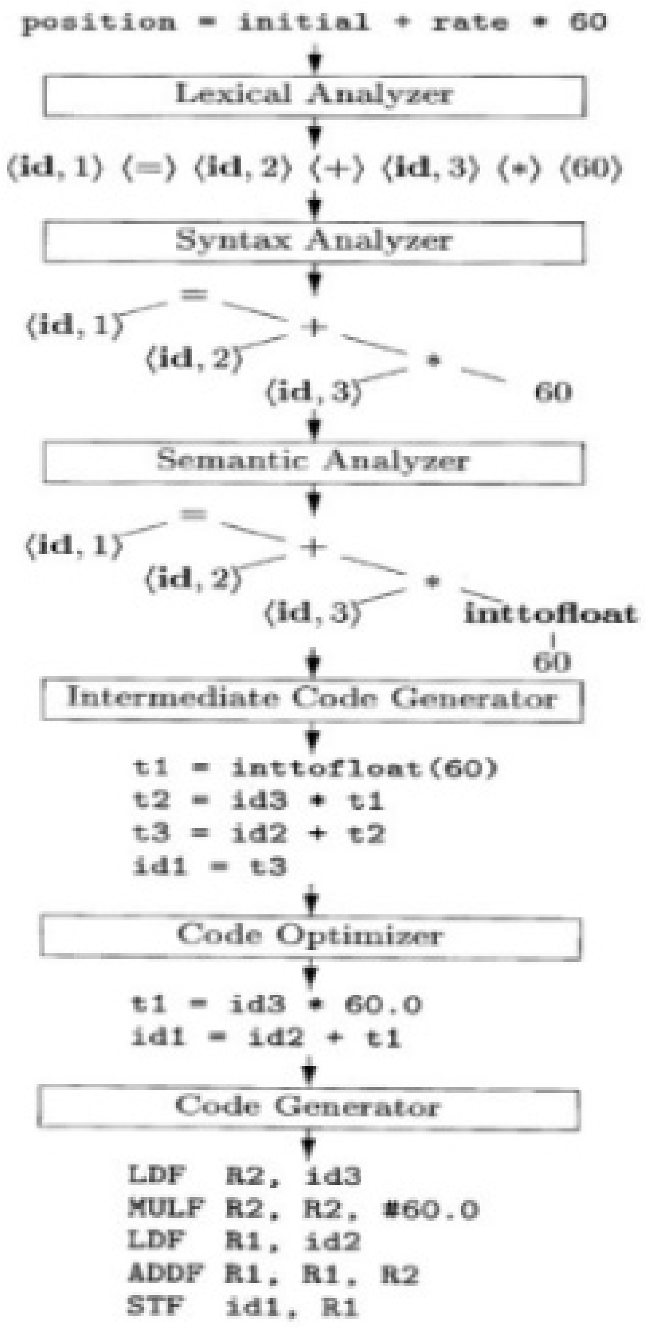
- It is a data-structure maintained throughout all the phases of a compiler. All the identifier's names along with their types are stored here. The symbol table makes it easier for the compiler to quickly search the identifier record and retrieve it. The symbol table is also used for scope management.

Error Handler

- Each phase can encounter error. However after detecting an error, a phase must somehow deal with the error, so that the compilation can proceed, allowing further errors in the source program to be detected.
- The errors are reported in the form of message.

1	position	...
2	initial	...
3	rate	...

SYMBOL TABLE



```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

```
t1 = id3 * 60.0
id1 = id2 + t1
```

```
LDF  R2, id3
MULF R2, R2, #60.0
LDF  R1, id2
ADDF R1, R1, R2
STF  id1, R1
```

Example: position = initial + rate * 60

1.”position” is a lexeme mapped into a token (id, 1), where id is an abstract symbol standing for identifier and 1 points to the symbol table entry for position. The symbol-table entry for an identifier holds information about the identifier, such as its name and type.

2. = is a lexeme that is mapped into the token (=). Since this token needs no attribute-value, we have omitted the second component. For notational convenience, the lexeme itself is used as the name of the abstract symbol.

3. “initial” is a lexeme that is mapped into the token (id, 2), where 2 points to the symbol-table entry for initial.

4. + is a lexeme that is mapped into the token (+).

5. “rate” is a lexeme mapped into the token (id, 3), where **3** points to the symbol-table entry for rate.

6. * is a lexeme that is mapped into the token (*) .

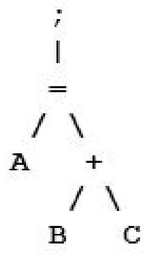
7. 60 is a lexeme that is mapped into the token (60)

Blanks separating the lexemes would be discarded by the lexical analyzer.

Table

token	lexem
id 1	
id 2	
id 3	

The Phases of a Compiler

Phase	**Output**	**Sample**
Programmer (source code producer)	Source string	`A=B+C;`
Scanner (performs *lexical analysis*)	Token string	`'A', '=', 'B', '+', 'C', ';'` And *symbol table* with names
Parser (performs *syntax analysis* based on the grammar of the programming language)	Parse tree or abstract syntax tree	
Semantic analyzer (type checking, etc)	Annotated parse tree or abstract syntax tree	

Intermediate code generator	Three-address code, quads, or RTL	`int2fp B       t1`
		`+      t1   C  t2`
		`:=     t2      A`

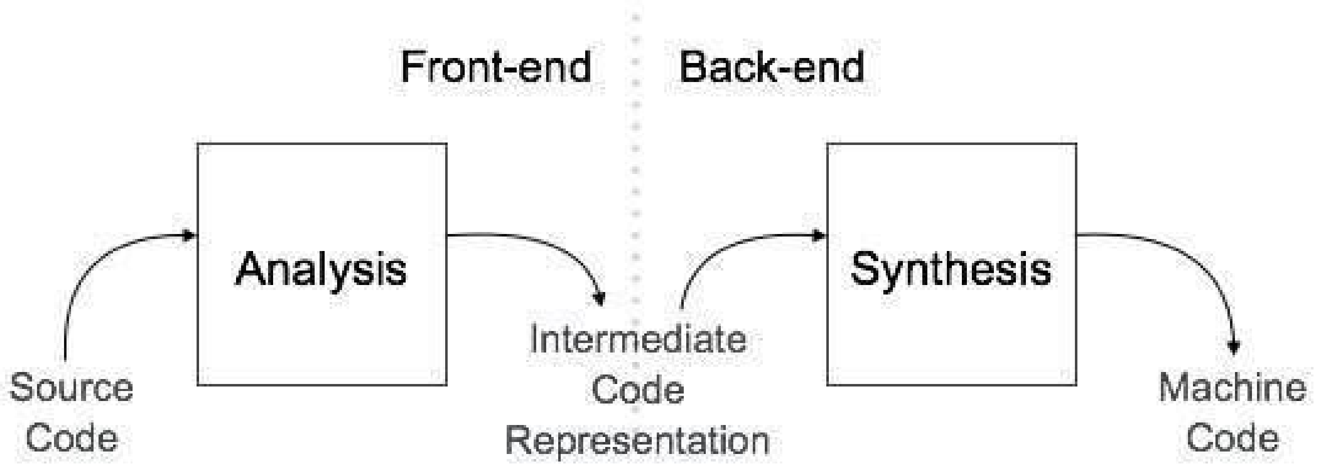
Optimizer	Three-address code, quads, or RTL	`int2fp B       t1`
		`+      t1 #2.3 A`

Code generator	Assembly code	`MOVF  #2.3,r1`
		`ADDF2 r1,r2`
		`MOVF  r2,A`

Peephole optimizer	Assembly code	`ADDF2 #2.3,r2`
		`MOVF  r2,A`

Compiler Architecture

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

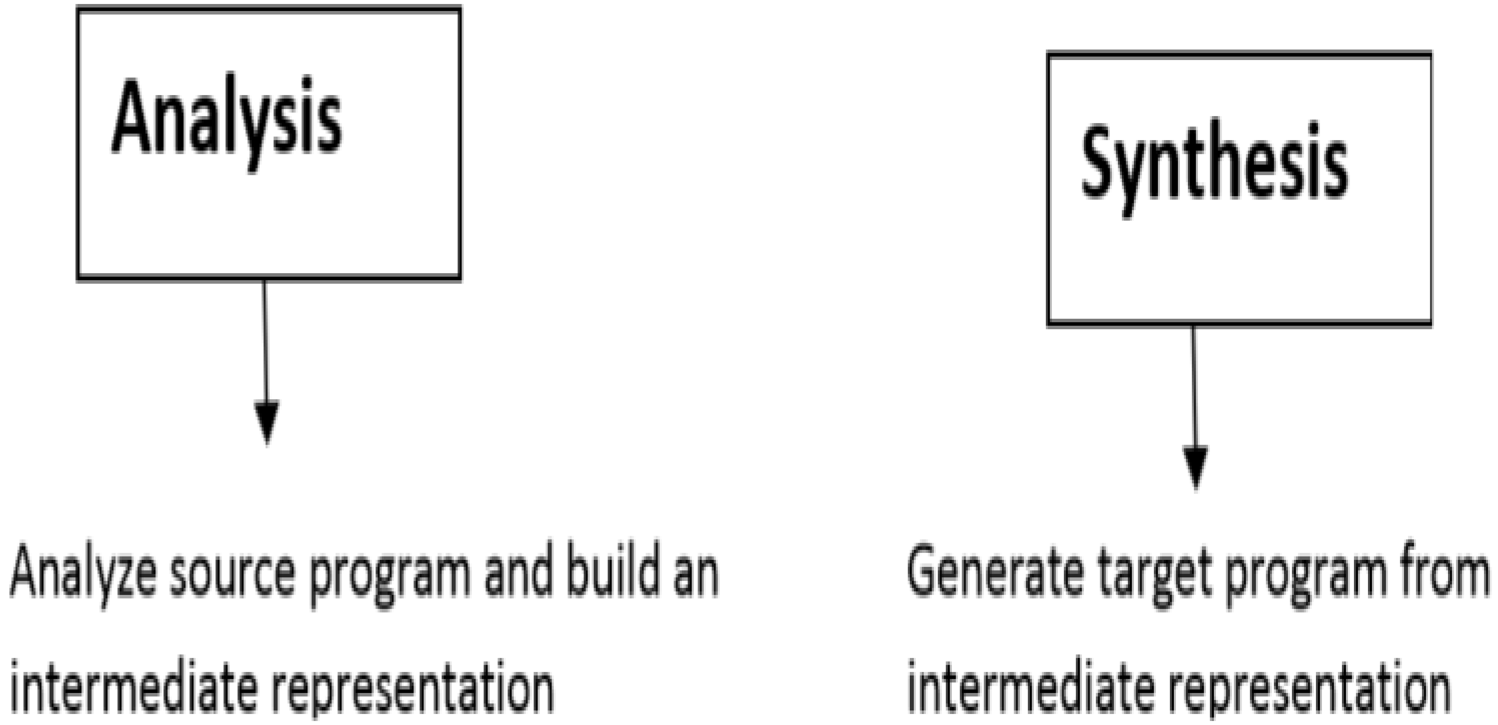


Analysis Phase

- 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.



Analyze source program and build an intermediate representation

Generate target program from intermediate representation

Front End /Back End Model

- **Front end** consists of those phases that depends mainly on source language and are largely independent on the target machine.
- So it normally includes lexical and syntax analysis ,creation of symbol table, semantic analysis and generation of intermediate code.
- A certain amount of code optimization can also be done here.
- It also includes error handling that goes along with each of these phases.

Front End /Back End Model

- The Back end includes those portion of a compiler that depends on the target machine.
- In this we find code optimization and code generation phase along with necessary error handling and symbol table operations.

Advantages: Front End /Back End Model

- 1. By Keeping the same front end & attaching different back ends, one can produce a compiler for same source language on different machines.
- 2. By keeping different front ends and same backend, one can compile several different languages on the same machine.

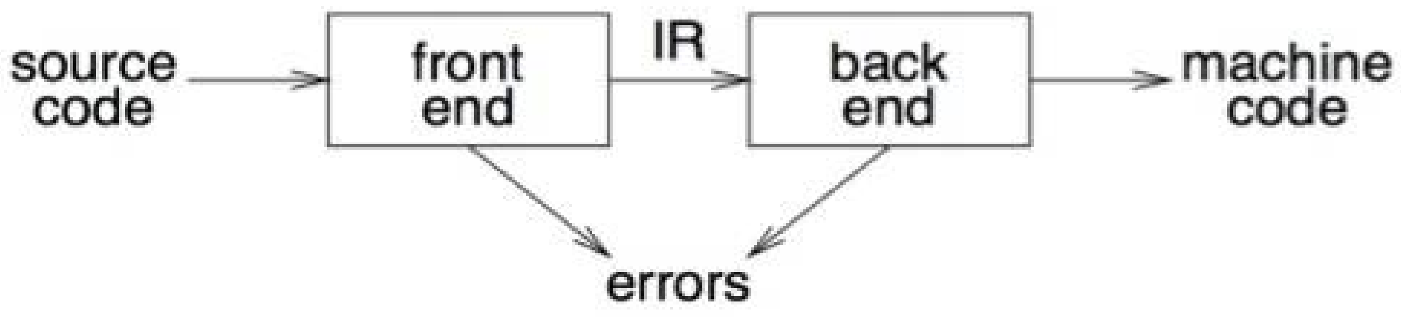
Passes of Compiler

- In an implementation of a compiler, portions of one or more phases are combined into a module called as pass.
- A Pass reads the source program or output of previous pass, makes the transformation and write its output into an intermediate file, which may then be read by a subsequent pass.
- Each pass communicate with other pass via temporary file.

Passes of Compiler

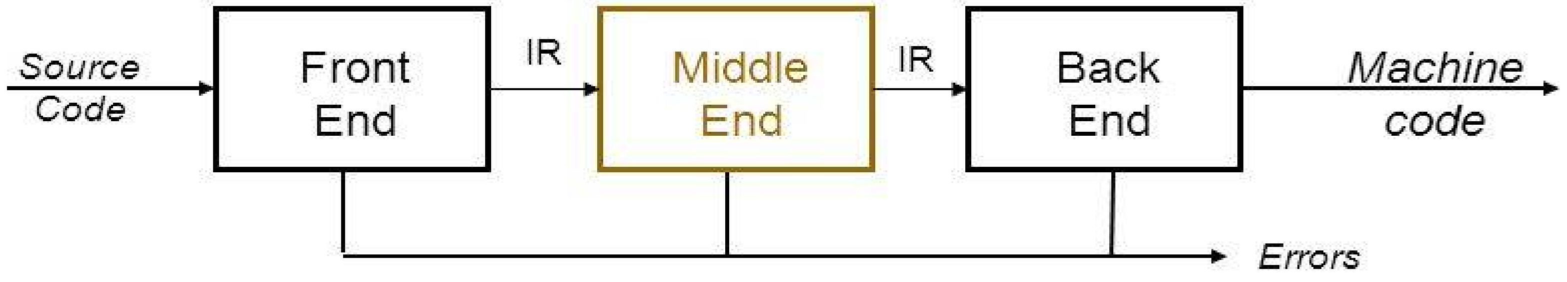
- The structure of the source language and the environment in which compiler operate, has a strong effect on the number of passes.
- Based on which we have single pass compiler and multi pass compiler.

Two pass compiler



- intermediate representation (IR)
- front end maps legal code into IR
- back end maps IR onto target machine
- simplify retargeting
- allows multiple front ends
- multiple passes $\Rightarrow$ better code

Traditional Three-pass Compiler



- Code Improvement (or *Optimization*)
- Analyzes IR and rewrites (or *transforms*) IR
- Primary goal is to reduce running time of the compiled code
    - May also improve space, power consumption, …
- Must preserve "meaning" of the code

Measured by values of named variables

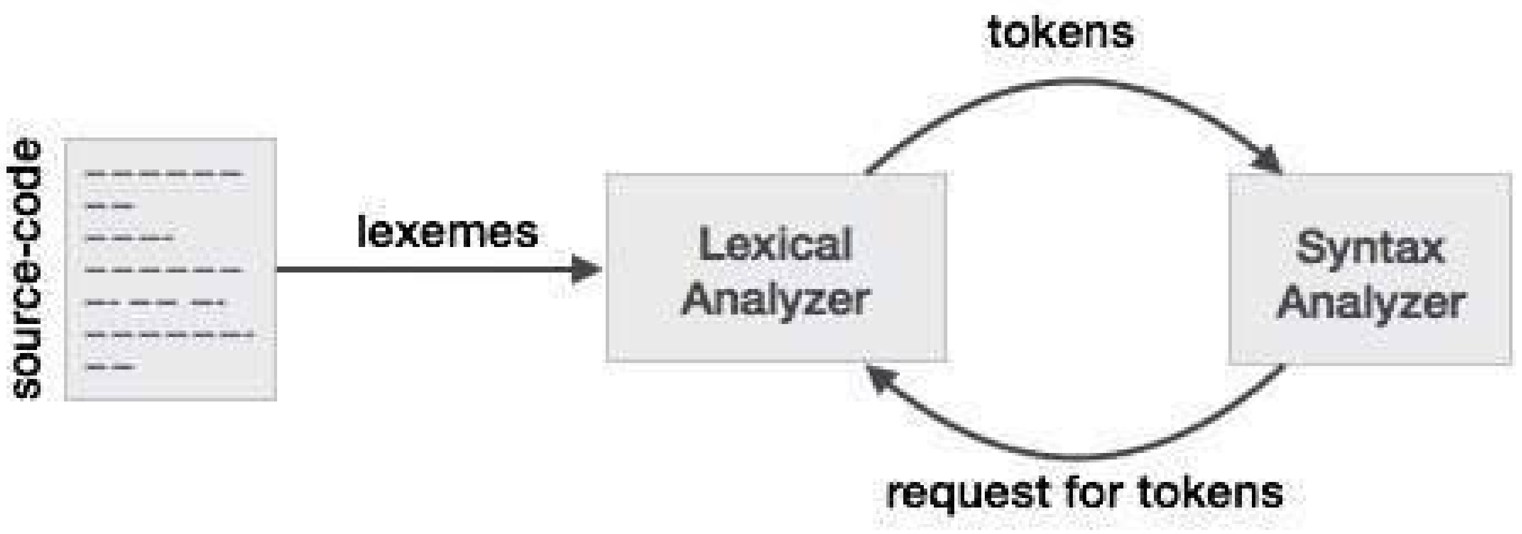
Advantage of Single-pass compilers

- One-pass compilers are smaller and faster than multi-pass compilers.
- This is in contrast to a multi-pass compiler which converts the program into one or more intermediate representations in steps between source code and machine code, and which reprocesses the entire compilation unit in each sequential pass.

Advantage of Multi-pass compilers

- One-pass compilers are unable to generate as efficient programs as multi-pass compilers due to the limited scope of available information.
- **Machine Independent**: Since the multiple passes include a modular structure, and the code generation decoupled from the other steps of the compiler, the passes can be reused for different hardware/machines.

LEXICAL ANALYSIS
(Linear Analysis)



1.Lexical analysis is the first phase of a compiler.
2.The lexical analyzer works closely with the syntax analyzer. It reads character streams from the source code, checks for legal tokens, and passes the data to the syntax analyzer when it demands.
3.If the lexical analyzer finds a token invalid, it generates an error.

- **Language:** Computer languages are considered as finite sets, and mathematically set operations can be performed on them. Finite languages can be described by means of regular expressions.
- **Regular Expressions :** Regular expressions have the capability to express finite languages by defining a pattern for finite strings of symbols. The grammar defined by regular expressions is known as **regular grammar**. The language defined by regular grammar is known as **regular language**.

Operations

- Regular expression is an important notation for specifying patterns. There are a number of algebraic laws that are obeyed by regular expressions.
- The various **operations** on languages are:
- **Union** of two languages L and M is written as

  L U M = {s | s is in L **or** s is in M}
- **Concatenation** of two languages L and M is written as

  LM = {st | s is in L **and** t is in M}
- The Kleene **Closure** of a language L is written as

  L* = Zero or more occurrence of language L.

Notations

If r and s are regular expressions denoting the languages L(r) and L(s), then

- **Union** : (r)|(s) is a regular expression denoting L(r) U L(s)
- **Concatenation** : (r)(s) is a regular expression denoting L(r)L(s)
- **Kleene closure** : (r)* is a regular expression denoting (L(r))*
- (r) is a regular expression denoting L(r)

Precedence and Associativity

- *, concatenation (.), and | (pipe sign) are **left associative**
- * has the highest precedence
- Concatenation (.) has the second highest precedence.
- | (pipe sign) has the lowest precedence of all.

If x is a regular expression, then:

- x* means zero or more occurrence of x.
  i.e., it can generate { e, x, xx, xxx, xxxx, ... }

- x+ means one or more occurrence of x.
  i.e., it can generate { x, xx, xxx, xxxx ... } or x.x*

- x? means at most one occurrence of x
  i.e., it can generate either {x} or {e}.

Finite Automata

- Finite automata is a recognizer for regular expressions.
- When a regular expression string is fed into finite automata, it changes its state for each literal.
- If the input string is successfully processed and the automata reaches its final state, it is accepted, i.e., the string just fed was said to be a valid token of the language in hand.

The mathematical model of finite automata consists of:

- Finite set of states (Q)
- Finite set of input symbols (Σ)
- One Start state (q0)
- Set of final states (qf)
- Transition function (δ)

The transition function (δ) maps the finite set of state (Q) to a finite set of input symbols (Σ), $Q \times \Sigma \rightarrow Q$

Longest Match Rule

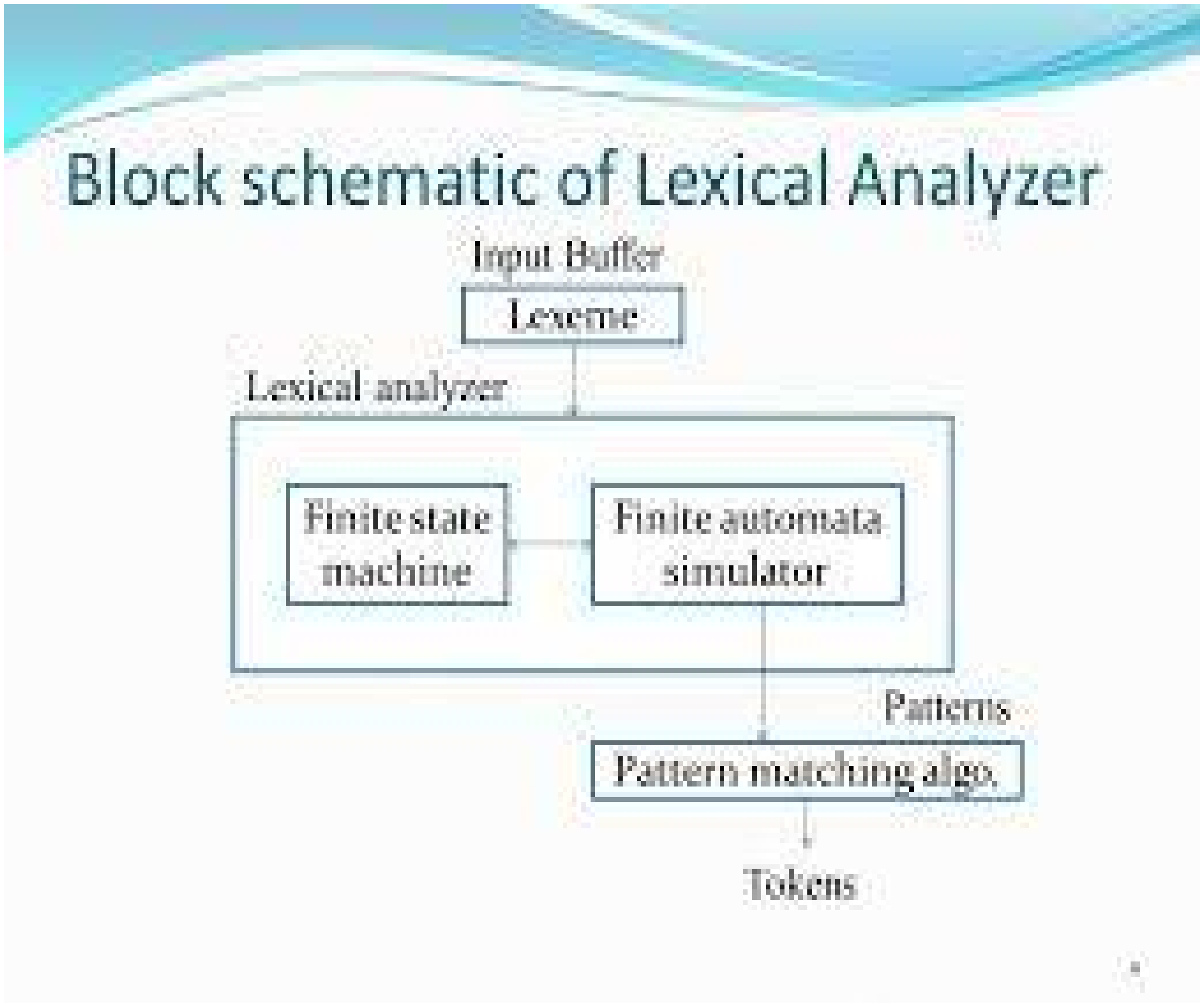
When the lexical analyzer read the source-code, it scans the code letter by letter; and when it encounters a whitespace, operator symbol, or special symbols, it decides that a word is completed.

For example:

int intvalue;

- While scanning both lexemes till ‘int’, the lexical analyzer cannot determine whether it is a keyword *int* or the initials of identifier int value.
- The Longest Match Rule states that the lexeme scanned should be determined based on the longest match among all the tokens available.
- The lexical analyzer also follows **rule priority** where a reserved word, e.g., a keyword, of a language is given priority over user input.
- That is, if the lexical analyzer finds a lexeme that matches with any existing reserved word, it should generate an error.

Lexical Analysis



- Tokens are specified using Regular Expressions and are recognized by Finite Automata.
- R.E. is a notation used to describe the Tokens.
- F.A. is a mechanism used to recognize these tokens in the input stream.
- Similarly CFG is particularly useful in specifying the syntactic structure of a language.

RegExpr → NFA → DFA

Topics

- Thompson Construction
- Subset construction

Thompson's construction

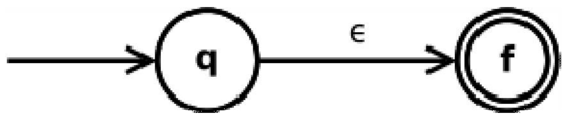
- Thompson's construction is an algorithm for transforming a regular expression into an equivalent nondeterministic finite automaton (NFA). This NFA can be used to match strings against the regular expression.
- Hence, this algorithm is of practical interest, since it can compile regular expressions into NFAs.

The algorithm

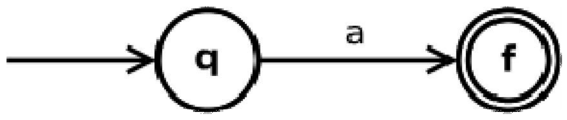
The algorithm works recursively by splitting an expression into its constituent subexpressions, from which the NFA will be constructed using a set of rules.

Rules

- The **empty-expression** ε is converted to



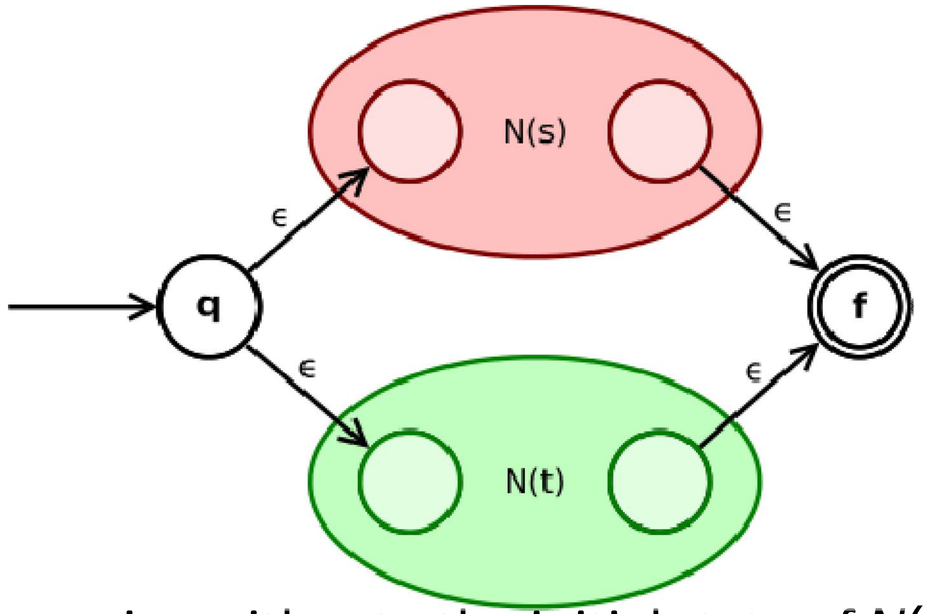
A **symbol** *a* of the input alphabet is converted to



Rules

N(*s*) and *N*(*t*) are the NFA of the sub expressions *s* and *t*, respectively.

- The **union expression** *s*|*t* is converted to

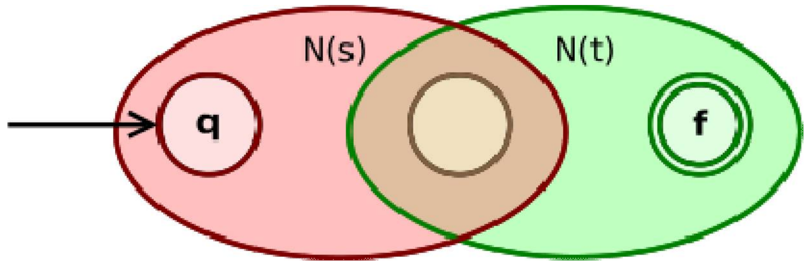


- State *q* goes via ε either to the initial state of *N*(*s*) or *N*(*t*). Their final states become intermediate states of the whole NFA and merge via two ε-transitions into the final state of the NFA.

Rules

N(*s*) and *N*(*t*) are the NFA of the sub expressions *s* and *t*, respectively.

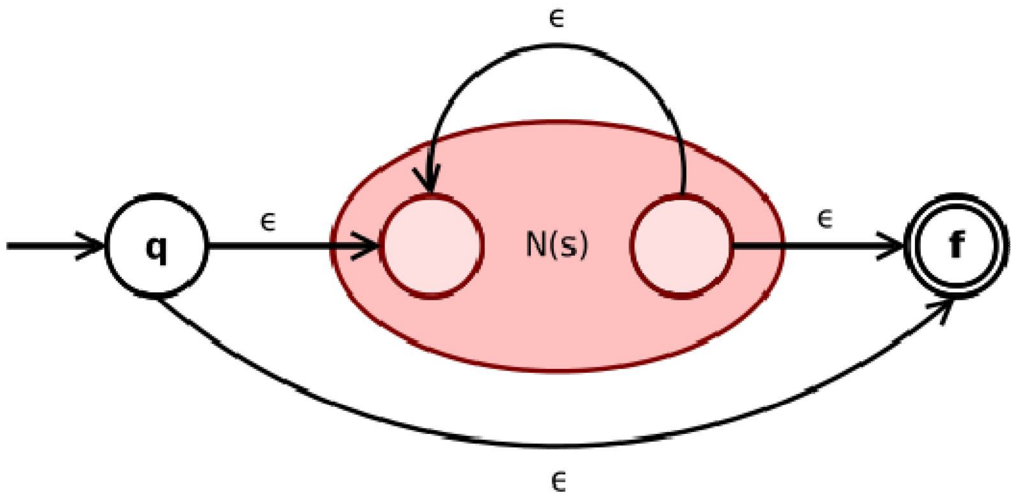
- The **concatenation expression** *st* is converted to



- The initial state of *N*(*s*) is the initial state of the whole NFA.
- The final state of *N*(*s*) becomes the initial state of *N*(*t*).
- The final state of *N*(*t*) is the final state of the whole NFA.

Rules

- The Kleene star expression *s** is converted to

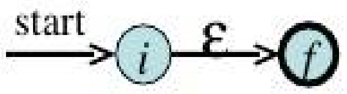


- An ε-transition connects initial and final state of the NFA with the sub-NFA *N*(*s*) in between. Another ε-transition from the inner final to the inner initial state of *N*(*s*) allows for repetition of expression *s* according to the star operator.

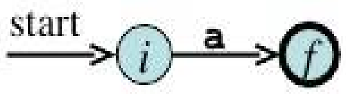
- The **parenthesized expression** (*s*) is converted to *N*(*s*) itself.

From Regular Expression to NFA (Thompson's Construction)

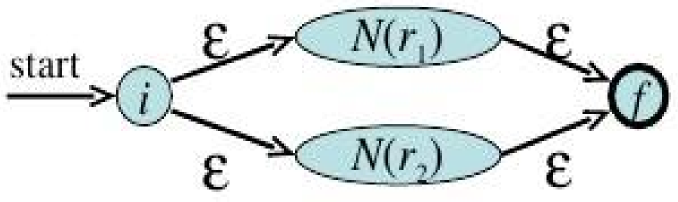
ε



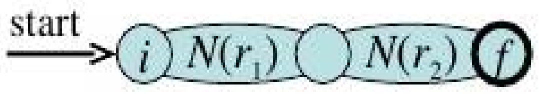
`a`



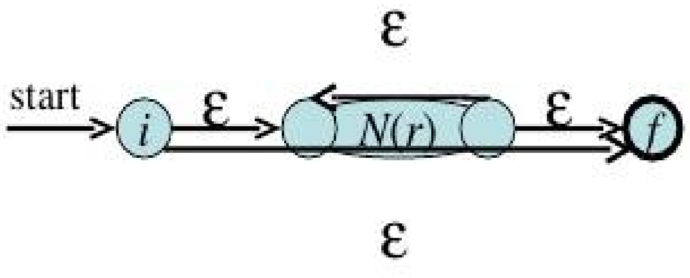
$r_1 \mid r_2$



$r_1 r_2$



r^*



Thompson's Construction

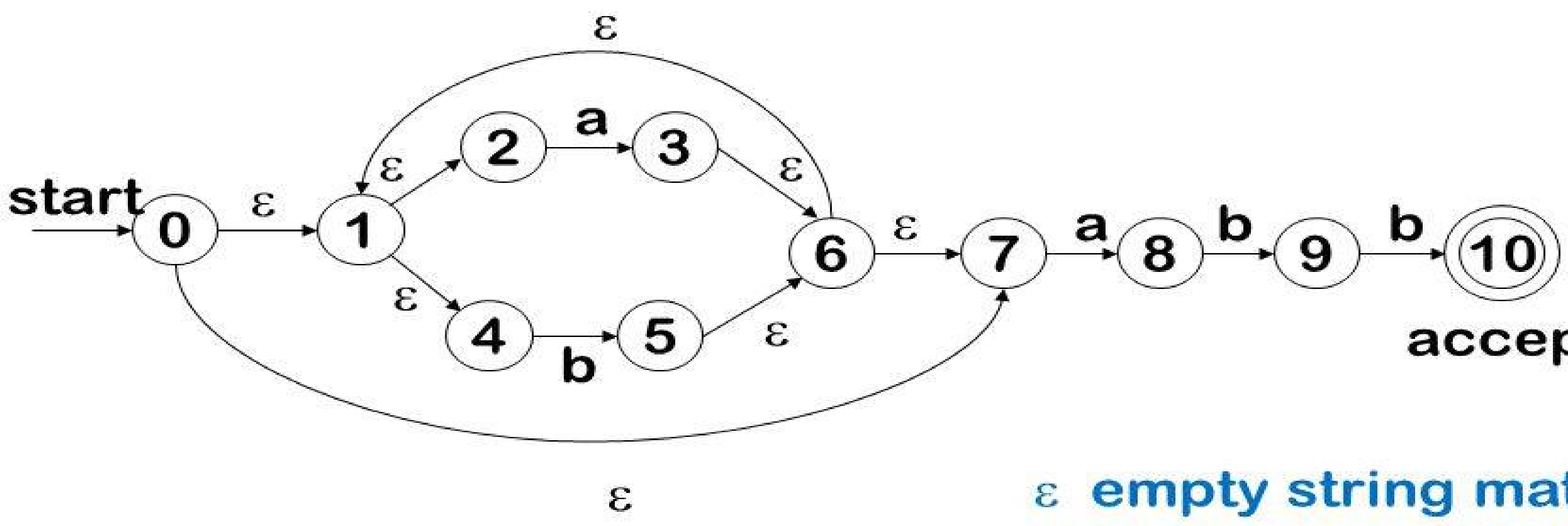
NFA properties

- Each NFA has a single start state and a single final state
- The only transition that enters the initial state is the initial transition
- No transitions leave the final state
- An empty transition always connects two states that were start or final states of a component NFA
- A state has at most two entering and two exiting empty transitions

try to convince yourself that these properties hold

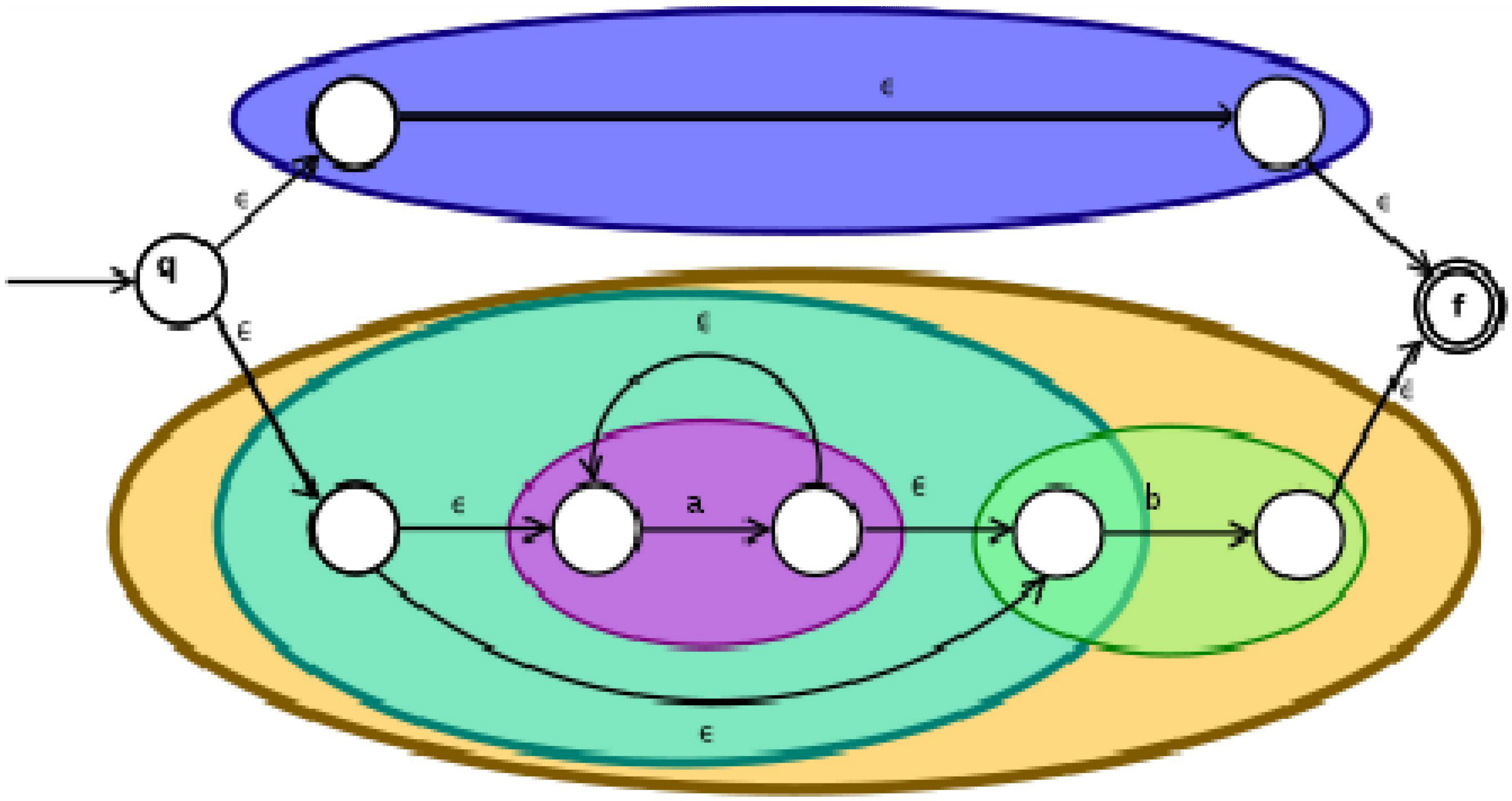
From a Regular Expression to an NFA Thompson’s Construction

(a | b)* abb

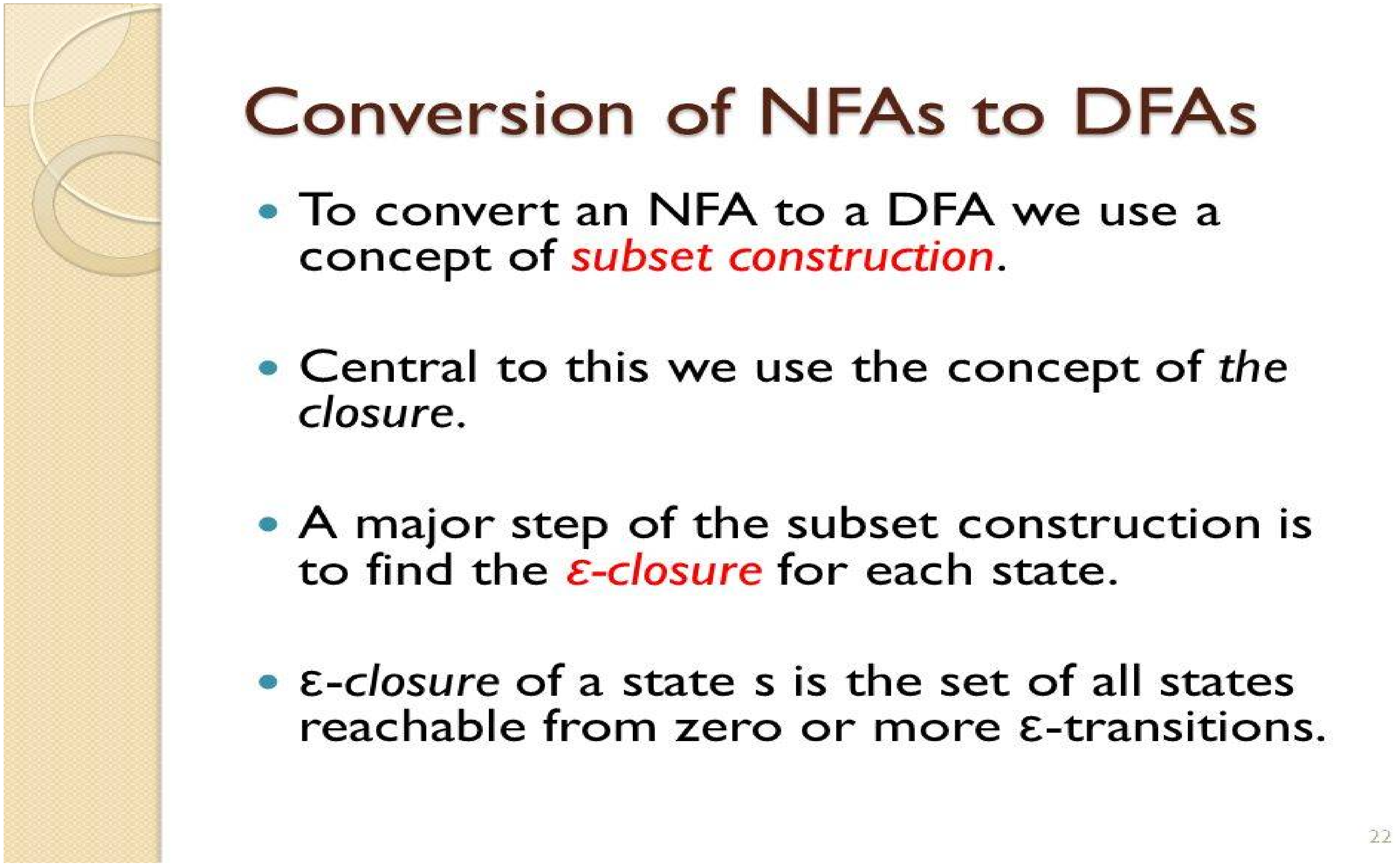


ε empty string match consumes no input

Example :(ε|a*b) using Thompson's construction



Subset Construction



Conversion of NFAs to DFAs

- To convert an NFA to a DFA we use a concept of *subset construction*.
- Central to this we use the concept of *the closure*.
- A major step of the subset construction is to find the *ε-closure* for each state.
- *ε-closure* of a state s is the set of all states reachable from zero or more ε-transitions.

22

How to compute €-Closure

- The €-closure function takes a state and returns the set of states reachable from it based on (one or more) € -transitions.
- Note that this will always include the state itself. We should be able to get from a state to any state in its € -closure without consuming any input.
- The function move takes a state and a character, and returns the set of states reachable by one transition on this character.

The Subset Construction Algorithm

- Create the start state of the DFA by taking the €-closure of the start state of the NFA.
- Perform the following for the new DFA state:
  For each possible input symbol:
  - Apply move to the newly-created state and the input symbol; this will return a set of states.
  - Apply the € -closure to this set of states, possibly resulting in a new set.
- This set of NFA states will be a single state in the DFA.
- Each time we generate a new DFA state, we must apply step 2 to it. The process is complete when applying step 2 does not yield any new states.
- The finish states of the DFA are those which contain any of the finish states of the NFA.

Minimization of DFA

Suppose there is a DFA , D < Q, ∑, q0, δ, F > which recognizes a language L. Then the minimized DFA, D < Q’, ∑, q0, δ’, F’ > can be constructed for language L as:

Step 1: We will divide Q (set of states) into two sets. One set will contain all final states and other set will contain non-final states. This partition is called P0.
Step 2: Initialize k = 1
Step 3: Find P_k by partitioning the different sets of P_{k-1}. In each set of P_{k-1}, we will take all possible pair of states. If two states of a set are distinguishable, we will split the sets into different sets in P_k.
Step 4: Stop when $P_k = P_{k-1}$ (No change in partition)
Step 5: All states of one set are merged into one. No. of states in minimized DFA will be equal to no. of sets in P_k.

Input Buffering in Compiler Design

- To ensure that a right lexeme is found, one or more characters have to be looked up beyond the next lexeme.
- Hence a two-buffer scheme is introduced to handle large look aheads safely.
- Techniques for speeding up the process of lexical analyzer such as the use of sentinels to mark the buffer end have been adopted.

There are three general approaches for the implementation of a lexical analyzer:

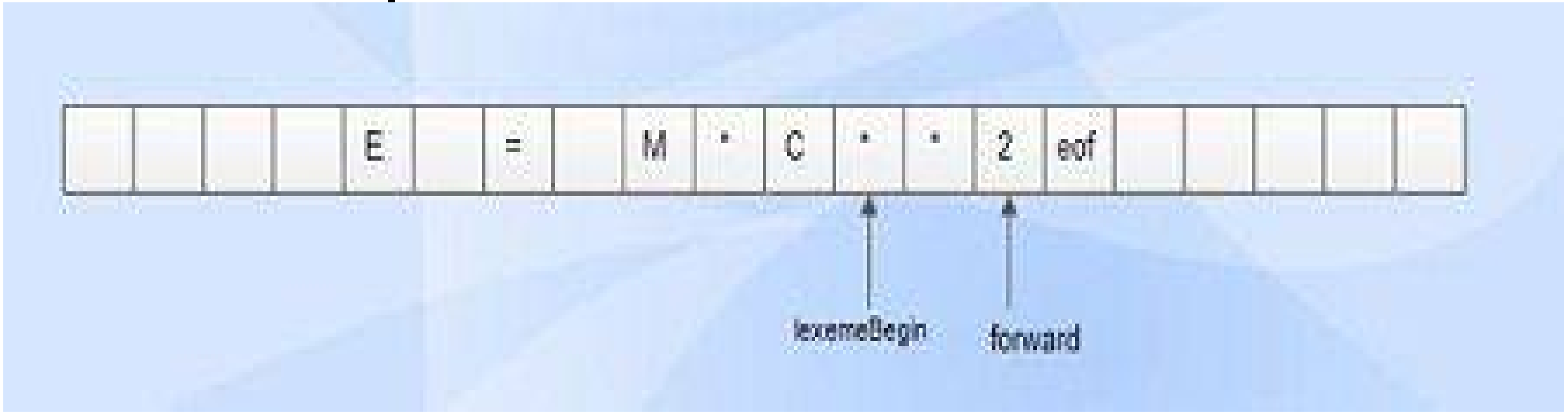
- (i) By using a lexical-analyzer generator, such as lex compiler to produce the lexical analyzer from a regular expression based specification. In this, the generator provides routines for reading and buffering the input.

- (ii) By writing the lexical analyzer in a conventional systems-programming language, using I/O facilities of that language to read the input.

- (iii) By writing the lexical analyzer in assembly language and explicitly managing the reading of input.

Input Buffering in Compiler Design

- Because of large amount of time consumption in moving characters, specialized buffering techniques have been developed to reduce the amount of overhead required to process an input character.
- Fig shows the buffer pairs which are used to hold the input data.



Scheme

- Consists of two buffers, each consists of N-character size which are reloaded alternatively.
- N-Number of characters on one disk block, e.g., 4096.
- N characters are read from the input file to the buffer using one system read command.
- *eof* is inserted at the end if the number of characters is less than N.

Pointers

- Two pointers *lexemeBegin* and *forward* are maintained.
- ***lexeme Begin*** points to the beginning of the current lexeme which is yet to be found.
- ***forward*** scans ahead until a match for a pattern is found.
- Once a lexeme is found, *lexemebegin* is set to the character immediately after the lexeme which is just found and *forward* is set to the character at its right end.
- Current lexeme is the set of characters between two pointers.

Disadvantages of this scheme

- This scheme works well most of the time, but the amount of lookahead is limited.
- This limited lookahead may make it impossible to recognize tokens in situations where the distance that the forward pointer must travel is more than the length of the buffer.

  (e.g.) DECLARE (ARGl, ARG2, . . . , ARGn) in PL/1 program;
- It cannot determine whether the DECLARE is a keyword or an array name until the character that follows the right parenthesis.

Sentinels

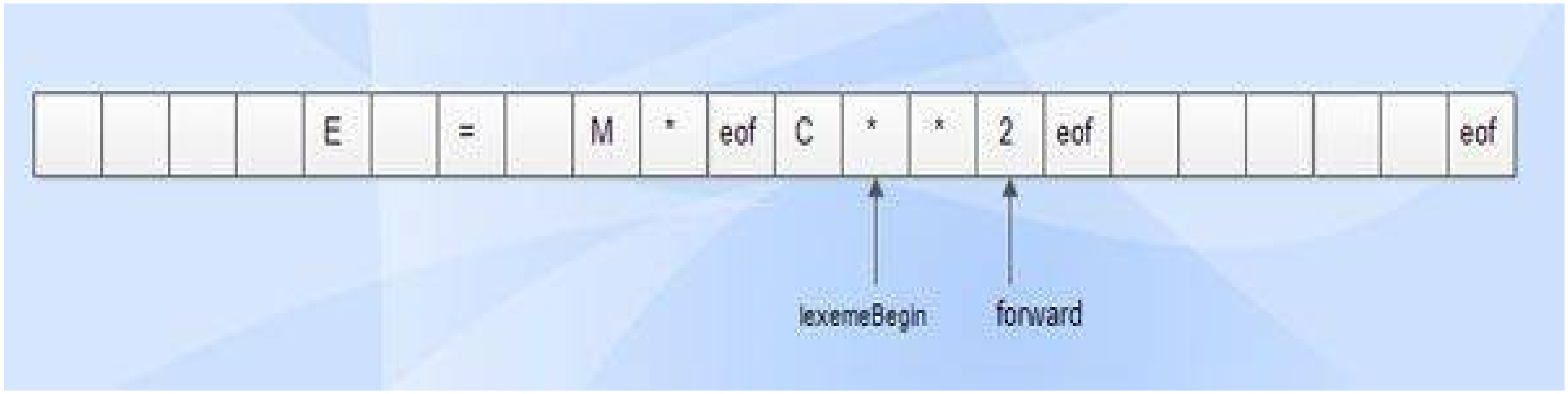
- In the previous scheme, each time when the forward pointer is moved, a check is done to ensure that one half of the buffer has not moved off. If it is done, then the other half must be reloaded.
- Therefore the ends of the buffer halves require two tests for each advance of the forward pointer.

Test 1: For end of buffer.

Test 2: To determine what character is read.

- The usage of sentinel reduces the two tests to one by extending each buffer half to hold a sentinel character at the end.
- The sentinel is a special character that cannot be part of the source program. (*eof* character is used as sentinel).

Advantages



- Most of the time, It performs only one test to see whether forward pointer points to an *eof.*
- Only when it reaches the end of the buffer half or *eof,* it performs more tests.
- Since N input characters are encountered between *eofs,* the average number of tests per input character is very close to 1.

Syntactic Specification of a Programming Language

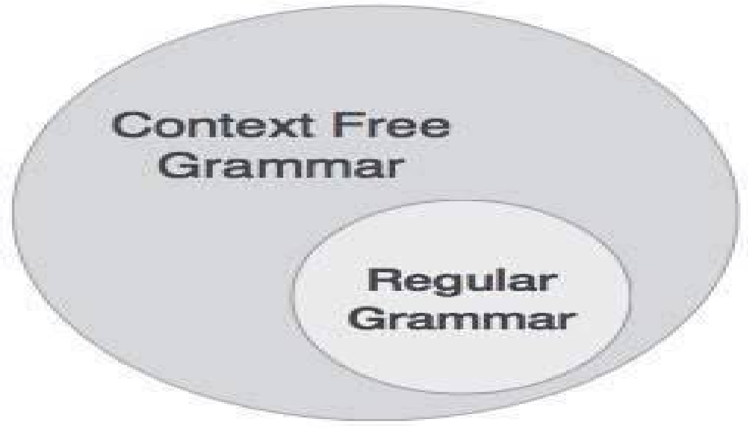
- CFG
- Capabilities of CFG
- Derivations
- Parse Trees
- Ambiguity
- BNF Notation

Regular Expression limitation.....

- We have seen that a lexical analyzer can identify tokens with the help of regular expressions and pattern rules. But a lexical analyzer cannot check the syntax of a given sentence due to the limitations of the regular expressions. Regular expressions cannot check balancing tokens, such as parenthesis. Therefore, CFG is a helpful tool in describing the syntax of programming languages.

Context-Free Grammar

- CFG is a superset of Regular Grammar, as depicted below:
- It implies that every Regular Grammar is also context-free
- CFG is a helpful tool in describing the syntax of programming languages.



What are Context Free Grammars?

- In Formal Language Theory , a Context free Grammar(CFG) is a formal grammar in which every production rule is of the form

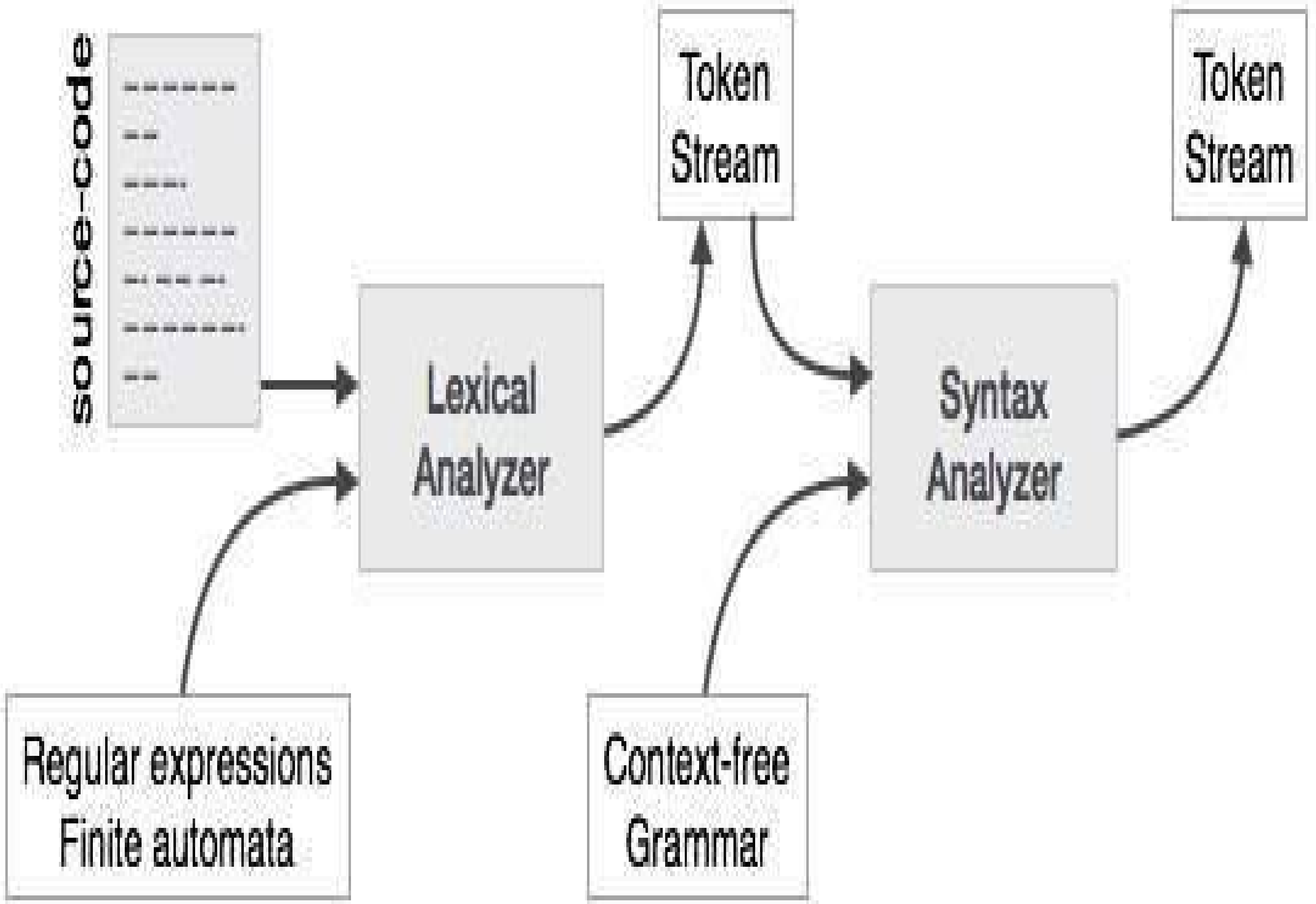
$V \longrightarrow w$

Where V is a single nonterminal symbol and w is a string of terminals and/or nonterminals (w can be empty)

- The languages generated by context free grammars are knows as the context free languages

The capabilities of CFG :

- Context free grammars are capable of describing most of the syntax of programming language.
- Suitable grammars for expressions can often be constructed using associatively & precedence information.
- So, context free grammar are most useful in describing nested structures such as balanced parentheses, matching begin-end's, corresponding if-then-else's & so on. These nested structures cannot be described by regular expression.
- The following grammars the string, which serves the language.


source-code
Lexical Analyzer
Token Stream
Syntax Analyzer
Token Stream
Regular expressions Finite automata
Context-free Grammar

Derivation

- A derivation is basically a sequence of production rules, in order to get the input string.
- During parsing, we take two decisions for some sentential form of input:

1. Deciding the non-terminal which is to be replaced.

2. Deciding the production rule, by which, the non-terminal will be replaced.

Derivation

To decide which non-terminal to be replaced with production rule, we can have two options.

- **Left-most Derivation:** If the sentential form of an input is scanned and replaced from left to right, it is called left-most derivation. The sentential form derived by the left-most derivation is called the left-sentential form.
- **Right-most Derivation:** If we scan and replace the input with production rules, from right to left, it is known as right-most derivation. The sentential form derived from the right-most derivation is called the right-sentential form.

Example: Input string: id + id * id

- Production rules:

E → E + E

E → E * E

E → id

- The left-most derivation is:

E → E * E

E → E + E * E

E → id + E * E

E → id + id * E

E → id + id * id

Notice that the left-most side non-terminal is always processed first.

Example: Input string: id + id * id

- Production rules:

E → E + E

E → E * E

E → id

- The right-most derivation is:

E → E + E

E → E + E * E

E → E + E * id

E → E + id * id

E → id + id * id

Parse Tree

- In a parse tree:
- All leaf nodes are terminals.
- All interior nodes are non-terminals.
- In-order traversal gives original input string.
- A parse tree depicts associativity and precedence of operators.
- The deepest sub-tree is traversed first, therefore the operator in that sub-tree gets precedence over the operator which is in the parent nodes.

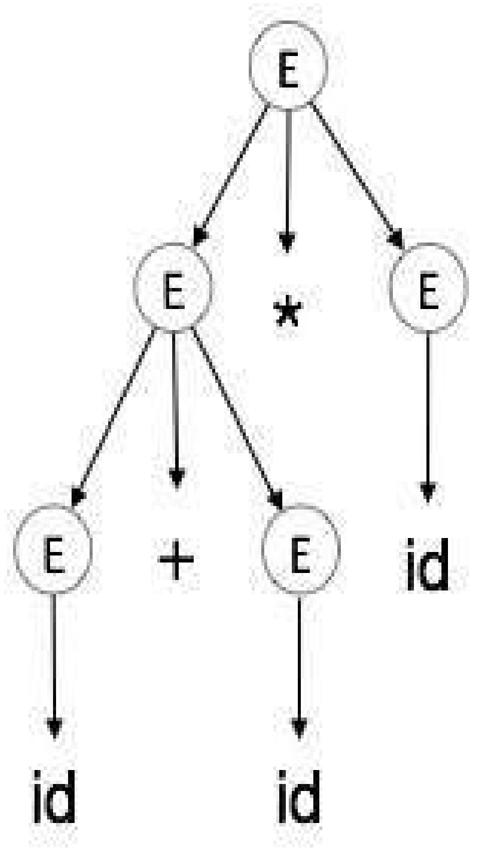
Parse Tree

- A parse tree is a graphical depiction of a derivation.
- It is convenient to see how strings are derived from the start symbol.
- The start symbol of the derivation becomes the root of the parse tree.

E → id + id * id

We take the left-most derivation of

a + b * c

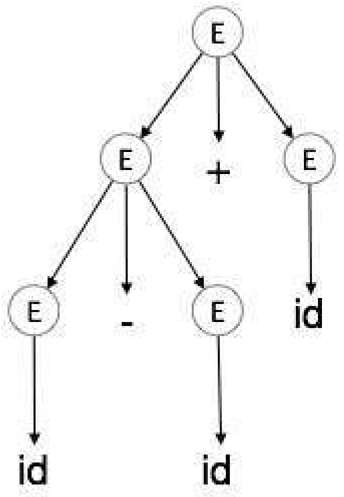


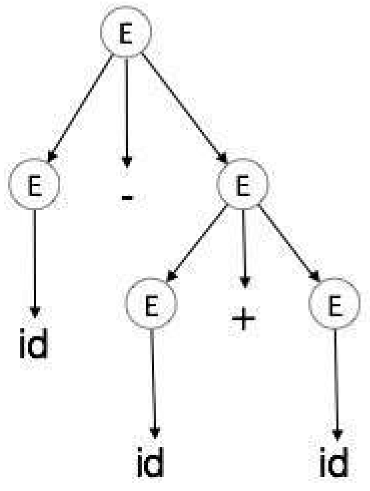
Ambiguity

- A grammar G is said to be ambiguous if it has more than one parse tree (left or right derivation) for at least one string.

Example: $E \rightarrow E + E$ $\quad E \rightarrow E – E$ $\quad E \rightarrow id$

- For the string, **id + id – id**, the above grammar generates two parse trees:





Ambiguity

- The language generated by an ambiguous grammar is said to be **inherently ambiguous**. Ambiguity in grammar is not good for a compiler construction. No method can detect and remove ambiguity automatically, but it can be removed by either re-writing the whole grammar without ambiguity, or by setting and following **associativity and precedence** constraints.

Associativity

- If an operand has operators on both sides, the side on which the operator takes this operand is decided by the associativity of those operators. If the operation is left-associative, then the operand will be taken by the left operator or if the operation is right-associative, the right operator will take the operand.

Example

- Operations such as Addition, Multiplication, Subtraction, and Division are left associative.
- If the expression contains**:** **id op id op id**
- it will be evaluated as: (id op id) op id
- For example, (id + id) + id
- Operations like Exponentiation are right associative, i.e., the order of evaluation in the same expression will be:
- id op (id op id)
- For example, id ^ (id ^ id)

Precedence

- If two different operators share a common operand, the precedence of operators decides which will take the operand.
- That is, 2+3*4 can have two different parse trees, one corresponding to (2+3)*4 and another corresponding to 2+(3*4).
- By setting precedence among operators, this problem can be easily removed.
- These methods decrease the chances of ambiguity in a language or its grammar.

BNF Notation

- One of the earliest forms of notation used for describing the syntax of a programming language was **Backus-Naur Form** (BNF).
- This is basically just a variant of a context-free grammar, with the symbol **``::="** used in place of “→” to mean ``is defined as''.
- Additionally, non-terminals are usually written in **angle-brackets** and terminals in **quotes.**

- For example, we could describe a block of statements as:
- <block> ::=``BEGIN''  <opt-stats> ``END''.
- <opt-stats> ::= <stats-list> | € .
- <stats-list> ::= <statement >.
- <stats-list> ::= < statement> ``;'' <stats-list> .

Compiler Construction Tools

- These tools can be used to implement a compiler.
- These tools have been created fot the automatic design of specific compiler components.
- These tools use specialized languages for specifying & implementing the component.
- These tools hide the details of generation algorithm & produce components that can be easily integrated into the remainder of a compiler.

Compiler Construction Tools

- Some commonly used compiler-construction tools include:
  1. Parser generators.
  2. Scanner generators.
  3. Syntax-directed translation engines.
  4. Automatic code generators.
  5. Data-flow analysis engines.
  6. Compiler-construction toolkits.

- **Scanner Generators**

  Input: Regular expression description of the tokens of a language
  Output: Lexical analyzers.

  Scanner generator generates lexical analyzers from a regular expression description of the tokens of a language.

- **Parser Generators:**

Input: Grammatical description of a programming language

Output: Syntax analyzers.

Parser generator takes the grammatical description of a programming language and produces a syntax analyzer.

- **Syntax-directed Translation Engines**

  Input: Parse tree.
  Output: Intermediate code.

  Syntax-directed translation engines produce collections of routines that walk a parse tree and generates intermediate code.

- **Automatic Code Generators**

  Input: Intermediate language.
  Output: Machine language.

  Code-generator takes a collection of rules that define the translation of each operation of the intermediate language into the machine language for a target machine.

- **Data-flow Analysis Engines:**

  Data-flow analysis engine gathers the information, that is, the values transmitted from one part of a program to each of the other parts. Data-flow analysis is a key part of code optimization.

- **Compiler Construction Toolkits**

  The toolkits provide integrated set of routines for various phases of compiler. Compiler construction toolkits provide an integrated set of routines for construction of phases of compiler.

- A compiler for a programming language is often decomposed into two parts:

1. Read the source program & discover its structure.
2. Process this structure

The task of discovering the source program again is decomposed into subtasks:

1. Split the source file into Tokens(LEX)
2. Find the hierarchical structure of the program(YACC)

LEX: A Lexical Analyzer Generator

Lex source is a table of regular expressions and corresponding program fragments.

The table is translated to a program which reads an input stream, copying it to an output stream and partitioning the input into strings which match the given expressions.

As each such string is recognized, the corresponding program fragment is executed.

The recognition of the expressions is performed by a deterministic finite automaton generated by Lex.

The program fragments written by the user are executed in the order in which the corresponding regular expressions occur in the input stream.

FLEX: A Fast Scanner Generator

- flex is a tool for generating scanners.
- Programs which recognized lexical patterns in text, flex reads the given input files, for a description of a scanner to generate.
- The description is in the form of pairs of regular expressions and C code, called rules.
- flex generates as output a C source file, \`lex.yy.c', which defines a routine \`yylex()'.
- This file is compiled and linked with the \`-lfl' library to produce an executable.
- When the executable is run, it analyzes its input for occurrences of the regular expressions. Whenever it finds one, it executes the corresponding C code.

Lex source program format

- The Lex program has three sections, separated by %%:

```
declarations
%%
transition rules
%%
auxiliary code
```

Transition rules

- The *translation rules* of a Lex program are statements of the form :
- *p1* *{action 1}*
- *p2* *{action 2}*
- *p3* *{action 3}*

  … …

  where each *p* is a regular expression and each *action* is a program fragment describing what action the lexical analyzer should take when a string pattern matches with *p* matches a lexeme.

  In Lex the actions are written in C.

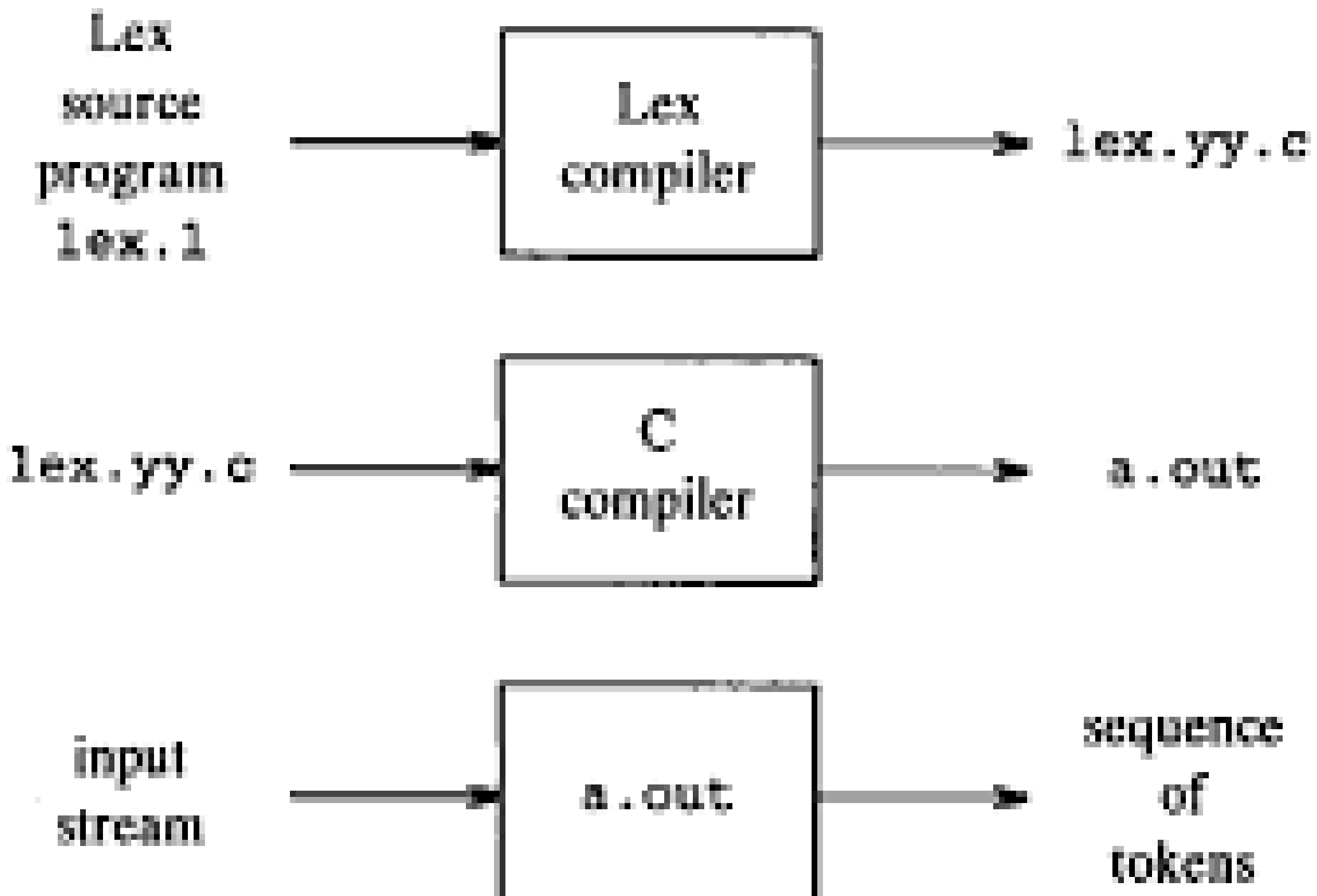


Fig. 3.17. Creating a lexical analyzer with Lex.

LEX

→ It is a tool used to specify lexical analyzers

→ It uses lexical language (i.e. Pattern) for its input specification.

→ when lex-specification is given as i/p to LEX-compiler, it generates as o/p a C source file, 'lex.yy.c'

→ when this 'lex.yy.c' is given as i/p to the C-compiler in produces an executable file a.out.

→ when the executable is run, it analyzes i/p & produces sequence of token

Lex-specification consists of three sections –

- Declaration → Declaration of variables & their regular definitions

  %%

- Transition Rules → Rules are stated i.e. $P_i \rightarrow$ action → written in C or in any implementation language

  (Pattern/R.E.)

  %%

- Auxiliary Rules → Additional Procedures needed

  %%

install_id()
install_num() } to install the lexeme

yylval, yyleng, yytext are the variables used.

yylval → pass an attribute value with information

yyleng → tells the length of the lexeme

yytext → pointer to the first character of the lexeme

YACC: Yet Another Compiler Compiler

- Yacc provides a general tool for describing the input to a computer program.
- The Yacc user specifies the structures of his input, together with code to be invoked as each such structure is recognized.
- Yacc turns such a specification into a subroutine that handles the input process.
- it is convenient and appropriate to have most of the flow of control in the user's application handled by this subroutine.

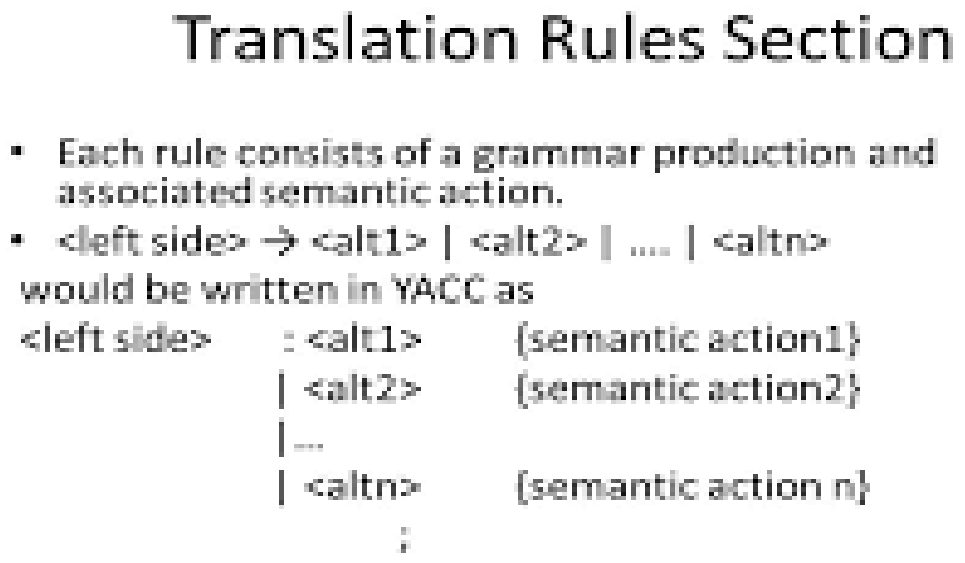
Bison: The YACC-compatible Parser Generator

- Bison is a general-purpose parser generator that converts a grammar description for an LALR(1) context-free grammar into a C program to parse that grammar.

YACC Specifications

- Similar structure to LEX
  - Declaration Section
  - Translation Rules
  - Supporting C/C++ code

```
Declaration
%%
Translation rules
%%
Supporting C/C++ code
```



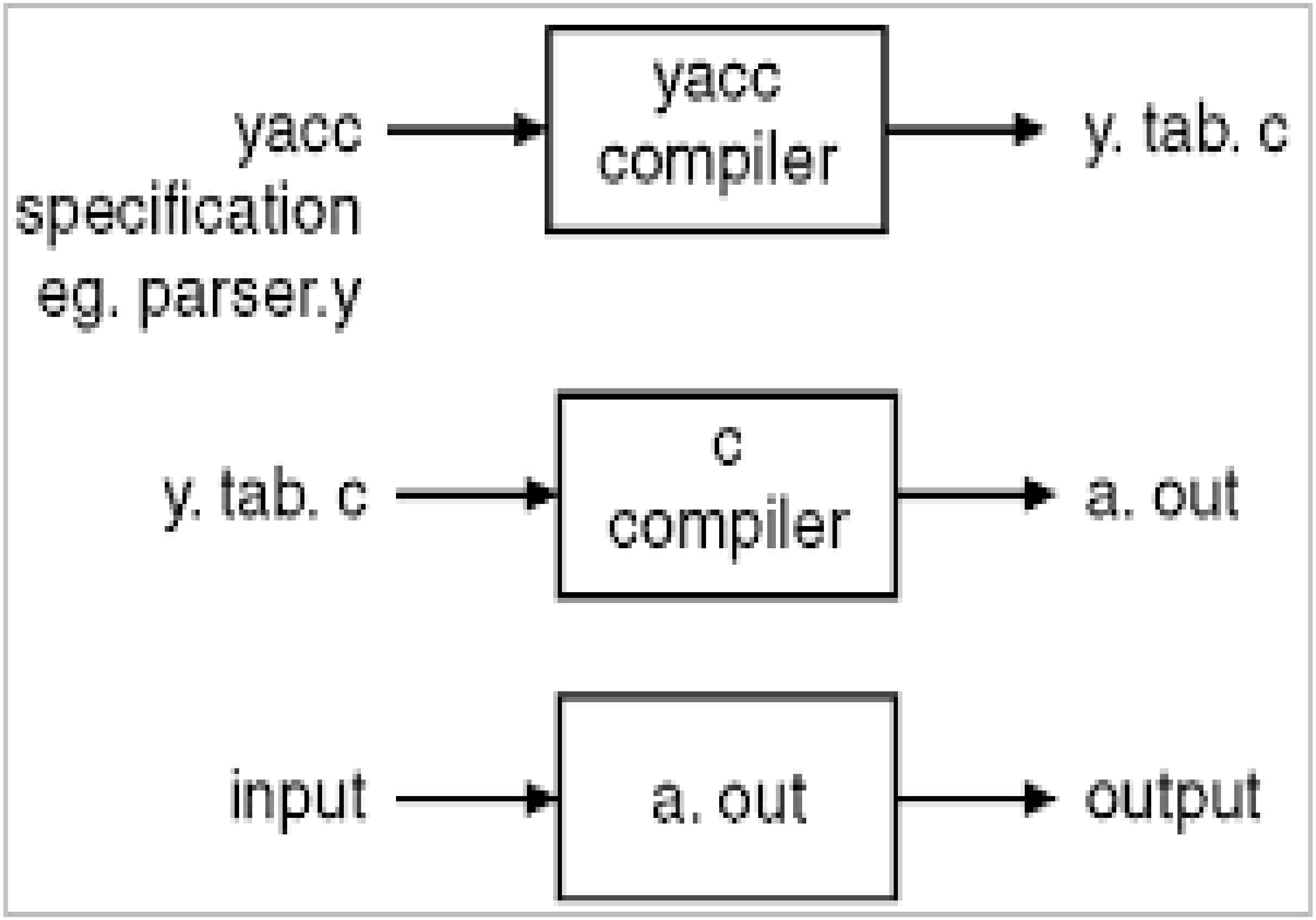
Translation Rules Section

- Each rule consists of a grammar production and associated semantic action.
- <left side> → <alt1> | <alt2> | .... | <altn>

would be written in YACC as

```
<left side>      : <alt1>        {semantic action1}
                 | <alt2>        {semantic action2}
                 |...
                 | <altn>        {semantic action n}
                        ;
```

Semantic Action is a sequence of C- Statements.


yacc specification eg. parser.y
yacc compiler
y. tab. c
y. tab. c
c compiler
a. out
input
a. out
output

YACC

→ Acronym for Yet Another Compiler Compiler
→ It is a tool for imposing structure on the i/p
→ YACC specification is given as CFG.

A file translate.y (containing a YACC specification) is prepared.

YACC compiler transforms the file translate.y into a C program called y.tab.c, using LALR method.

By compiling y.tab.c on C-compiler, we obtain the desired object program a.out.

a.out performs the translation specified by the original YACC program.

* If other procedures are needed, they can be compiled or loaded with y.tab.c

YACC specification consists of 3-sections

- Declarations
  %   %
- Translation Rules : <left side> : <alt> { semantic Action }
  %   %
- supporting C-routines

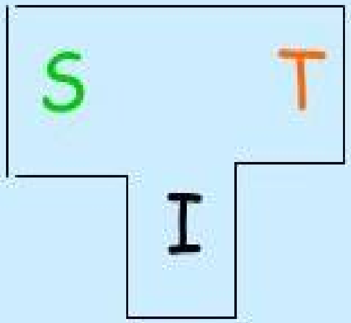
⇒ semantic Action is a sequence of C-statements.

Bootstrapping

- In computer science, bootstrapping is the **technique for producing a self-compiling compiler** - that is, using the facilities offered by a language to compile itself is the essence of bootstrapping.
- Even C compilers are written in C.
- The chicken and egg problem: If one needs to compile a compiler for language X (written in language X), there is the issue of how the first compiler can be compiled.
- The T-diagram is a notation used to explain the compiler bootstrap techniques.

Bootstrapping ...

- A compiler can be characterized by three languages: the source language (S), the target language (T), and the implementation language (I)
- The three language S, I, and T can be quite different. Such a compiler is called cross-compiler
- This is represented by a T-diagram as:



- In textual form this can be represented as

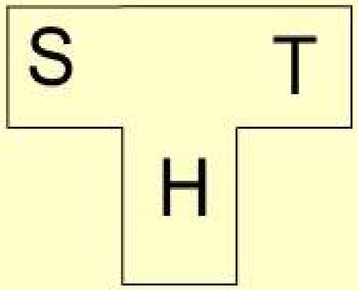
$$S_I T$$

T-diagrams

- Tombstone diagrams (T-diagrams) consist of a set of "puzzle pieces" representing compilers and other related language processing programs.
- They are used to illustrate and reason about transformations from a source language(left of T) to a target language (right of T) realized in an implementation language(bottom of T).
- They are most commonly found in describing complicated processes for bootstrapping, porting, and self-compiling of compilers, interpreters, and macro-processors.
- T-diagrams were first introduced for describing bootstrapping and cross-compiling compilers

Cross Compiler

- a compiler which generates target code for a different machine from one on which the compiler runs.
- A host language is a language in which the compiler is written.
  - T-diagram



- Cross compilers are used very often in practice.

Cross Compiler

- A cross compiler is a compiler capable of creating executable code for a platform other than the one on which the compiler is running.
- For example, a compiler that runs on a **Windows 7**PC but generates code that runs on Android Smartphone is a cross compiler.
- Many mini computers and microprocessors compilers are implemented this way. They run on a bigger machine and produce object code for the smaller machine.

Note:

- When T-diagrams are put together ,the implementation language of the putting T must be the same as the source language of the existing compiler and that the target language of the existing compiler must be that same as the implementation language of the translated one.

Example:

- Suppose we write a cross compiler for a new language L in implementation language S to generate code for machine N.
- If an existing compiler for S runs on machine M and generates code for M.
- So, it can be thought of as an equation:

$$L_S N + S_M M = L_M N$$

Thus we get a compiler from L to N that runs on M.

Example

- For the advantage of bootstrapping to be realized fully, a compiler has to be written in a language it compiles.
- Suppose we write a compiler $L_L N$, development take place on machine M, where an existing compiler $L_M M$ is available for L.
- So, by cross compilation process, we obtain $L_M N$

$$L_L N + L_M M = L_M N$$

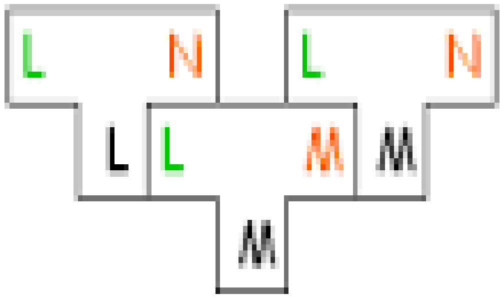
Example...

- The compiler L_LN can be compiled a second time, this time using the generated cross compiler,

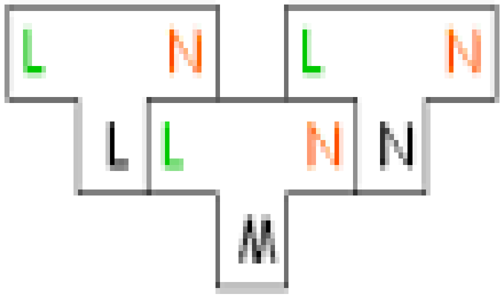
$$L_LN + L_MN = L_NN$$

- The result of the second compilation is a compiler that runs on N and generates code for N.
- Thus, using bootstrapping techniques, an optimizing compiler can optimize itself.

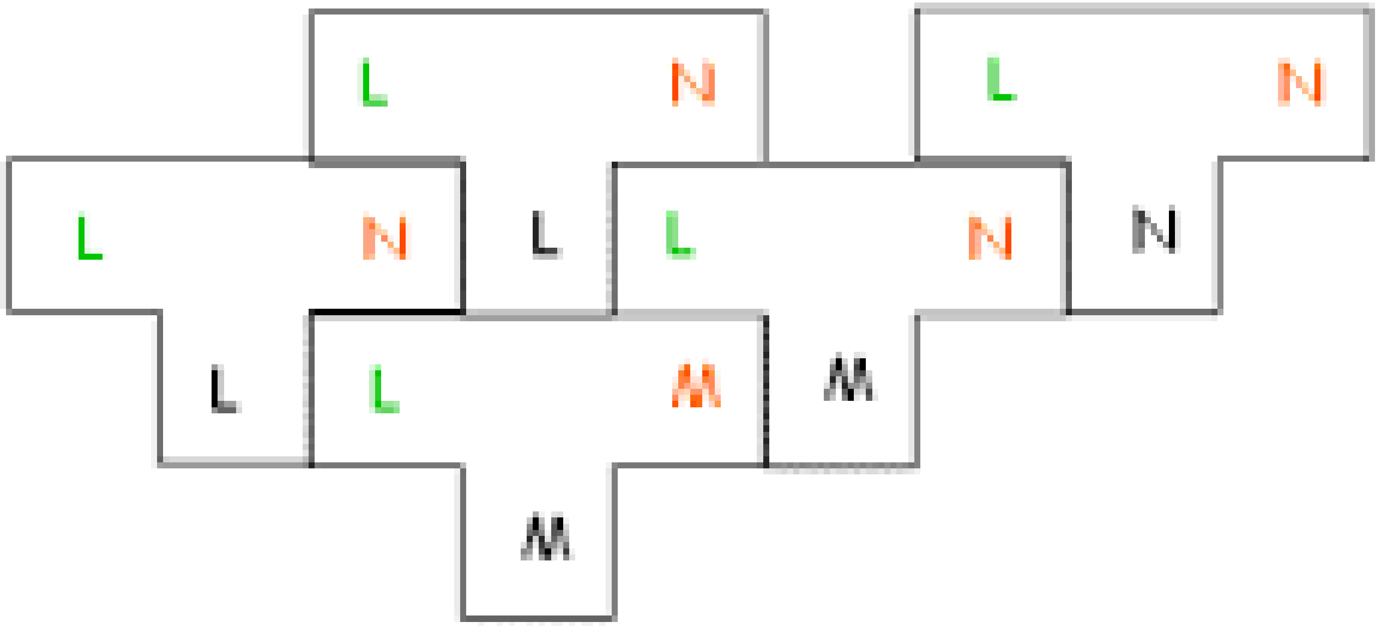
- Suppose $L_L N$ is to be developed on a machine M where $L_M M$ is available

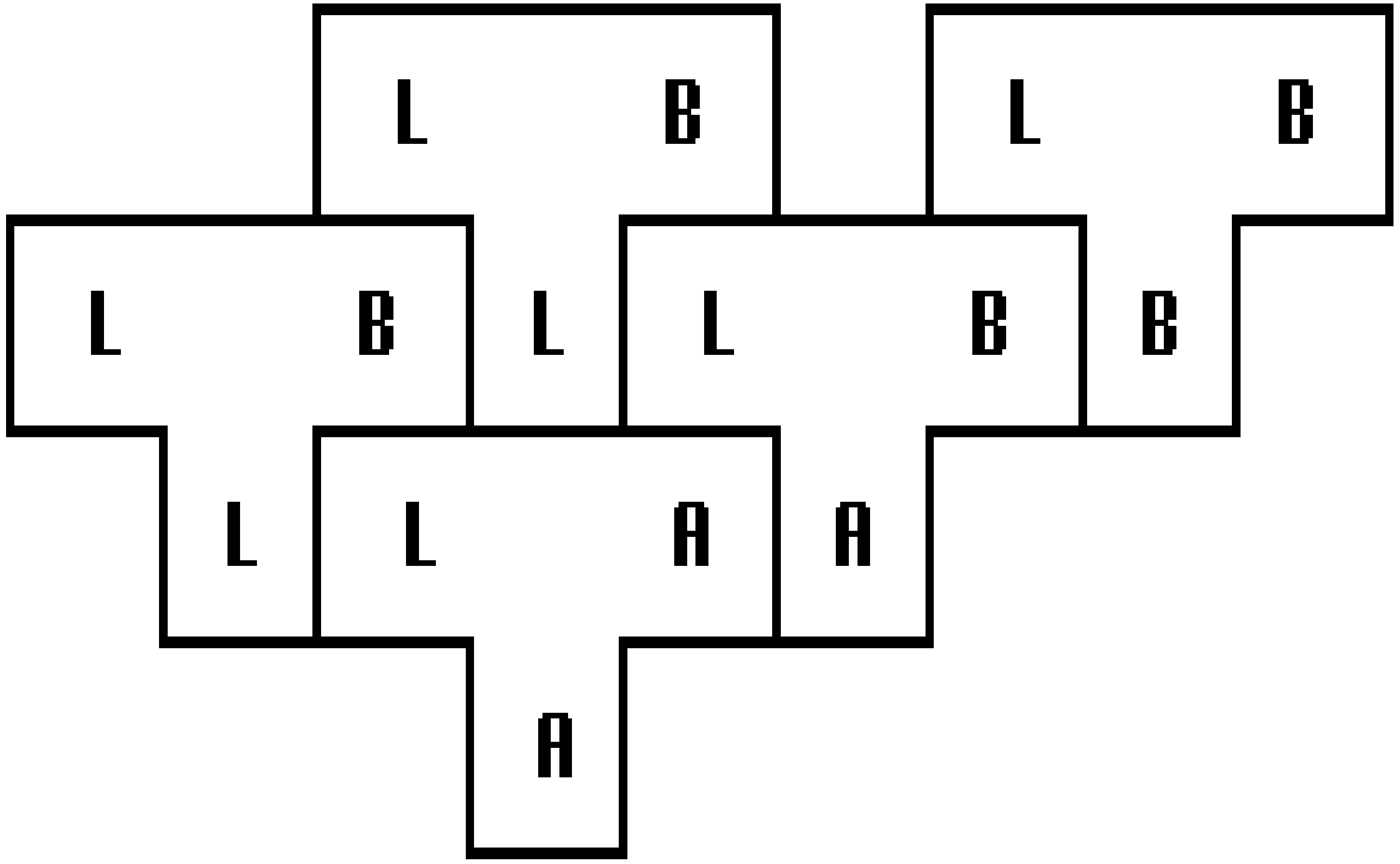


- Compile $L_L N$ second time using the generated compiler



Bootstrapping a Compiler: the Complete picture




L
B
L
B
L
B
L
L
B
B
L
L
A
A
A

END of UNIT-1