PES UNIVERSITY ONLINE

Compiler Design

Preet Kanwal

Department of Computer Science & Engineering

Teaching Assistant : Lavitra Kshitij Madan

Compiler Design

Unit 1

Compilers

Preet Kanwal

Department of Computer Science & Engineering

Compiler Design

Unit Overview

PES UNIVERSITY ONLINE

In this unit, you will learn about:

- Compilers, history and variants of compilers
- The language processing system
- The phases of a compiler
- Writing grammar for C Language
- Parsing code in C Language as per the grammar
- Lexical Analysis
- Design of a Lexical Analyzer Generator
- Specification and recognition of Tokens
- Implementation of a lexer

Compiler Design

Unit Outcomes

After studying this unit, you will be able to:

- Explain how GCC transforms source files to an executable file
- Construct a symbol table
- Parse an input in C Language containing Variable Declaration (+ initialization) Statements, Expressions (involving binary, unary, logical and relational operators), if, if-else, while, do-while, for-loop
- Draw a syntax tree for each of the above mentioned statements
- Use and write lex and yacc programs
- Create a lexer for C language using the lex tool

Compiler Design

Lecture Overview

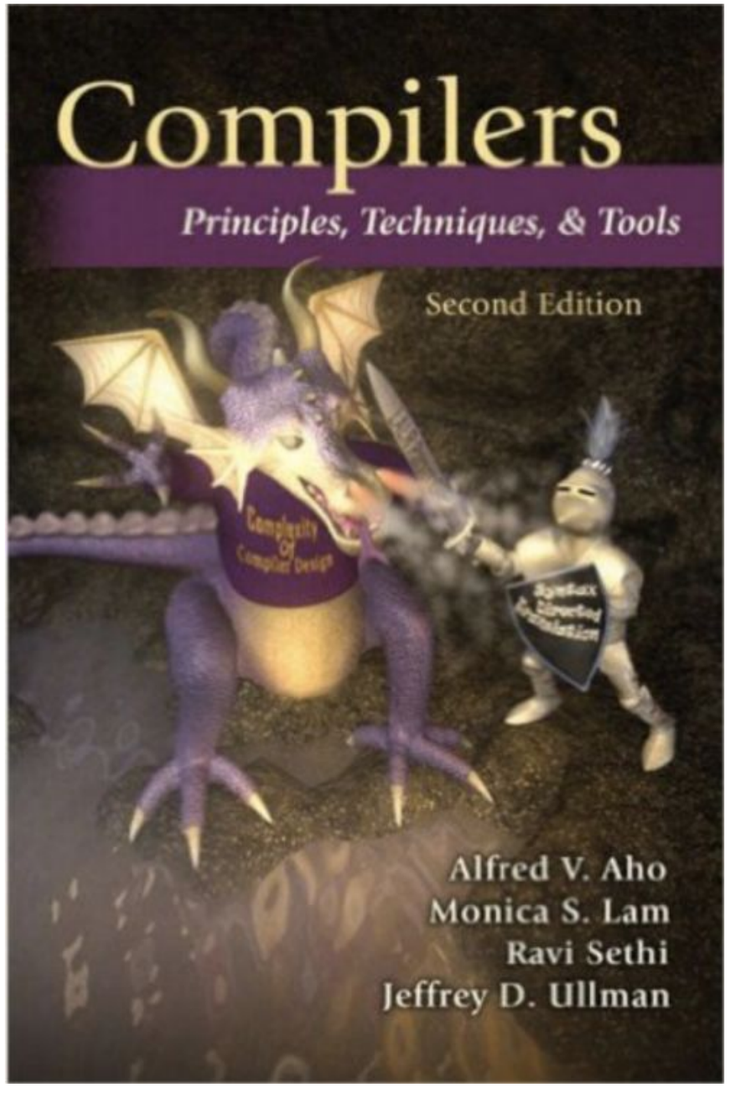
PES UNIVERSITY ONLINE

In this lecture, you will learn about:

- Introduction to the course
- Introduction to compilers
- History of compilers
- Variants of compilers:
  - Self-hosting Compilers
    - Bootstrapping
  - Cross Compiler
  - Transpiler
  - Decompiler
  - Compiler-Compiler

Compiler Design

Introduction: Compiler



T1

R1

T1 -

"Compilers–Principles, Techniques and Tools" Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffery D. Ullman; 2nd Edition

R1 –

"Modern Compiler Design", Dick Grune, Kees van Reeuwijk, Henri E. Bal, Ceriel J.H. Jacobs, Koen Langendoen; 2nd Edition

Compiler Design

Why study compilers?

- Build a large, ambitious software system. See theory come to life.
- Learn how to build programming languages.
- Learn how programming languages work.
- Learn tradeoffs in language design.

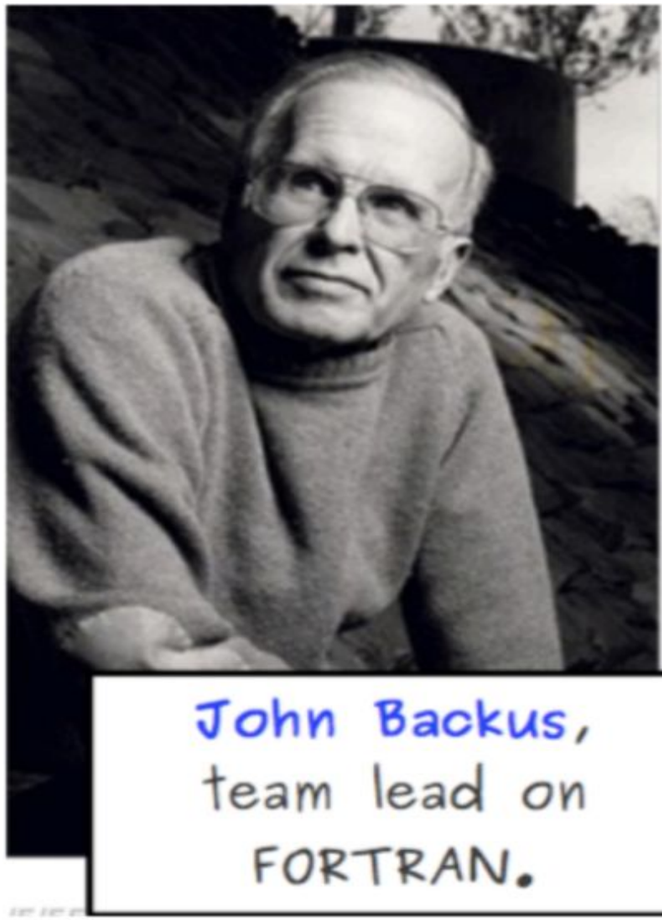
A short history of compilers:-
- First, there was nothing.
- Then, there was machine code.
- Then, there were assembly languages.
- Programming expensive; 50% of costs for machines went into programming.

Compiler Design

Introduction: Compiler

Rear Admiral Grace Hopper:

Inventor of A-0(1952), COBOL(1959), and the term "compiler."



FORTRAN (Formula Translation) : first widely used high level programming language. First unambiguously complete compiler. (1954-57)

Compiler Design

First C Compiler

- First C Compiler (Modified B Compiler : Written in B)

- TMG (TransMoGrifier) was the compiler definition tool used by Ken Thompson to write the compiler for the B language on his PDP-7 (an assembler) in 1970.

- B was the immediate ancestor of C.

Compiler Design

Self-hosting Compiler

- **Self-hosting Compiler : Compiler that can compile its own source code.**

- **Example : C, C++ , C#, Java, Pascal, Python, VisualBasic etc**

- **In some of these cases, the initial implementation was not self-hosted, but rather, written in another language (or even in machine language); in other cases, the initial implementation was developed using bootstrapping.**

Compiler Design

Bootstrapping (Compilers)

- **Bootstrapping is the technique for producing a self- compiling compiler.**
- **An initial core version of the compiler - the bootstrap compiler - is generated in a different language (which could be assembly language); successive expanded versions of the compiler are developed using this minimal subset of the language.**
- **Example : BASIC, ALGOL, C, D, Pascal, Lisp, Java, Rust, Python.**

- **Assemblers were the first language tools to bootstrap themselves!**

Compiler Design

Cross Compiler

PES UNIVERSITY ONLINE

Native Compiler

Generates code for the same Platform on which it runs. Converts high language into computer's native language.

Example: Turbo C or GCC compiler

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. USE : Embedded Computers (Microwave oven, Washing Machine)

A cross compiler is for cross-platform software development of binary code

Compiler Design

Other Variants

TRANSPILER : Source-to-Source Compiler

(High level language to High level Language)

DECOMPILER : Low-level lang to High-level lang.

Compiler-compiler refers to tools used to create parsers that perform syntax analysis.

Compiler Design

Other Variants - Compiler-compiler (Compiler generator)

- A programming tool that creates a parser, interpreter, or compiler from some form of formal description of a programming language and machine.
- The most common type of compiler-compiler is more precisely called a parser generator, and only handles syntactic analysis:
  - its input is a grammar, typically written in Backus–Naur form (BNF) or extended Backus–Naur form (EBNF) that defines the syntax of a programming language
  - and its output is source code of a parser for the programming language.
  - Parser generators do not handle the semantics of the programming language, or the generation of machine code for the target machine

THANK YOU

Preet Kanwal

Department of Computer Science & Engineering

preetkanwal@pes.edu

PES UNIVERSITY ONLINE

Compiler Design

Preet Kanwal

Department of Computer Science & Engineering

Teaching Assistant : Lavitra Kshitij Madan

Compiler Design

Unit 1

The Language Processing System

Preet Kanwal

Department of Computer Science & Engineering

Compiler Design

Lecture Overview

PES UNIVERSITY ONLINE

In this lecture, you will learn about:

- Introduction to GCC
- Introduction to the language processing system
- GCC transformations
- The phases of a compiler
- Grouping of phases into passes
  - Single pass
  - Traditional two-pass
  - Three pass

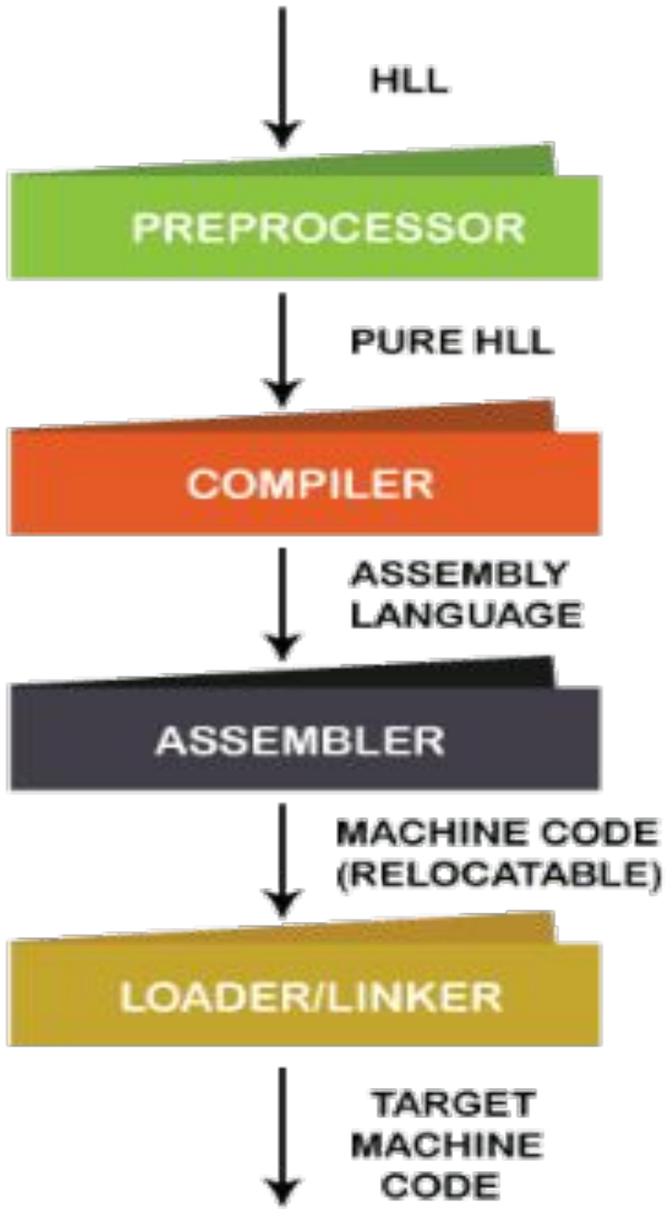
Compiler Design

GCC

- The acronym GCC is used to refer to the "GNU Compiler Collection". Over time GCC has been extended to support many additional languages, including Fortran, ADA, Java and ObjectiveC.
- GCC is written in C with a strong focus on portability, and can compile itself, so it can be adapted to new systems easily.
- GCC is a portable compiler--it runs on most platforms available today, and can produce output for many types of processors.
- GCC is not only a native compiler--it can also cross-compile any program, producing executable files for a different system from the one used by GCC itself.
- GCC has multiple language frontends, for parsing different languages. Programs in each language can be compiled, or cross-compiled, for any architecture. For example, an ADA program can be compiled for a microcontroller, or a C program for a supercomputer.

Compiler Design

Language Processing System



→ Ex1.c

→ gcc -E Ex1.c > Ex1.i
('.i' for C programs and '.ii' for C+ + programs)

→ gcc -S Ex1.i
(resulting assembly language is stored -> 'Ex1.s')

→ as Ex1.s -o Ex1.o

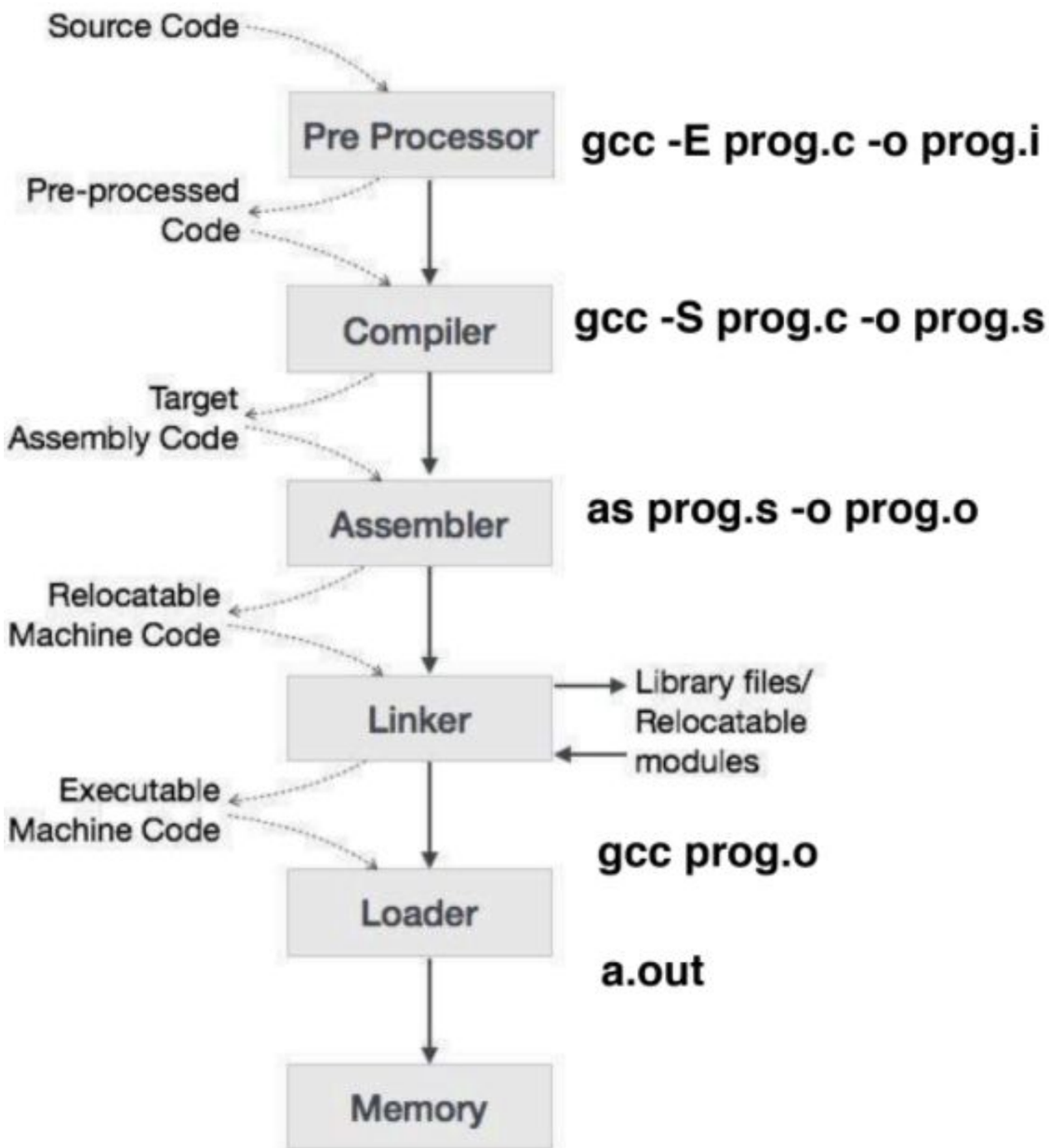
→ gcc Ex1.o // Linking
(links the object file 'Ex1.o' to the C standard library)

→ ./a.out //Loading

Content Of Symbol Table:- nm a.out

Compiler Design

The language processing system



Compiled version in memory is interpreted.

Compiler Design

Static Library

A **library** is a collection of pre-compiled object files that can be linked into your programs via the linker. Examples are the system functions such as printf() and sqrt().

There are two types of external libraries: **static library** and **shared library**.

A **static library** has file extension of ".a" (archive file) in Unixes or ".lib" (library) in Windows. When your program is linked against a static library, the machine code of external functions used in your program is copied into the executable. A static library can be created via the archive program “ar.exe".

Compiler Design

Shared Library

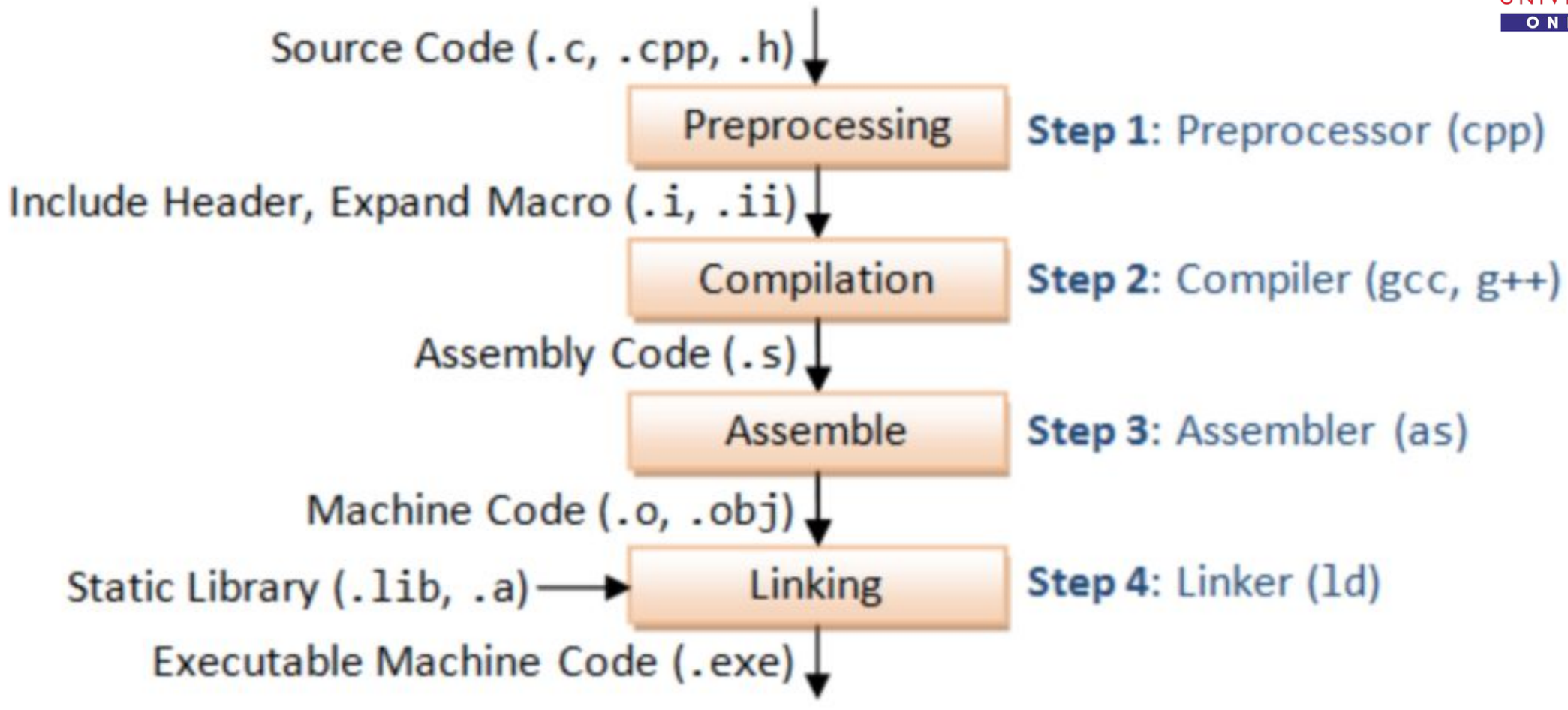
- A **shared library** has file extension of ".so" (shared objects) in Unixes or ".dll“ (dynamic link library) in Windows.
- When your program is linked against a shared library, only a small table is created in the executable.
- Before the executable starts running, the operating system loads the machine code needed for the external functions - a process known as *dynamic linking*.
- Furthermore, most operating systems allows one copy of a shared library in memory to be used by all running programs, thus, saving memory. The shared library codes can be upgraded without the need to recompile your program.

Dynamic linking

makes executable files smaller and saves disk space, because one copy of a library can be shared between multiple programs.

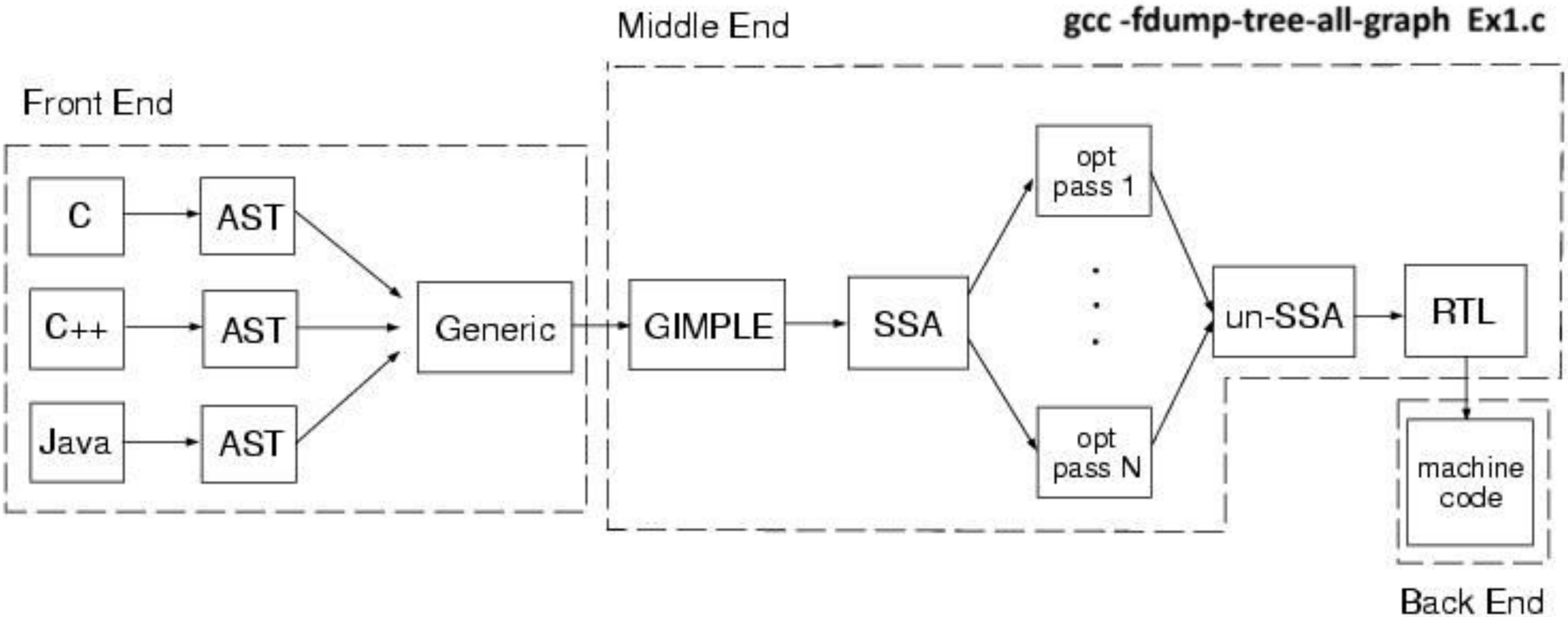
Compiler Design

GCC Compilation Process



Compiler Design

GCC Compiler framework



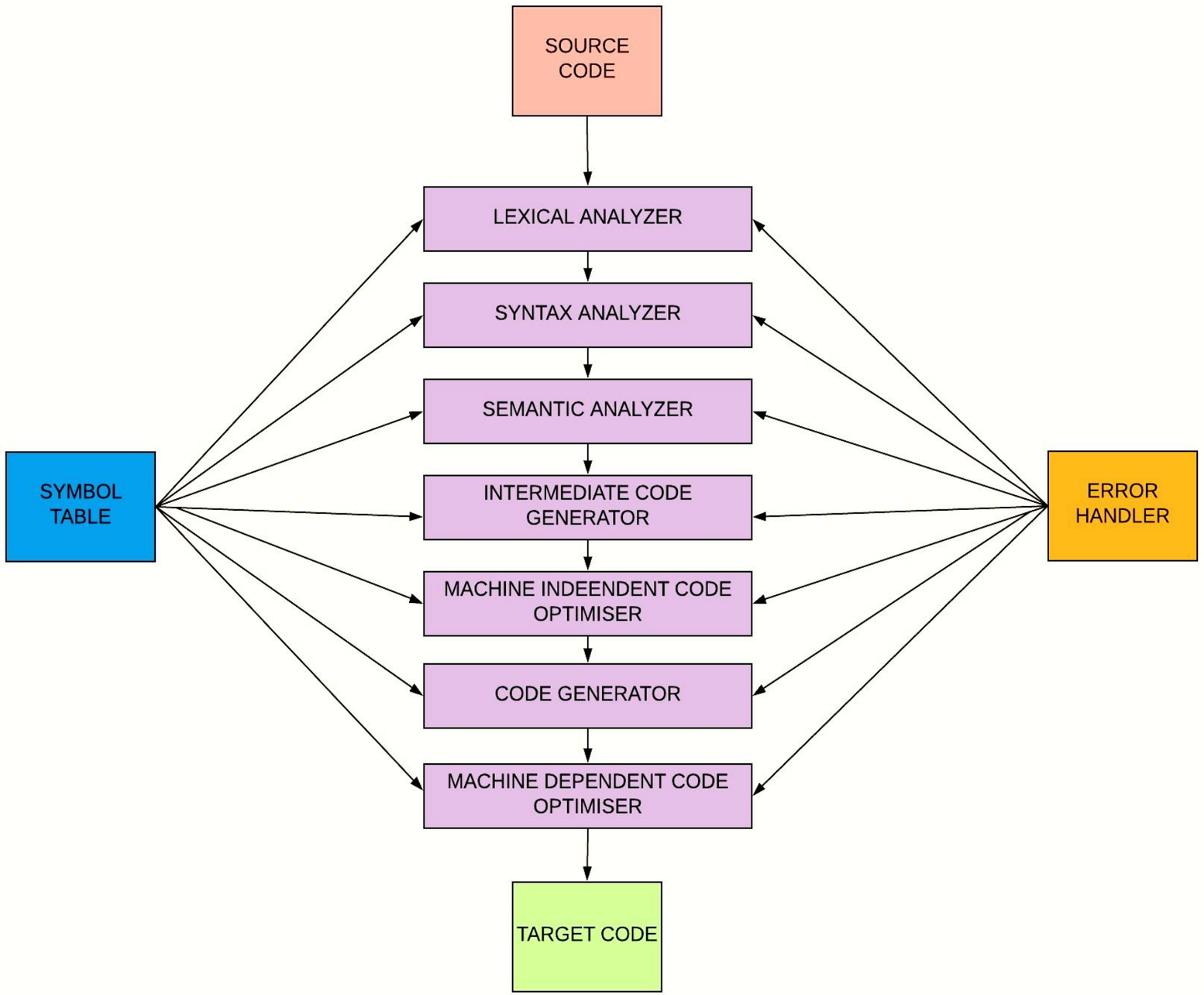
Refer to the following link for further information

http://gcc.gnu.org/onlinedocs/gccint/

Picture taken from grc/iitb

Compiler Design

7 Phases of Compiler



The compiler does its work in seven different phases and each of them have access to something known as "Symbol Table" and to an "Error Handler".

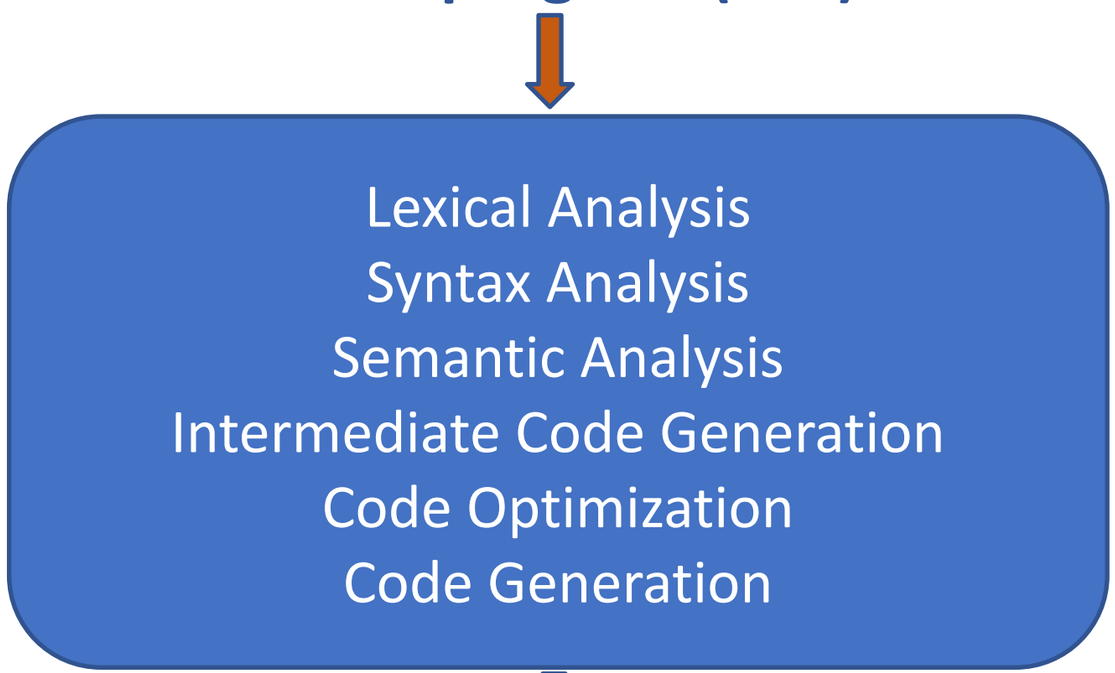
These two entities are explained in the upcoming slides

Compiler Design

Grouping of phases into passes

Single pass Compiler – All phases are grouped into one part.

Source program (HLL)



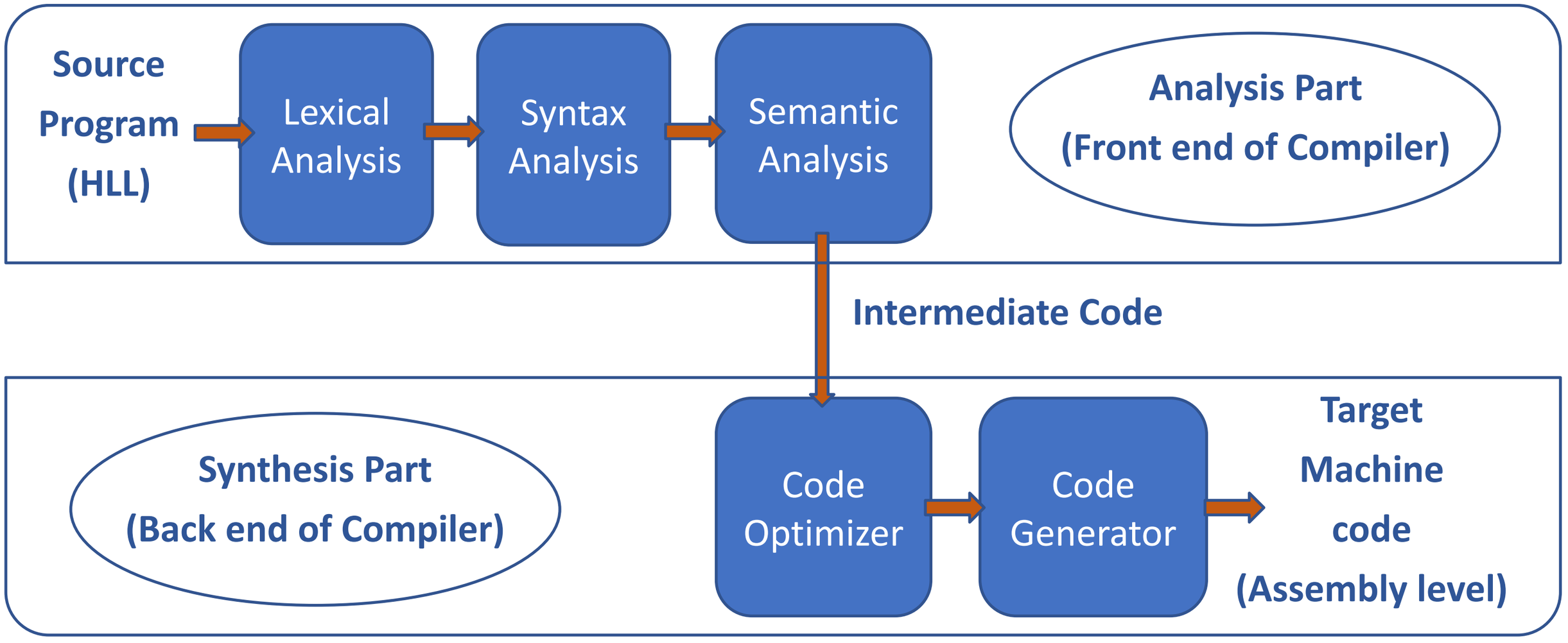
Target Machine code

(Assembly language code)

Compiler Design

Grouping of phases into passes

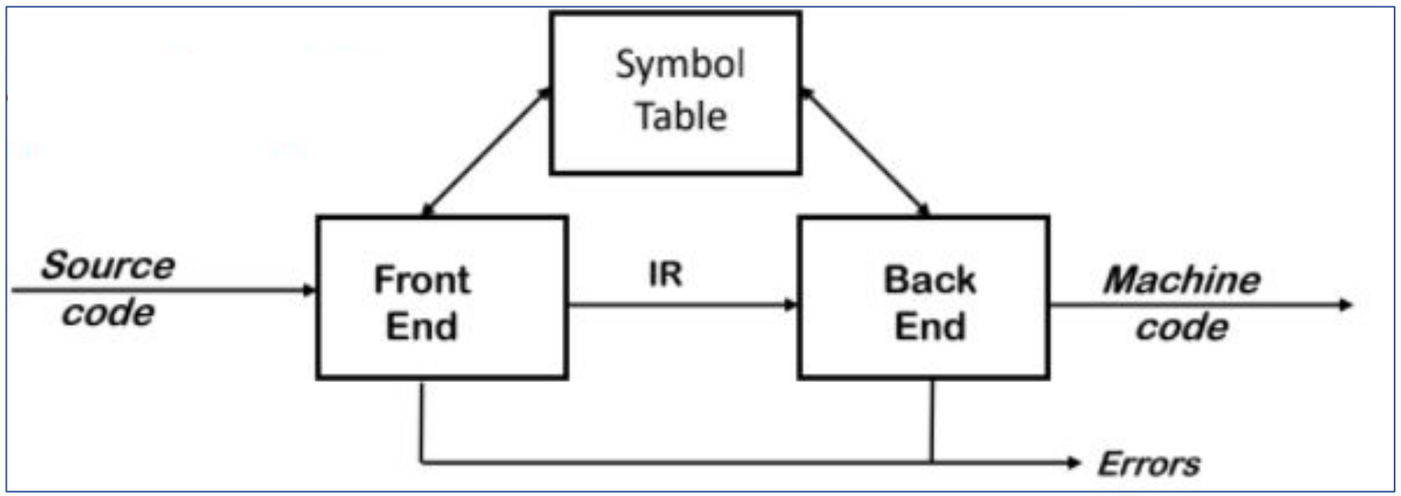
Two pass Compiler- The phases are grouped into two parts.



Compiler Design

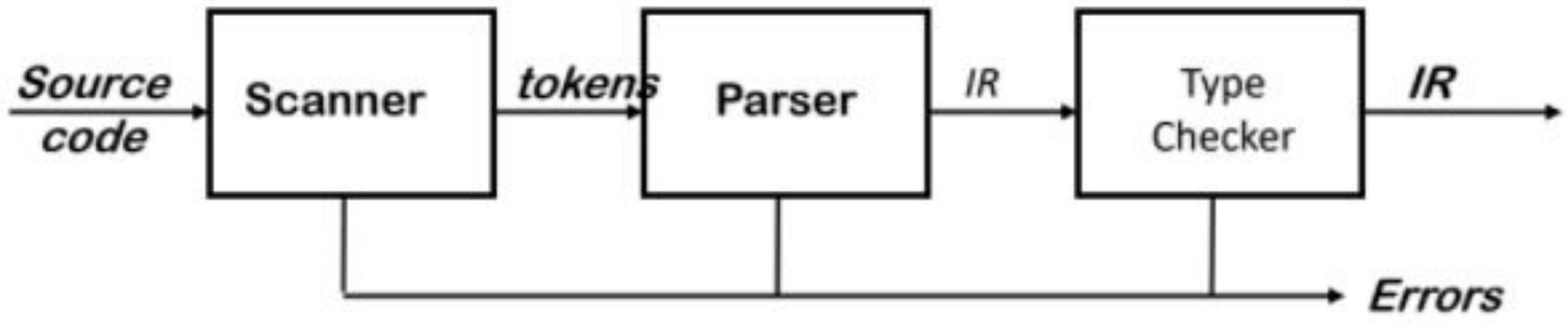
Traditional two-pass compiler

- Use an intermediate representation (IR)
- Front end maps legal source code into IR
- Back end maps IR into target machine code
- Admits multiple front ends and multiple passes
- Typically, front end is O(n) or O(n log n), while back end is NPC
- Different phases of compiler also interact through the symbol table



Compiler Design

The front-end

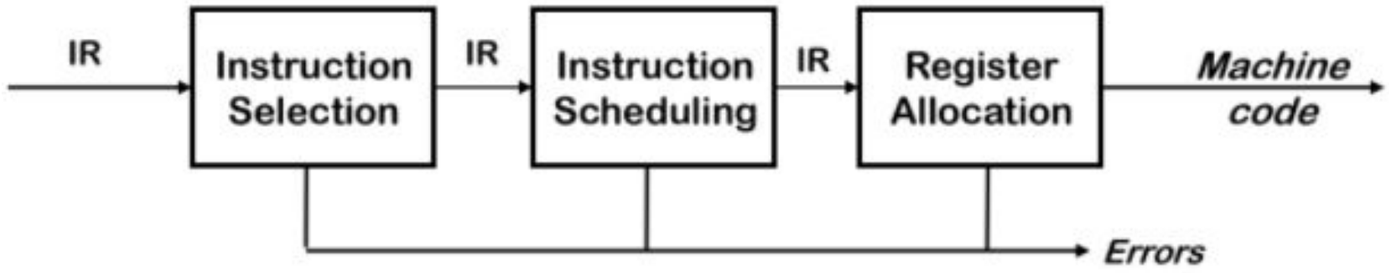


Responsibilities

- Recognize legal programs
- Report errors for the illegal programs in a useful way
- Produce IR and construct the symbol table
- Much of front end construction can be automated

Compiler Design

The back-end



Responsibilities

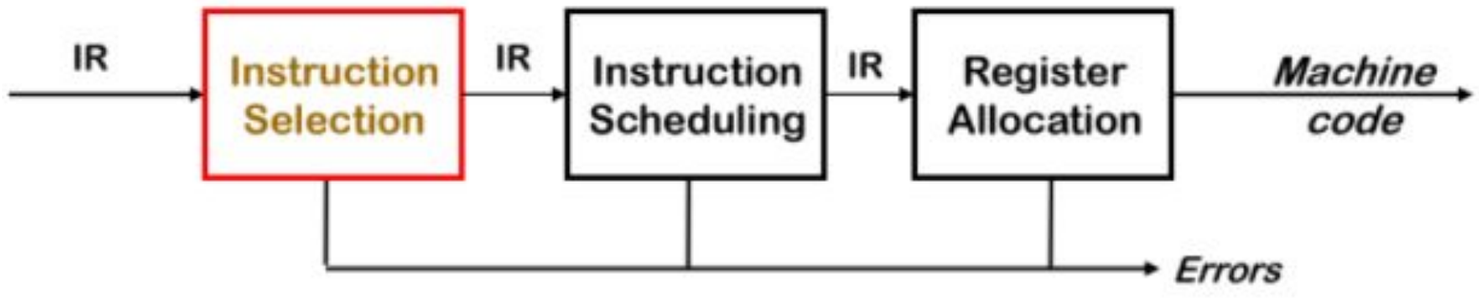
- **Translate IR into target machine code**
- **Choose instructions to implement each IR operation**
- **Decide which values to keep in registers**
- **Schedule the instructions for instruction pipeline**

Automation has been *much less* successful in the back end

We can not run these “IR” files since it is machine independent because the code gets transferred into another form

Compiler Design

The back-end (Instruction Selection)



- Produce fast, compact code
- Take advantage of target language features such as addressing modes
- Usually viewed as a pattern matching problem

This was the problem of the future in late 70's when instruction sets were complex

- RISC architectures simplified this problem

Compiler Design

Code generation: Instruction Selection

Source code

a = b + c;

d = a + e;

If we generate code for each statement separately we will not generate efficient code

Target code

MOV b,R0

ADD c,R0

MOV R0,a

MOV a,R0

ADD e,R0

MOV R0,d

Code for first statement

Redundant

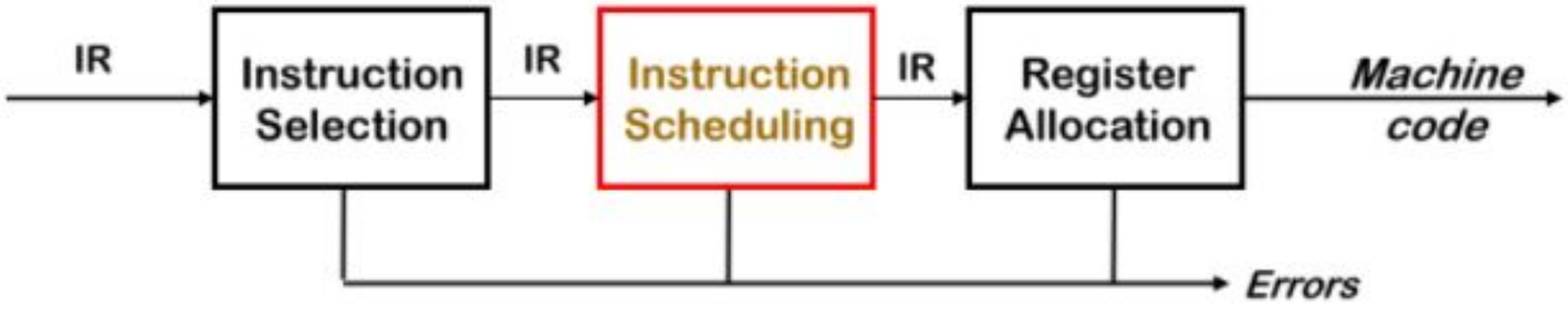
Code for second statement

If c = 1, then:-

MOV b,R0
INC R0
ADD e.R0
MOV R0,d

Compiler Design

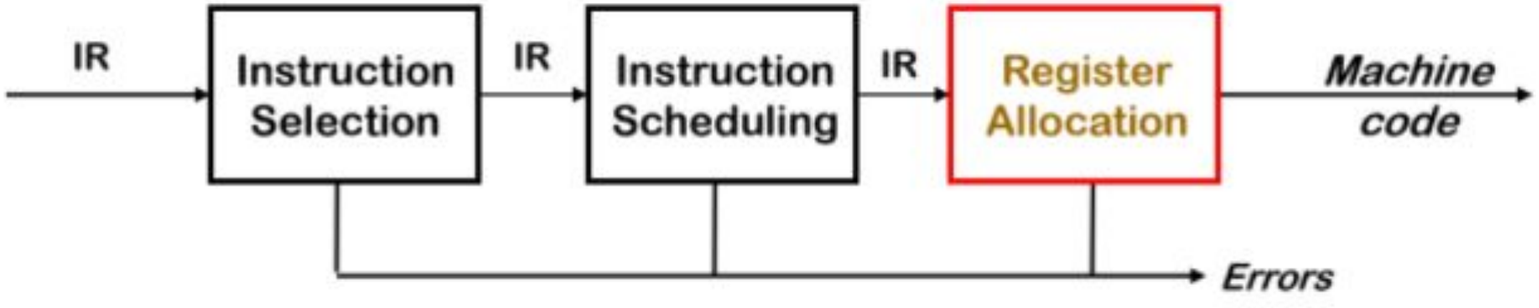
The back-end (Instruction Scheduling)



- Avoid hardware stalls (keep pipeline moving)
- Use all functional units productively
- Optimal scheduling is the focus

Compiler Design

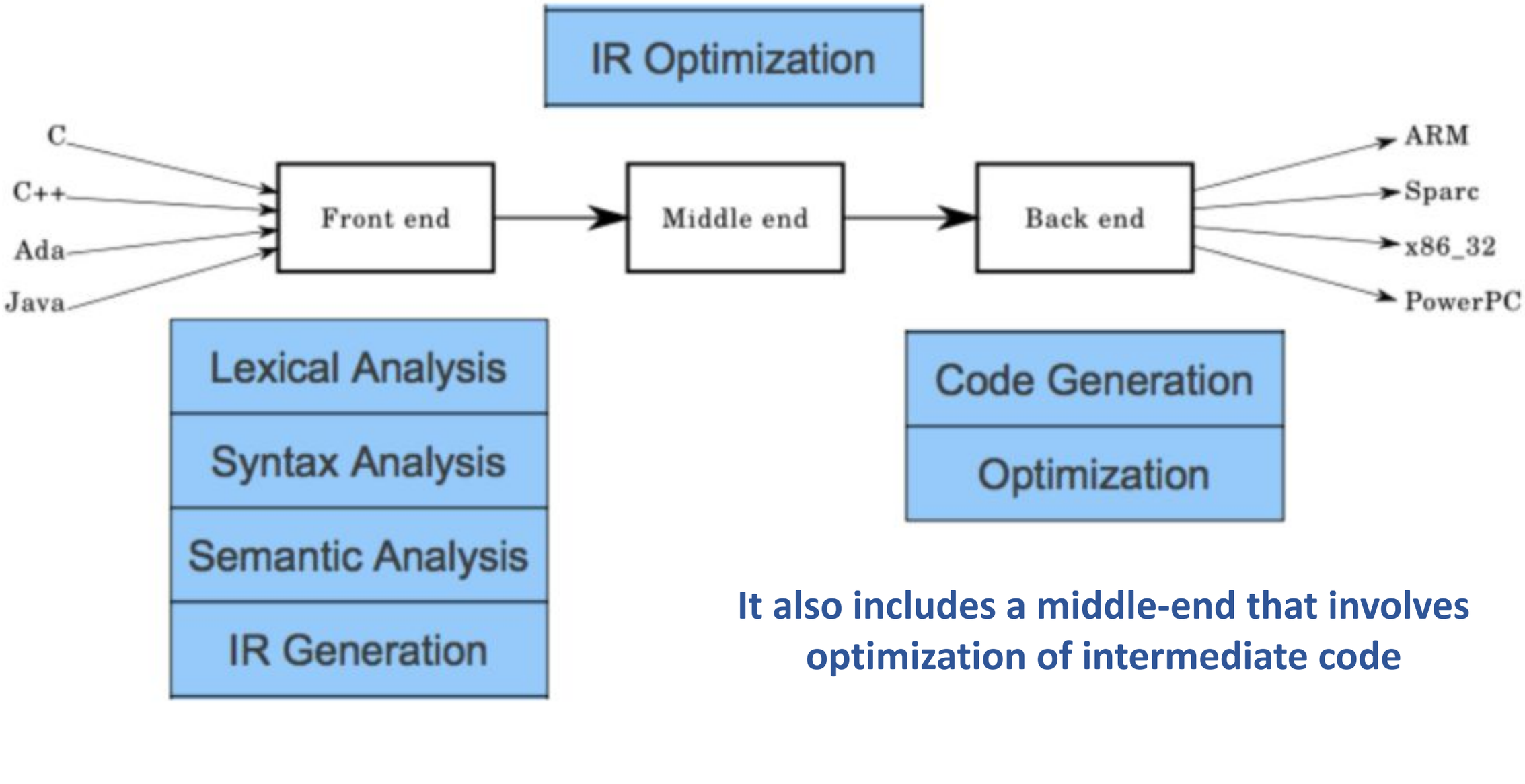
The back-end (Register Allocation)



- Have each value in a register when it is used
- Manage a limited set of registers
- Can change instruction choices and insert LOADs and STOREs
- Optimal allocation is a focus

Compiler Design

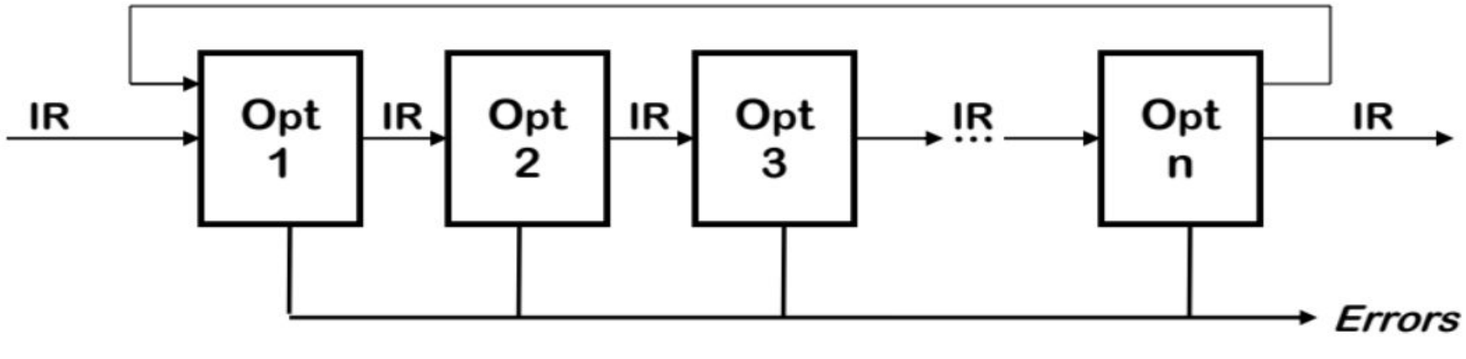
Three-pass compiler



It also includes a middle-end that involves optimization of intermediate code

Compiler Design

The middle-end (Code Optimization)



Modern optimizers are structured as a series of passes

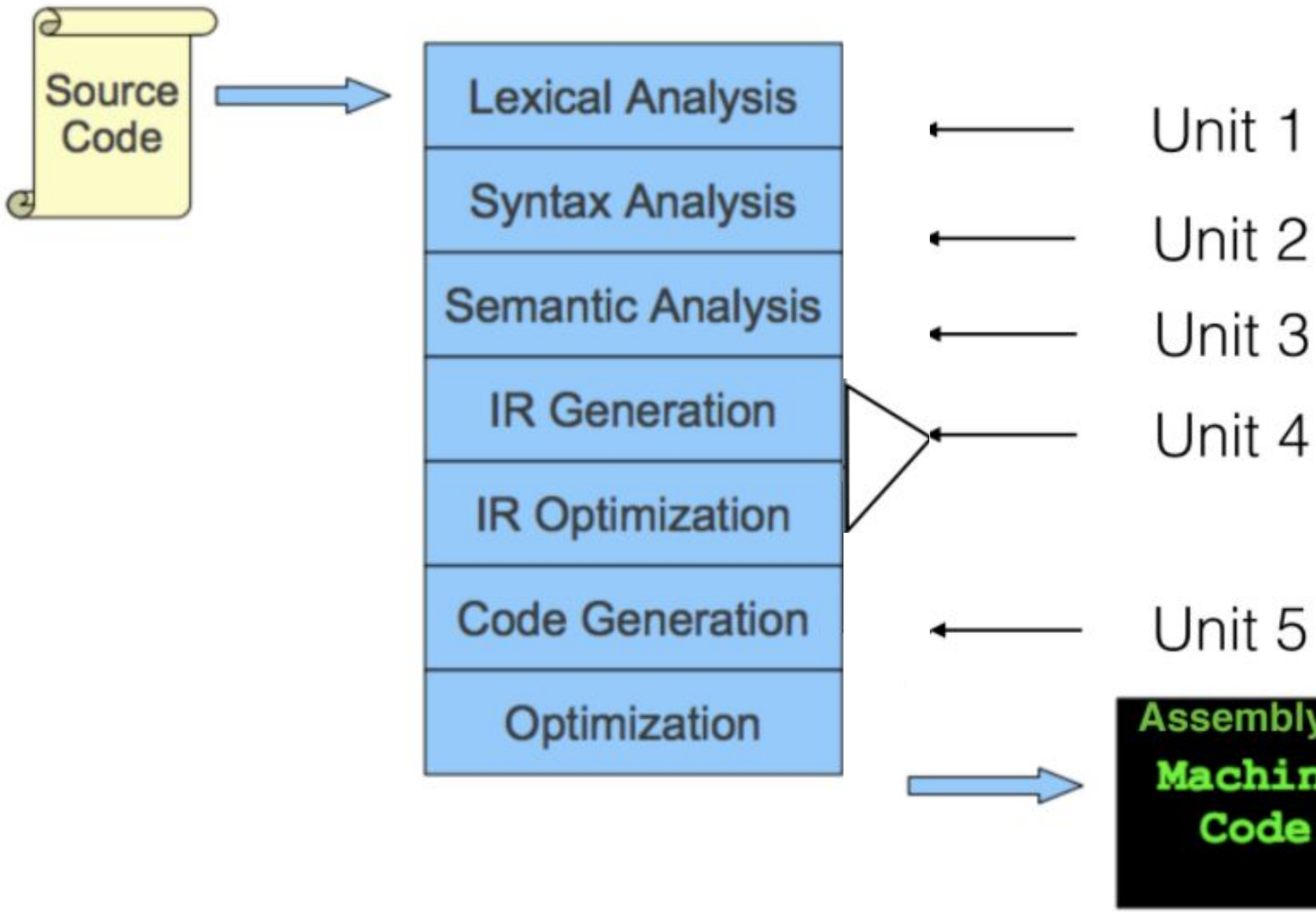
- Analyzes IR and transforms IR
- Primary goal is to reduce running time of the compiled code
- May also improve space, power consumption (mobile computing)
- Must also preserve “meaning” of the code

Typical Transformations:

- Discover and propagate constant values
- Discover a redundant computation and remove it
- Remove unreachable code

Compiler Design

Course Design Course Structure



THANK YOU

Preet Kanwal

Department of Computer Science & Engineering

preetkanwal@pes.edu

PES UNIVERSITY ONLINE

Compiler Design

Preet Kanwal

Department of Computer Science & Engineering

Teaching Assistant : Lavitra Kshitij Madan

Compiler Design

Unit 1

The Phases of a Compiler

Preet Kanwal

Department of Computer Science & Engineering

Compiler Design

Lecture Overview

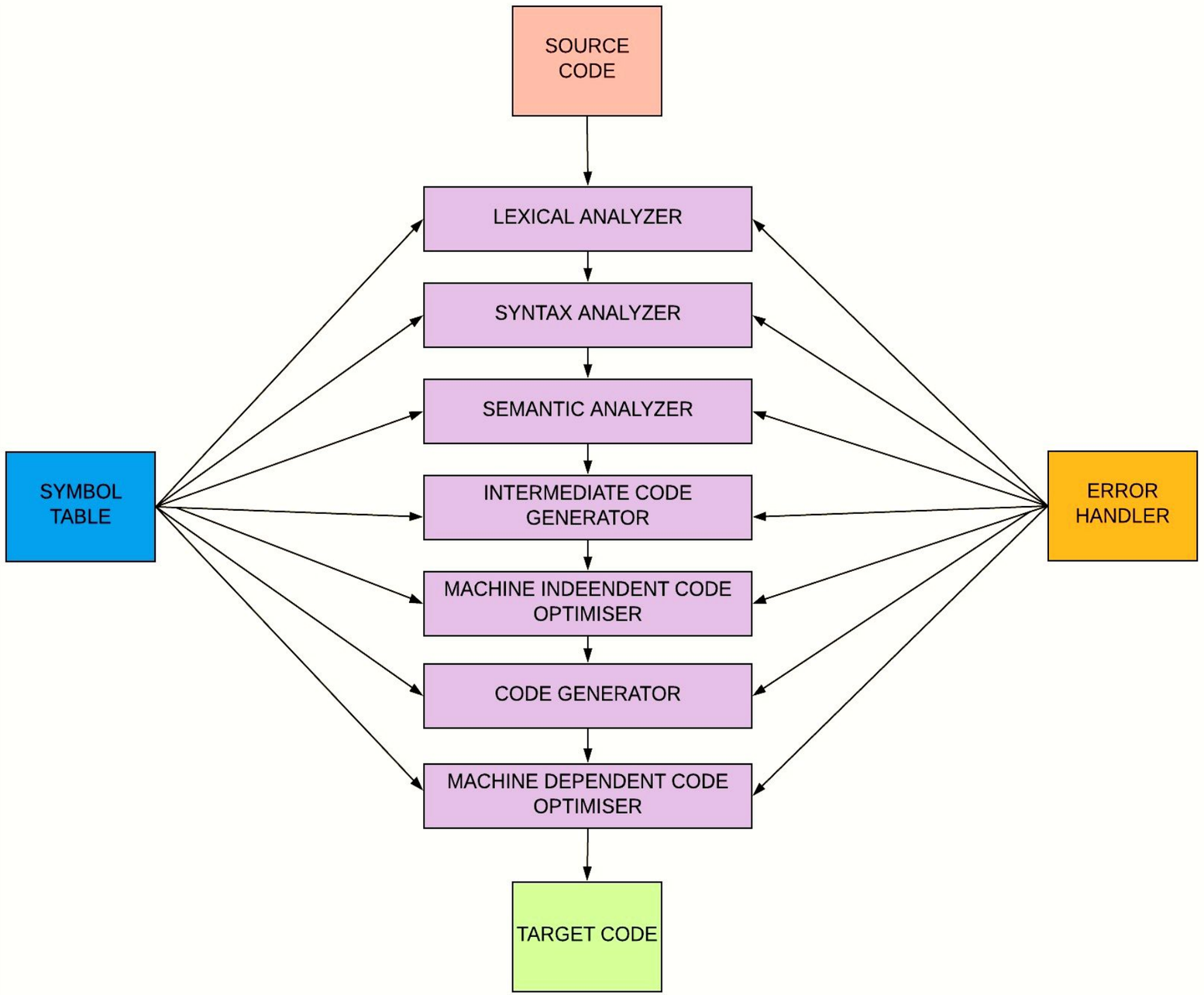
PES UNIVERSITY ONLINE

In this lecture, you will learn about:

- Symbol Table
- Construction of symbol table

Compiler Design

7 Phases of Compiler



The compiler does its work in seven different phases and each of them have access to something known as "Symbol Table" and to an "Error Handler".

These two entities are explained in the upcoming slides

Compiler Design

Symbol Table

What is a symbol table?

Data structure containing a record for each variable name with fields for the attributes of the name

How does it look like?

Attributes provides the information about:

- Storage allocation
- Type
- Scope (where in the program its value may be used)
- Procedure names
  - Number and types of its arguments
  - Method of passing arguments (call by value or reference)

Compiler Design

Symbol Table

PES UNIVERSITY ONLINE

Why symbol table?

- Used during all phases of compilation
- Maintains information about many source language constructs
- Incrementally constructed and expanded during the analysis phases
- Used directly in the code generation phases
- Efficient storage and access important in practice

Symbol Table

When and where is it used?

- Lexical Analysis time
  - Lexical Analyzer scans program
  - Finds Symbols
  - Adds Symbols to symbol table
- Syntactic Analysis Time
  - Information about each symbol is filled in/updated
- Used for type checking during semantic analysis

Symbol Table

More about symbol tables

- Each piece of info associated with a name is called an attribute.
- Attributes are language dependent.
  - Actual Characters of the name
  - Type
  - Storage allocation info (number of bytes).
  - Line number where declared
  - Lines where referenced.
  - Scope.

Symbol Table

Different Classes of Symbols have different attributes

- Variable, Type, Constant, parameter, record field.
  - Type, storage, name, scope.
- Procedure or function.
  - Number of parameters, parameters themselves, result type.
- Array
  - # of Dimensions, Array bounds.
- File
  - record size, record type

PES UNIVERSITY ONLINE

Other attributes:

- A scope of a variable can be represented by
  - A number (Scope is just one of attributes)
  - A different symbol table is constructed for different scope.

- Object Oriented Languages Have classes like
  - Method names, class names, object names.
  - Scoping is VERY important. (Inheritance).

Compiler Design

Constructing the symbol table

There are three main operations to be carried out on the symbol table:

- determining whether a string has already been stored
- inserting an entry for a string
- deleting a string when it goes out of scope

This requires three functions:

- lookup(s): returns the index of the entry for string s, or 0 if there is no entry
- insert(s,t): add a new entry for string s (of token t), and return its index
- delete(s): deletes s from the table (or, typically, hides it)

Compiler Design

Constructing the symbol table

The two symbol table mechanisms:

Linear lists and Hash tables

- Each scheme is evaluated on the basis of time required to add n entries and make e inquiries.

- A linear list is the simplest to implement, but its performance is poor when n and e are large.

- Hashing schemes provide better performance for greater programming effort and space overhead.

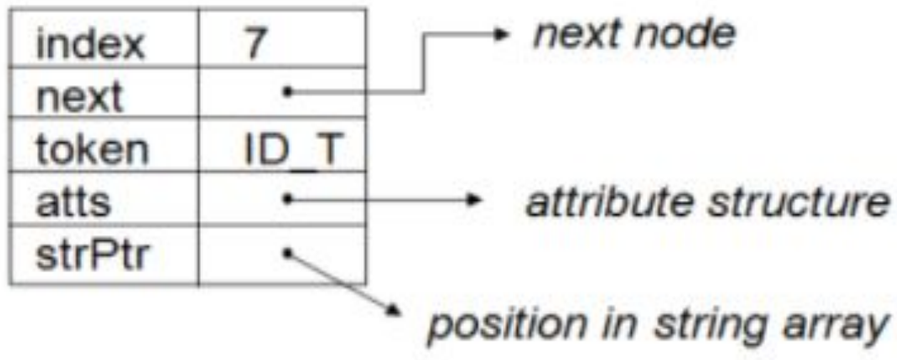
Compiler Design

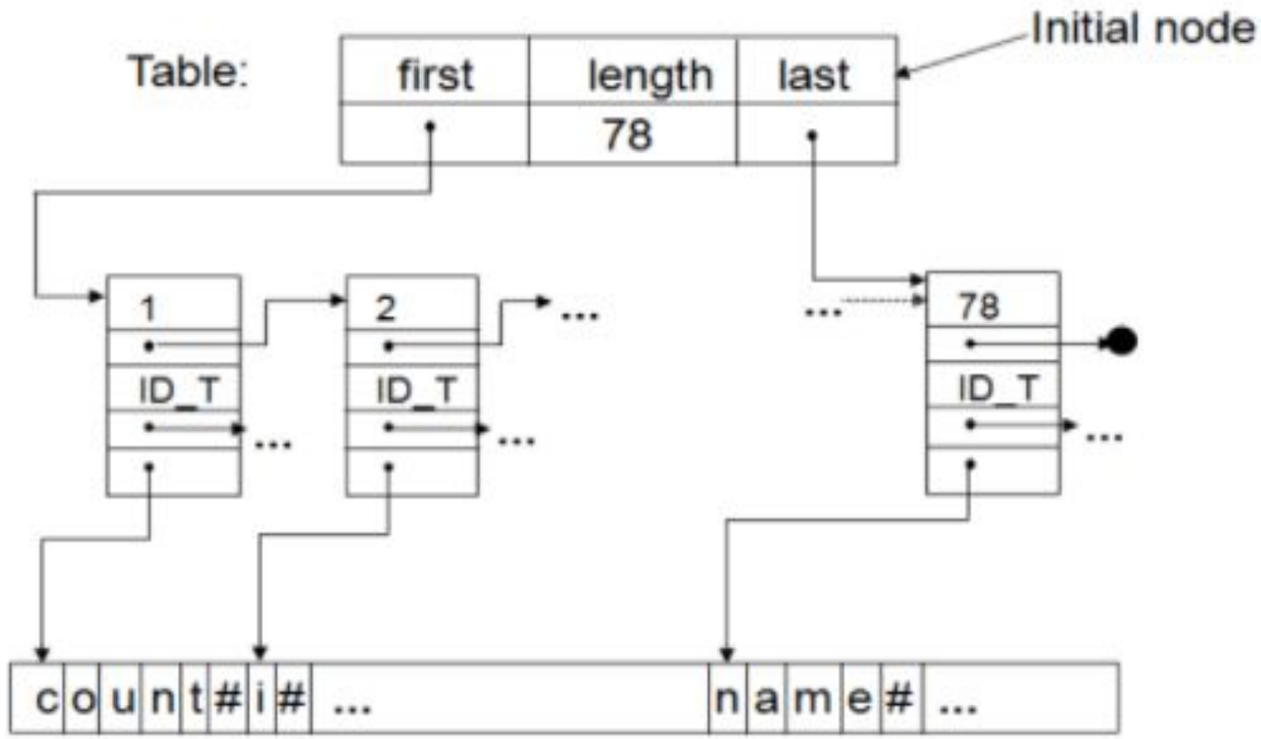
Constructing the symbol table (Example)

1. float fun2(int I, float j,){
2. int k,e;
3. float z;
4. ;
5. e=0;
6. k = I *j+k;
7. z = fun1(k,e,);
8. *z; }

Name	Token	Dtype	Value	Size	Scope	Other Attribute		
						Declared	Referred	Other
Fun2	TK_ID	procname				1		
I	TK_ID	Int		4	1	1	6	parameter
j	TK_ID	Int		4	1	1	6	parameter
k	TK_ID	Int		4	0	2	6,7	argument
e	TK_ID	Int	0	4	0	2	7	argument
z	TK_ID	Float		4	0	3	7,8	return
fun1	TK_ID	procname				7		proccall

Constructing the symbol table (Linked list implementation)





Compiler Design

Symbol table and scope

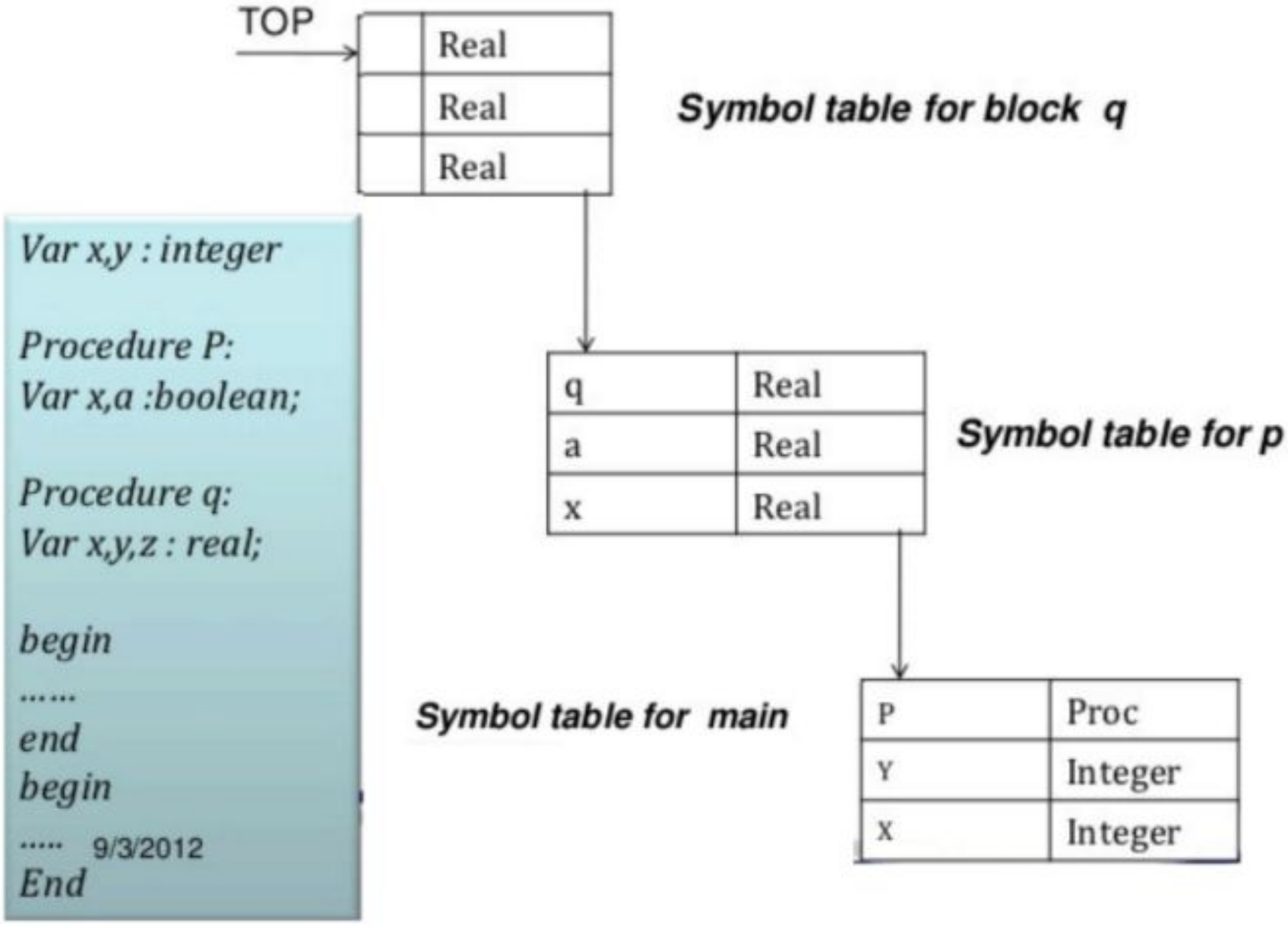
- **Symbol tables typically need to support multiple declarations of the same identifier within a program**

- **We shall implement scopes by setting up a separate symbol table for each scope**

The scope of a declaration is the portion of a program to which the declaration applies.

Compiler Design

Symbol table organization



THANK YOU

Preet Kanwal

Department of Computer Science & Engineering

preetkanwal@pes.edu

Contents
Generating Abstract Syntax Tree
Generating Intermediate Code : TAC, SSA
Difference between Abstract Syntax tree and Parse tree
Constructing Syntax tree for Different statements
Symbol-Table

Generating Abstract Syntax tree:

input file name : <test.c>

```
$gcc -fdump-tree-original-raw test.c
```

output: test.c.003t.original

```
$awk -f pre.awk test.c.003t.original |awk -f treeviz.awk > tree.dot
```

pre.awk and treeviz.awk transforms the file according to the dot format<graphviz package>
dot : DOcument Template file

```
$ gedit pre.awk
```

```
#! /usr/bin/gawk -f
/^[^;]/{
gsub(/^@/, "~@", $0);
gsub(/( *):( *)/, ":", $0);
print;
}
```

```
$gedit treeviz.awk
```

```
#! /usr/bin/gawk -f
BEGIN {RS = "~@"; printf "digraph G {\n node [shape = record];";}
/^[0-9]/{
s = sprintf("\n%s [label = \"{%s | {", $1, $1);
for(i = 2; i < NF; i++)
s = s sprintf("%s | ", $i);
s = s sprintf("%s}}\"];", $i);
$0 = s;
while (/([a-zA-Z0-9]+):@([0-9]+)/){
format = sprintf("\\1 \\3\n %s:\\1 -> \\2;", $1);
$0 = gensub(/([a-zA-Z0-9]+):@([0-9]+)(.*)$/, format, "g");
};
printf " %s", $0;
}
END {print "\n}"}
```

```
$dot -Tpng tree.dot > tree.png
```

Generating Intermediate Code:

3 forms : generic, gimple, rtl(register transfer language)

Intermediate code formats :
a) Three address code (gimple file)
b) SSA : static single assignment (ssa file)

```
$gcc -fdump-tree-all-graph test.c
$ls
```

a.out	test.c.007t.lower	test.c.015t.ssa.dot	test.c.036t.release_ssa	test.c.162t.cplxlower0.dot
test.c	test.c.007t.lower.dot	test.c.017t.inline_param1	test.c.036t.release_ssa.dot	test.c.169t.optimized
test.c.001t.tu	test.c.010t.eh	test.c.017t.inline_param1.dot	test.c.037t.inline_param2	test.c.169t.optimized.dot
test.c.003t.original	test.c.010t.eh.dot	test.c.018t.einline	test.c.037t.inline_param2.dot	test.c.249t.statistics
test.c.004t.gimple	test.c.011t.cfg	test.c.018t.einline.dot	test.c.161t.veclower	test.c.249t.statistics.dot
test.c.006t.omplower	test.c.011t.cfg.dot	test.c.033t.profile_estimate	test.c.161t.veclower.dot	
test.c.006t.omplower.dot	test.c.015t.ssa	test.c.033t.profile_estimate.dot	test.c.162t.cplxlower0	

test.c.004t.gimple : contains Three-address code
test.c.015t.ssa : contains SSA code

Three-address Code :

- **Three-address code** (often abbreviated to TAC or 3AC) is an intermediate **code** used by optimizing compilers to aid in the implementation of **code**-improving transformations.
- Each TAC instruction has at most **three** operands and one operation (is typically a combination of assignment and a binary operator).

Example:
x = a + y * 10

TAC :
t1 = y * 10
t2 = a + t1
x = t2

SSA – Single Static Assignment Form:

- In compiler design, **static single assignment form** (often abbreviated as **SSA form** or simply **SSA**) is a property of an intermediate representation (IR), which requires that each variable is assigned exactly once, and every variable is defined before it is used.
- Compiler optimization algorithms which are either enabled or strongly enhanced by the use of SSA.
- Converting ordinary code into SSA form is primarily a simple matter of replacing the target of each assignment with a new variable, and replacing each use of a variable with the "version" of the variable reaching that point.

Example :

y := 1
y := 2
x := y

SSA form:

$y_1 := 1$
$y_2 := 2$
$x_1 := y_2$

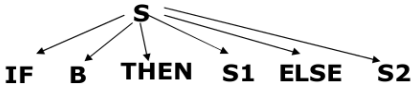
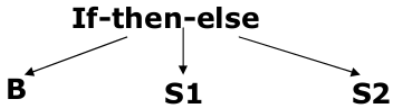
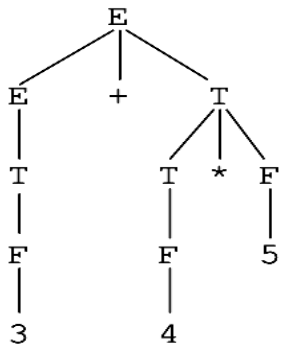
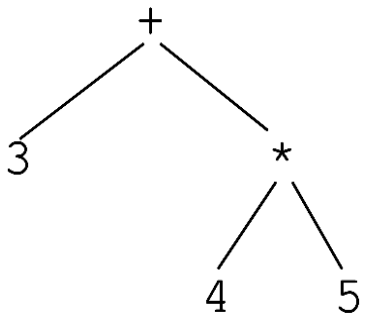
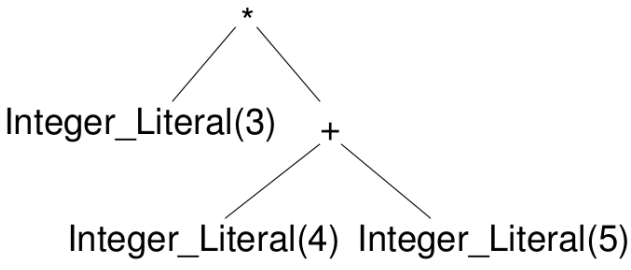
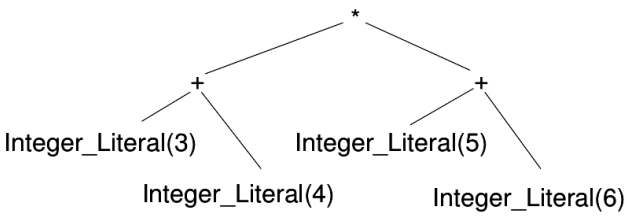
Displaying Control Flow graph:

```
$dot -Tpng test.c.015t.ssa.dot > ssa.png
```

Control flow graph : is obtained by dividing the entire program into multiple blocks. Blocks are constructed in such a way that a control can enter the block only from the first instruction of the block and leave the block from the last instruction of the block only.

- The Structure is desired for optimizing the code. The optimized code is available in test.c.169t.optimized .
- Optimization : local (within a block) or global(for entire program).
- One of the ways of performing local optimization is : DAG optimization.

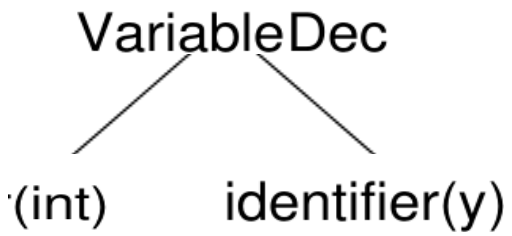
Difference between Abstract Syntax tree and Parse tree

Parse Tree (Concrete Syntax Tree)	Abstract Syntax Tree
• Concrete syntax: what the programmer wrote • tree representation of grammar derivation	• Abstract syntax: what the compiler needs condensed form of parse tree. • Abstract tree has less information than the concrete tree. • Operators and keywords do not appear as leaves • Chains of single productions are collapsed
	
E → E + T E → T T → T ∗ F T → F F → num | id | (E)  Parse tree for 3 + 4 * 5 3 * (4 + 5) (3 + 4) * (5 + 6) (((4)))	 Abstract Syntax Tree for 3 + 4 * 5 What about parentheses? Do we need to store them? • Parenthesis information is store in the shape of the tree • No extra information is necessary   Integer_Literal(4)

Constructing Syntax tree for Different statements

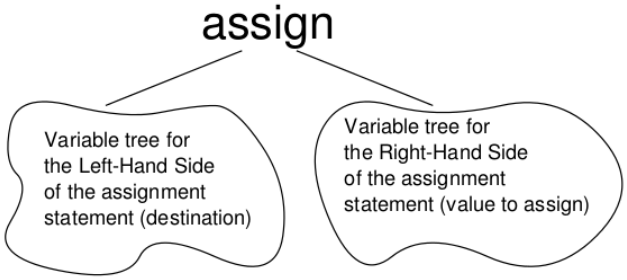
Decl -> Type VarList
Type -> int|float|char
VarList -> VarList, id | id

AST – Variable Declaration
int y;



AssignExpr -> id = E
E → E + T
E → T
T → T ∗ F
T → F
F → num | id | (E)

AST – assignment Statements



Sample CFG
S -> if(cond){S} | if(cond){S}else{S}
| while(cond){S}
| for(AssignExpr cond AssignExpr){S}
| AssignExpr
| S; S
| epsilon
| UnaryExpr
| Decl
| break ; | continue ;
| return <expr> ;
| goto <id> ; | epsilon

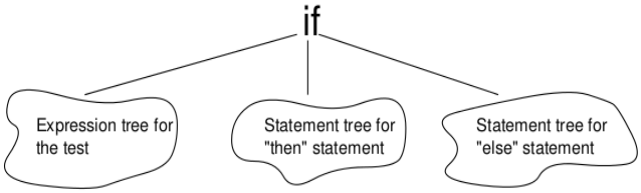
cond -> expr | expr logOp expr
expr -> relexp | logexp | E
relexp -> E relOp E
logexp -> E logOp E
logOp -> || | &&
relOp -> < | > | <= | >= | != | ==

AssignExpr -> id = E;
E -> E + T | T
T -> T * F | F
F -> id | num | (E)

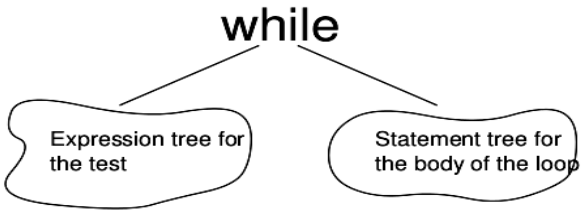
UnaryExpr -> ++ UE | UE++ | --UE | UE--
UE -> id | (E)

Decl -> Type VarList;
Type -> int|float|char
VarList -> VarList, id | id

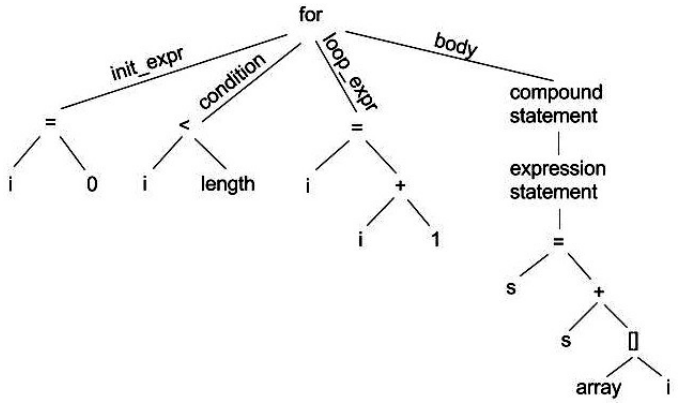
If Statement



While statement



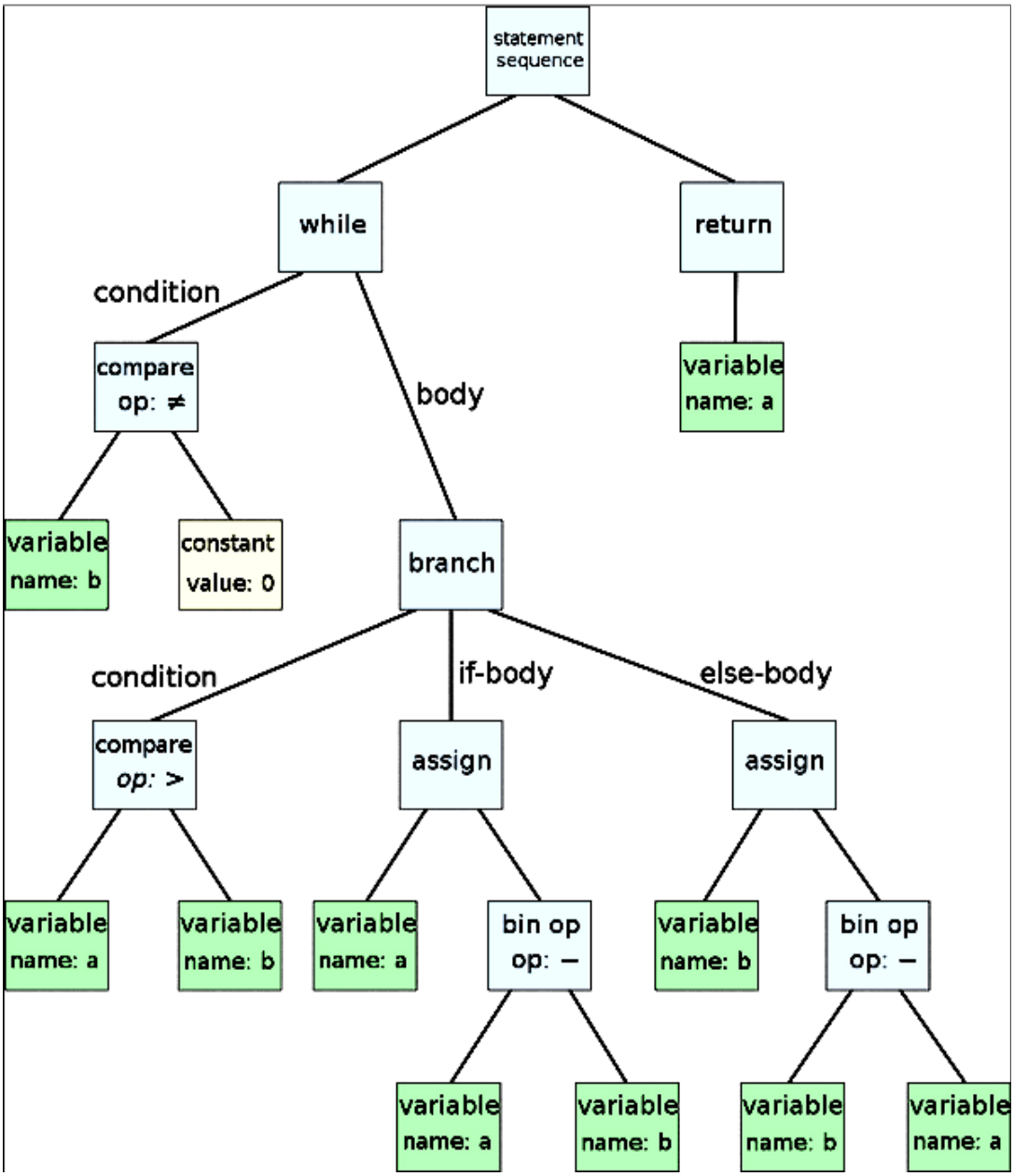
For Statement



Example

An abstract syntax tree for the following code for the Euclidean algorithm:

```
while b != 0
    if a > b
        a = a − b
    else
        b = b − a
return a
```



Symbol Table

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

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

- To store the names of all entities in a structured form at one place.
- To verify if a variable has been declared.
- To implement type checking, by verifying assignments and expressions in the source code are semantically correct.
- To determine the scope of a name (scope resolution).

A symbol table is simply a table which can be either linear or a hash table.

Symbol Table Requirements

1. **Fast lookup**. The parser of a compiler is linear in speed. The table lookup needs to be as fast as possible.
2. **Flexible in structure.** The entries must contain all necessary information, depending on usage of identifier
3. **Efficient use of space**. This will require runtime allocation (dynamic) of the space.
4. **Handle characteristics of language** (e.g., scoping, implicit declaration)
    - Scoping requires handling entry of a local block and exit.
    - Block exit requires removal or hiding of the entries of the block.

Symbol Table Operations

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

insert()

This operation is more frequently used by analysis phase, i.e., the first half of the compiler where tokens are identified and names are stored in the table. This operation is used to add information in the symbol table about unique names occurring in the source code. The format or structure in which the names are stored depends upon the compiler in hand.

An attribute for a symbol in the source code is the information associated with that symbol. This information contains the value, state, scope, and type about the symbol. The insert() function takes the symbol and its attributes as arguments and stores the information in the symbol table.

lookup()

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

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

Scope Management

A compiler maintains two types of symbol tables: a **global symbol table** which can be accessed by all the procedures and **scope symbol tables** that are created for each scope in the program.

Who Creates Symbol-Table Entries?

It is the semantic analysis phase which creates the symbol table because it is not until the semantic analysis phase that enough information is known about a name to describe it.

Many compilers set up a table at lexical analysis time for the various variables in the program, and fill in information about the symbol later during semantic analysis when more information about the variable is known.

With its knowledge of the syntactic structure of a program, a parser is often in a better position than the lexical analyzer to distinguish among different declarations of an identifier. In some cases, a lexical analyzer can create a symbol-table entry as soon as it sees the characters t h a t make up a lexeme. More often, the lexical analyzer can only return to the parser a token, say id, along with a pointer to the lexeme. Only the parser, however, can decide whether to use a previously created symbol-table entry or create a new one for the identifier.

PES UNIVERSITY ONLINE

Compiler Design

Preet Kanwal

Department of Computer Science & Engineering

Teaching Assistant : Lavitra Kshitij Madan

Compiler Design

Unit 1

The Phases of a Compiler

Preet Kanwal

Department of Computer Science & Engineering

Compiler Design

Lecture Overview

PES UNIVERSITY ONLINE

In this lecture, you will learn about:

- Challenges in scanning
- C Language Grammar

Compiler Design

Syntactic Sugar

Syntactic Sugar : Syntax is designed to make things easier to read or express.

Example:

- **In C : a[i] is a syntactic sugar for *(a+i)**
- **In C : a+=b is equivalent to a = a + b**
- **In C# : var x = expr (compiler deduces type of x from expr)**

Compilers expand sugared constructs into fundamental constructs (Desugaring).

Compiler Design

Challenge : Scanning is hard

Example 1: In FORTRAN, whitespace is irrelevant. For eg:

```
DO 5 I = 1,25
DO5I   = 1.25
```

This makes it difficult to decide where to partition the input.

Challenge : Scanning is hard

Example 2 : Different uses of the same character >,< in C++

Template syntax: a<b>

Stream syntax: cin>>var;

Binary right shift syntax: a>>4

Nested Template : A<B<C>>D;

Challenge : Scanning is hard

Example 3: When keywords are used as identifiers.

```
IF THEN THEN THEN = ELSE; ELSE ELSE = IF
```

This makes it difficult to name lexemes.

Lexer Feedback

C and C++ Lexers require lexical feedback to differentiate between typedef names and identifiers.

```
int foo;
typedef int foo;
foo x;
```

Challenge : Scanning is hard

Example 4: Scanning in Python - When Scope is handled using whitespace.

This requires whitespace tokens - Special tokens inserted to indicate changes in levels of indentation.

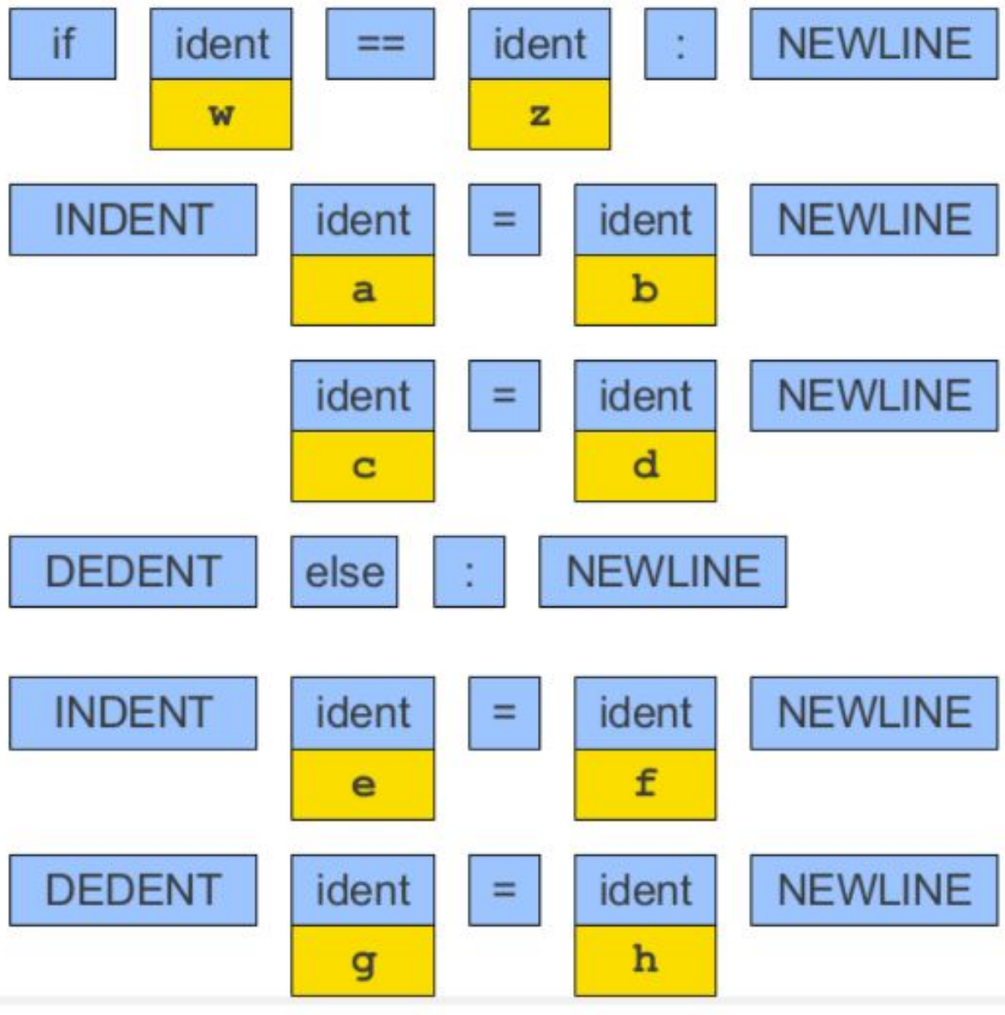
- **NEWLINE marks the end of a line.**
- **INDENT indicates an increase in indentation.**
- **DEDENT indicates a decrease in indentation.**
- **Note that INDENT and DEDENT encode change in indentation, not the total amount of indentation.**

Reference:
https://docs.python.org/3/reference/lexical_analysis.html

Challenge : Scanning is hard

Example 4: Scanning in python (cont.)

```
if w == z:
    a = b
    c = d
else:
    e = f
g = h
```



Compiler Design

C Language Grammar

To define the Grammar for C Language, let us first define the following:

P	**:**	**Program Beginning**
S	**:**	**Statement**
Declr	**:**	**Declaration**
Assign	**:**	**Assignment**
Cond	**:**	**Condition**
UnaryExpr	**:**	**Unary Expression**
Type	**:**	**Data type**
ListVar	**:**	**List of variables**
X	**:**	**(can take any identifier or assignment)**
RelOp	**:**	**Relational Operator**

Compiler Design

C Language Grammar

P → S

S → Declr; S | Assign; S | if (Cond) {S} S | while (Cond) {S} S |
if (Cond) {S} else {S} S | for (Assign; Cond; UnaryExpr) {S} S |
return E; S | λ

Declr → Type ListVar

Type → int | float

ListVar → X | ListVar, X

X → id | Assign

Assign → id = E

Cond → E RelOp E

RelOp → < | > | <= | >= | == | !=

UnaryExpr → E++ | ++E | E-- | --E

Recall that
E -> E + T | E – T | T
T -> T * F | T / F | F
F -> G ^ F | G
G -> (E) | id | num

THANK YOU

Preet Kanwal

Department of Computer Science & Engineering

preetkanwal@pes.edu

PES UNIVERSITY ONLINE

Compiler Design

Preet Kanwal

Department of Computer Science & Engineering

Teaching Assistant : Lavitra Kshitij Madan

Compiler Design

Unit 1

The Phases of a Compiler

Preet Kanwal

Department of Computer Science & Engineering

Compiler Design

Lecture Overview

PES UNIVERSITY ONLINE

In this lecture, you will learn about:

- C Language Grammar
- Running an input through the different phases of a compiler
- Syntax Tree versus Parse Tree

Compiler Design

C Language Grammar

P → S

S → Declr; S | Assign; S | if (Cond) {S} S | while (Cond) {S} S |
if (Cond) {S} else {S} S | for (Assign; Cond; UnaryExpr) {S} S |
return E; S | λ

Declr → Type ListVar

Type → int | float

ListVar → X | ListVar, X

X → id | Assign

Assign → id = E

Cond → E RelOp E

RelOp → < | > | <= | >= | == | !=

UnaryExpr → E++ | ++E | E-- | --E

Recall that
E -> E + T | E – T | T
T -> T * F | T / F | F
F -> G ^ F | G
G -> (E) | id | num

Running an input through the different phases of a compiler

Example 1 : Consider the following piece of code:

```
while (number > 0)
{
    factorial = factorial * number;
    --number;
}
```

We will now run this code through the different phases of a compiler and see what the output will be at each stage.

Compiler Design

Example 1 (continued)

Phase 1 : Lexical Analysis or Linear Analysis or Scanner

Pattern	Lexemes matched
Keyword	while
Identifier	number, factorial
RelOp (<, <=, >, >=, ==, !=)	>
LogOp (&&, ||, ~)	
Assign (=)	=
ArithOp (*, +, -, /)	*
Punctuation ((,), {, }, [,], ;)	( ) { ; ; }
Number	0
Literal – anything in double quotes	
Inc_op	
Dec_op	--

```
while (number > 0)
{
factorial = factorial * number;
--number;
}
```

Compiler Design

Example 1 (continued)

Output: # of tokens : 17

< keyword, while >
< punctuation, (> or < (>
< ID, 3 >
< RelOp, > > or < > >
< Number, 0 >
< punctuation,) > or <) >
< punctuation, { > or < { >
< ID, 4 >
< Assign, = > or < = >
< ID, 4 >
< ArithOp, * > or < * >
< ID, 3 >
< punctuation, ; > or < ; >
< Dec_op, -- > or < -- >
< ID, 3 >
< punctuation, ; > or < ; >
< punctuation, } > or < } >

Symbol Table

	name	type	…
1	…		
2	….		
3	factorial		
4	number		

< ID, 3 > corresponds to the third identifier in the symbol table

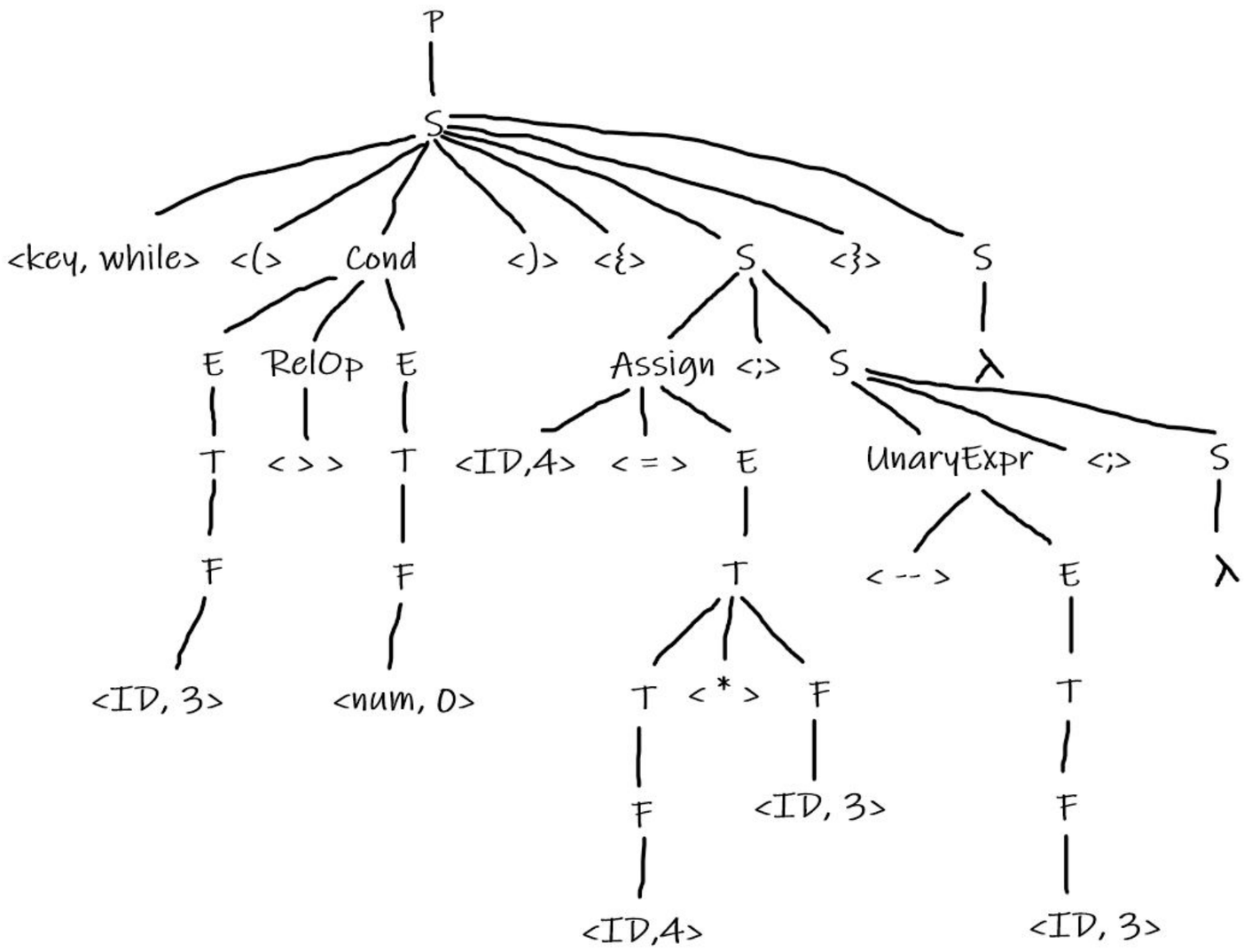
```
while (number > 0)
{
factorial = factorial * number;
--number;
}
```

Compiler Design

Example 1 (continued)

Phase 2 : Syntax Analysis or Parser

Output: Parse Tree or Syntax Tree



Use the grammar we have defined earlier

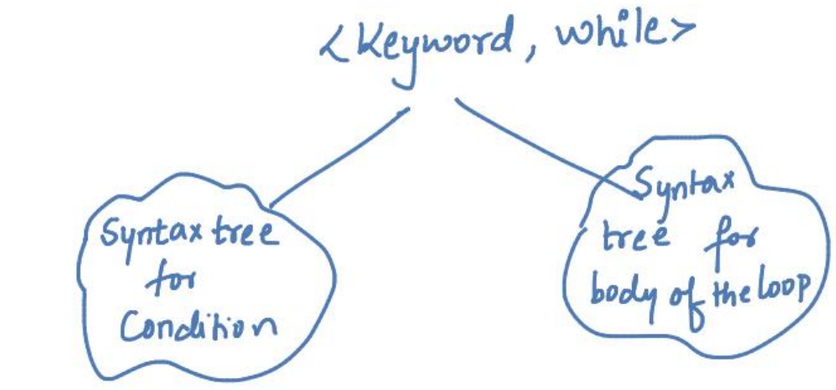
```
while (number > 0)
{
factorial = factorial * number;
--number;
}
```

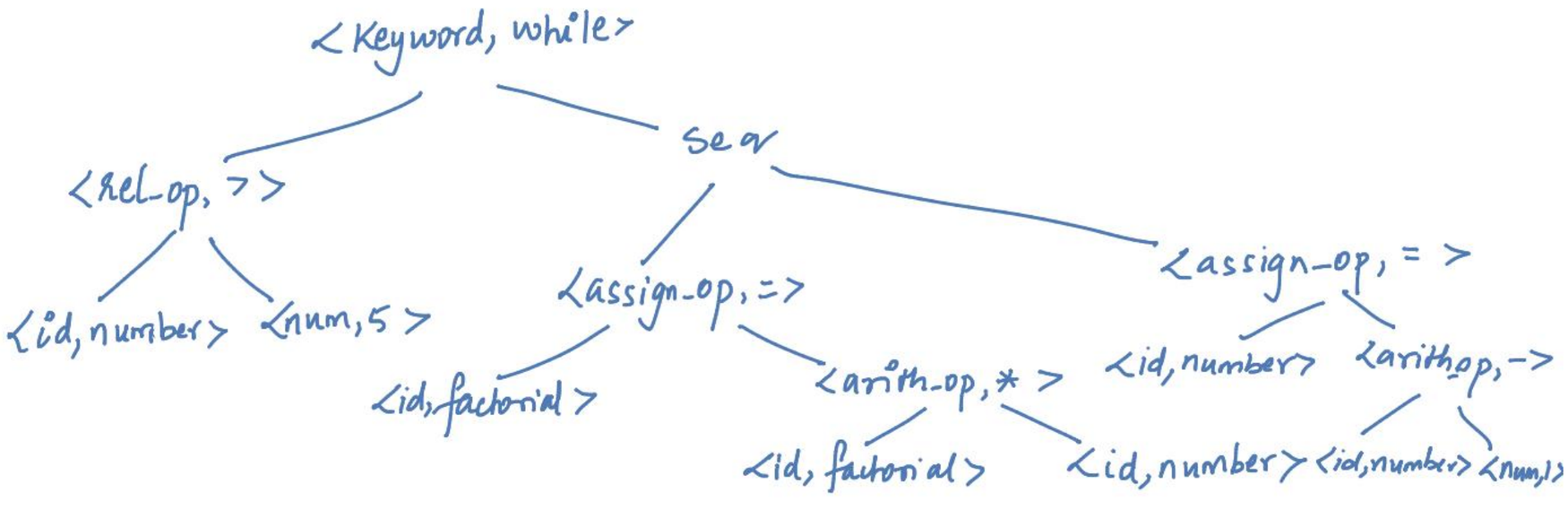
Compiler Design

Example 1 (continued)

Phase 3 : Semantic Analyzer

Output: Semantically checked Syntax Tree






```
while (number > 0)
{
factorial = factorial * number;
--number;
}
```

Compiler Design

Example 1 (continued)

Phase 4 : Intermediate Code Generator

Output: Three Address Code (3AC or TAC)

```
L0 : If number > 0 goto L1
goto L2
L1 : t1 = factorial * number
factorial = t1
t2 = number – 1
number = t2
goto L0
L2 : …
```

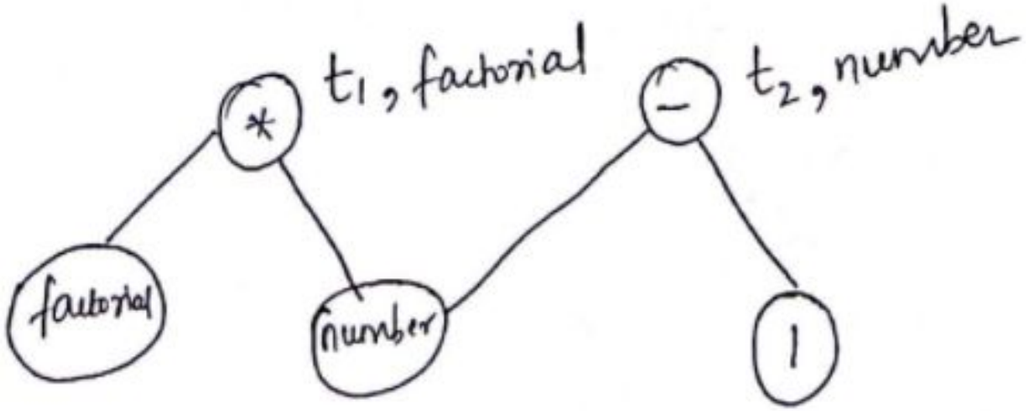
```
while (number > 0)
{
factorial = factorial * number;
--number;
}
```

Compiler Design

Example 1 (continued)

Phase 5 : Machine Independent Code Optimization

Output: DAG Optimization




```
L0 : If number > 0 goto L1

goto L2

L1 : factorial = factorial * number

number = number – 1

goto L0

L2 : …
```

```
while (number > 0)

{

factorial = factorial * number;

--number;

}
```

Compiler Design

Example 1 (continued)

Phase 6 : Code Generator

Output: Assembly Code

```
LD R1, #0
L1 : LD R4, number
SUB R2, R1, R4
BLTZ R2, L2
LD R3, factorial
MUL R3, R3, R4
ST factorial, R3
SUB R4, R4, #1
ST number, R4
BR L1
L2:
```

```
while (number > 0)
{
factorial = factorial * number;
--number;
}
```

Phase 7 : Machine Independent Optimized Code

Output: Optimized Target Code

Compiler Design

Simple IR Optimizations

Constant folding is the process of recognizing and evaluating constant expressions at compile time rather than computing them at runtime.

Constant propagation is the process of substituting the values of known constants in expressions at compile time.

Common Subexpression Elimination (CSE) is a compiler optimization that searches for instances of identical expressions (i.e., they all evaluate to the same value), and analyzes whether it is worthwhile replacing them with a single variable holding the computed value.

Syntax Tree vs Parse Tree

Parse Tree (Concrete Syntax Tree)	(Abstract) Syntax Tree
It is huge and focuses on all lexeme	It focuses only on the syntax
Consists of both Non-terminals (P, S, E, Cond, RelOp, etc.) and Terminals (leaf nodes like tokens)	All nodes are tokens; It also has dummy nodes (<body>) to maintain proper syntax
Complicated	No non-terminals; Easy
Has punctuation (; , { })	No punctuation symbols

Compiler Design

Example 2

Example 2 : Consider the following piece of code:

```
int rhs, lhs = 0;
rhs = 2;
if (lhs <= rhs)
{
    lhs = lhs – 2 * rhs;
}
```

Run this code through the different phases of a compiler and produce the output at each stage.

THANK YOU

Preet Kanwal

Department of Computer Science & Engineering

preetkanwal@pes.edu

PES UNIVERSITY ONLINE

Compiler Design

Preet Kanwal

Department of Computer Science & Engineering

Teaching Assistant : Lavitra Kshitij Madan

Compiler Design

Unit 1

The Phases of a Compiler

Preet Kanwal

Department of Computer Science & Engineering

Compiler Design

Lecture Overview

PES UNIVERSITY ONLINE

In this lecture, you will practise:

- C Language Grammar
- Running an input through the different phases of a compiler

Compiler Design

Example 2

Example 2 : Consider the following piece of code:

```
int rhs, lhs = 0;
rhs = 2;
if (lhs <= rhs)
{
    lhs = lhs – 2 * rhs;
}
```

We will now run this code through the different phases of a compiler and see what the output will be at each stage.

Compiler Design

Example 2 (continued)

Phase 1 : Lexical Analysis or Linear Analysis or Scanner

Pattern	Lexemes matched
Keyword	int if
Identifier	rhs lhs rhs lhs rhs lhs lhs rhs
RelOp (<, <=, >, >=, ==, !=)	<=
LogOp (&&, ||, ~)	
Assign (=)	= = =
ArithOp (*, +, -, /)	- *
Punctuation ((,), {, }, [,], ;)	; ; ( ) { ; }
Number	0 2 2
Literal – anything in double quotes	
Inc_op	
Dec_op	

```
int rhs, lhs = 0;
rhs = 2;
if (lhs <= rhs)
{
lhs = lhs – 2 * rhs;
}
```

Compiler Design

Example 2 (continued)

Output: # of tokens : 27

< keyword, int >
< ID, 3 >
< punctuation, , >
< ID, 4 >
< Assign, = >
< Number, 0 >
< punctuation, ; >
< ID, 3 >
< Assign, = >
< Number, 2 >
< punctuation, ; >
< keyword, if >
< punctuation, (>
< ID, 4 >
< RelOp, <= >
< ID, 3 >
< punctuation,) >
< punctuation, { >
< ID, 4 >
< Assign, = >
< ID, 4 >
< ArithOp, - >
< Number, 2 >
< ArithOp, * >
< ID, 3 >
< punctuation, ; >
< punctuation, } >

Symbol Table

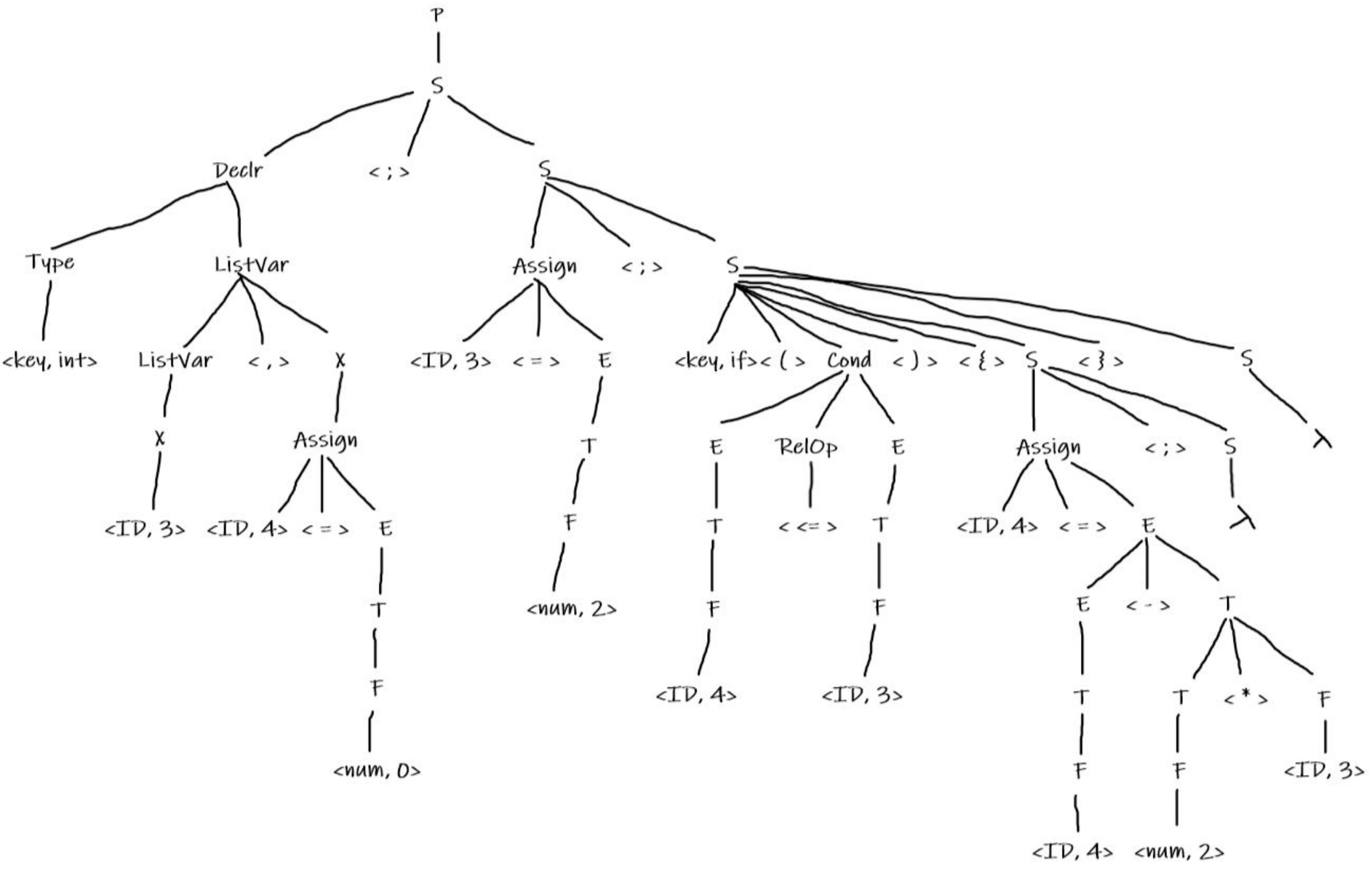
	name	type	…
1	…		
2	….		
3	rhs		
4	lhs		

```
int rhs, lhs = 0;

rhs = 2;

if (lhs <= rhs)

{

lhs = lhs – 2 * rhs;

}
```

Compiler Design

Example 2 (continued)

Phase 2 : Syntax Analysis or Parser



Use the grammar we have defined earlier

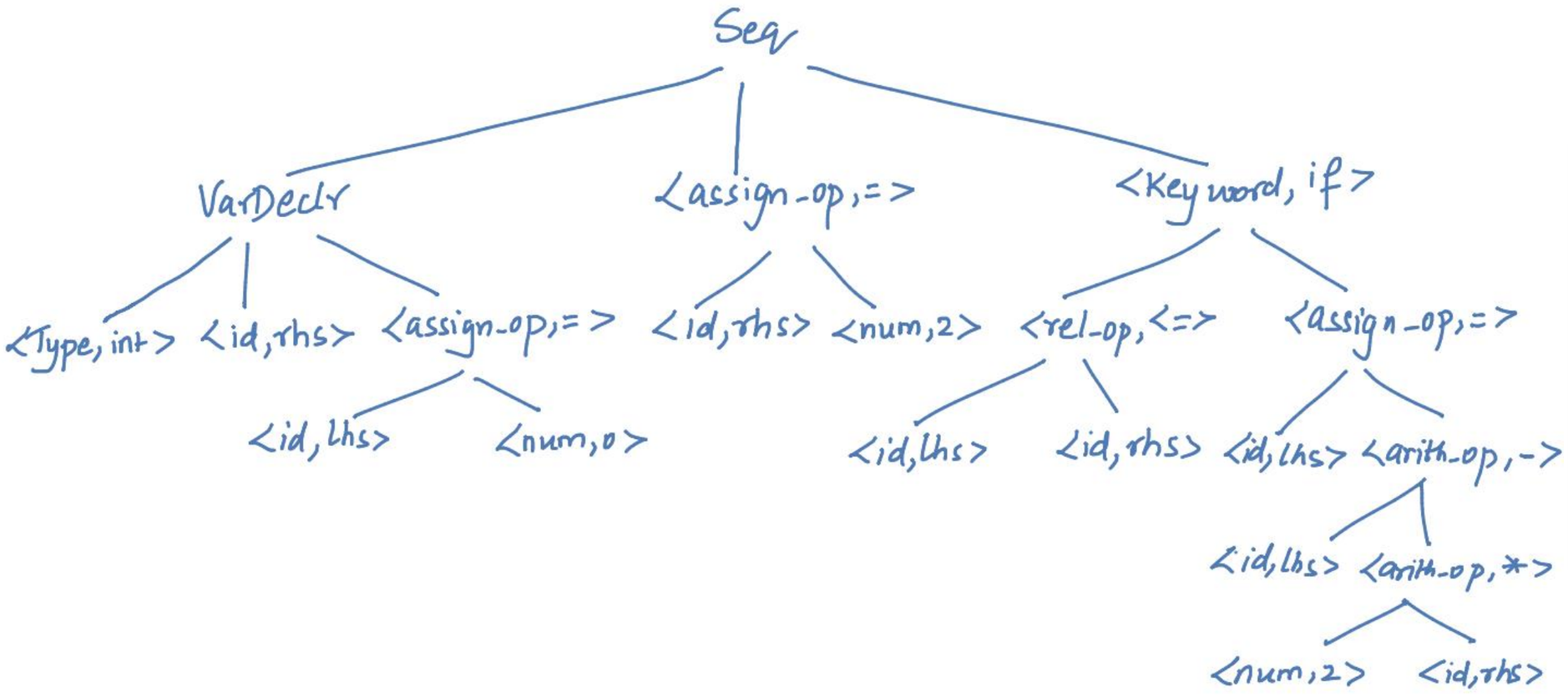
```
int rhs, lhs = 0;
rhs = 2;
if (lhs <= rhs)
{
lhs = lhs – 2 * rhs;
}
```

Compiler Design

Example 2 (continued)

Phase 3 : Semantic Analyzer

Output: Semantically checked Syntax Tree




```
int rhs, lhs = 0;
rhs = 2;
if (lhs <= rhs)
{
lhs = lhs – 2 * rhs;
}
```

Compiler Design

Example 2 (continued)

Phase 4 : Intermediate Code Generator

Output: Three Address Code (3AC or TAC)

```
lhs = 0
rhs = 2
t0 = lhs <= rhs
if t0 goto L1
goto L2
L1 : t1 = 2 * rhs
     t2 = lhs - t1
     lhs = t2
L2 : next
```

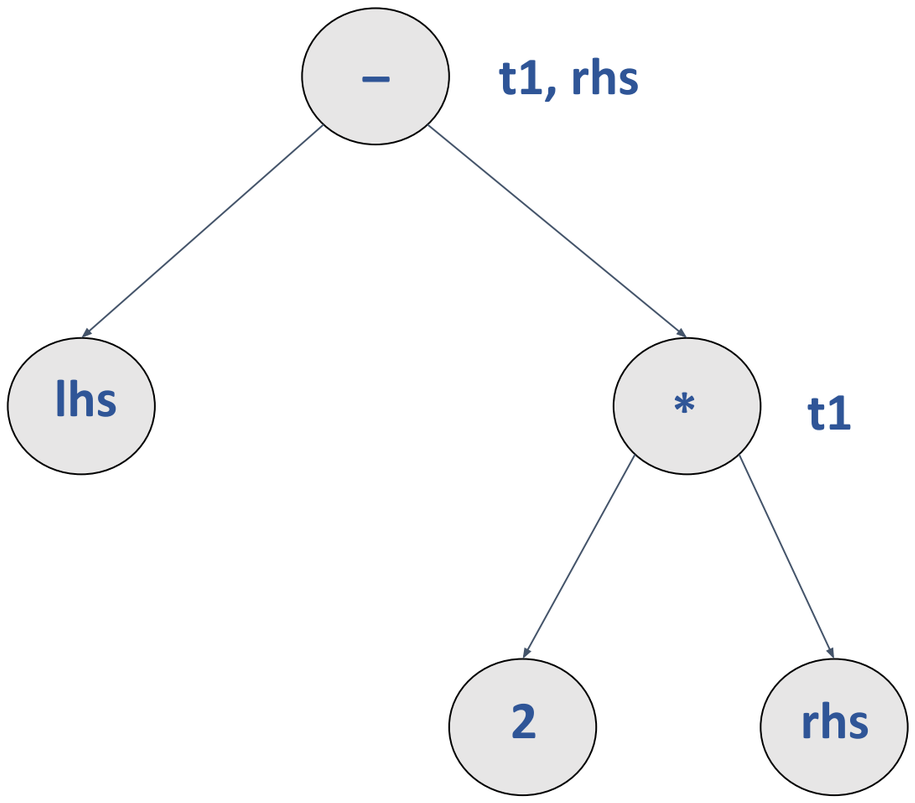
```
int rhs, lhs = 0;
rhs = 2;
if (lhs <= rhs)
{
lhs = lhs – 2 * rhs;
}
```

Compiler Design

Example 2 (continued)

Phase 5 : Machine Independent Code Optimization

Output: Optimized IR




```
lhs = 0
rhs = 2
t0 = lhs <= rhs
if t0 goto L1
goto L2
L1 : t1 = 2 * rhs
     lhs = lhs - t1
L2 : next
```

```
int rhs, lhs = 0;
rhs = 2;
if (lhs <= rhs)
{
lhs = lhs – 2 * rhs;
}
```

Compiler Design

Example 2 (continued)

Phase 6 : Code Generator

Output: Assembly Code

```
MOV R1, #0
ST lhs, R1
MOV R2, #2
ST rhs, R2
SUB R3, R1, R2
BLTZ R3, L2
LD R4, #2
MUL R4, R2
SUB R5, R1, R4
ST R5, lhs
L2 : ....
```

```
int rhs, lhs = 0;
rhs = 2;
if (lhs <= rhs)
{
lhs = lhs – 2 * rhs;
}
```

Example 3

Example 3 : Consider the following piece of code:

```
n = 23;
for (i = 0; i < n; i++)
{
    sum = sum * i;
}
```

We will now run this code through the different phases of a compiler and see what the output will be at each stage.

Compiler Design

Example 3 (continued)

Phase 1 : Lexical Analysis or Linear Analysis or Scanner

Pattern	Lexemes matched
Keyword	for
Identifier	n i i n i printf sum sum i
RelOp (<, <=, >, >=, ==, !=)	<
LogOp (&&, ||, ~)	
Assign (=)	= = =
ArithOp (*, +, -, /)	*
Punctuation ((,), {, }, [,], ;)	; (; ;) { ; }
Number	23 0
Literal – anything in double quotes	
Inc_op	++
Dec_op	

```
n = 23;
for (i = 0; i < n; i++)
{
sum = sum * i;
}
```

Compiler Design

Example 3 (continued)

Output: # of tokens : 25

< ID, 1 >
< = >
< Number, 23 >
< Keyword, for >
< (>
< ID, 2 >
<assign_op, =>
< Number, 0 >
< ; >
< ID, 2 >
< < >
< ID, 1 >
< ; >
< ID, 2 >
< Inc_op, ++ >
<) >

Symbol Table

	name	type	…
1	n		
2	i		
3	sum		
	…		

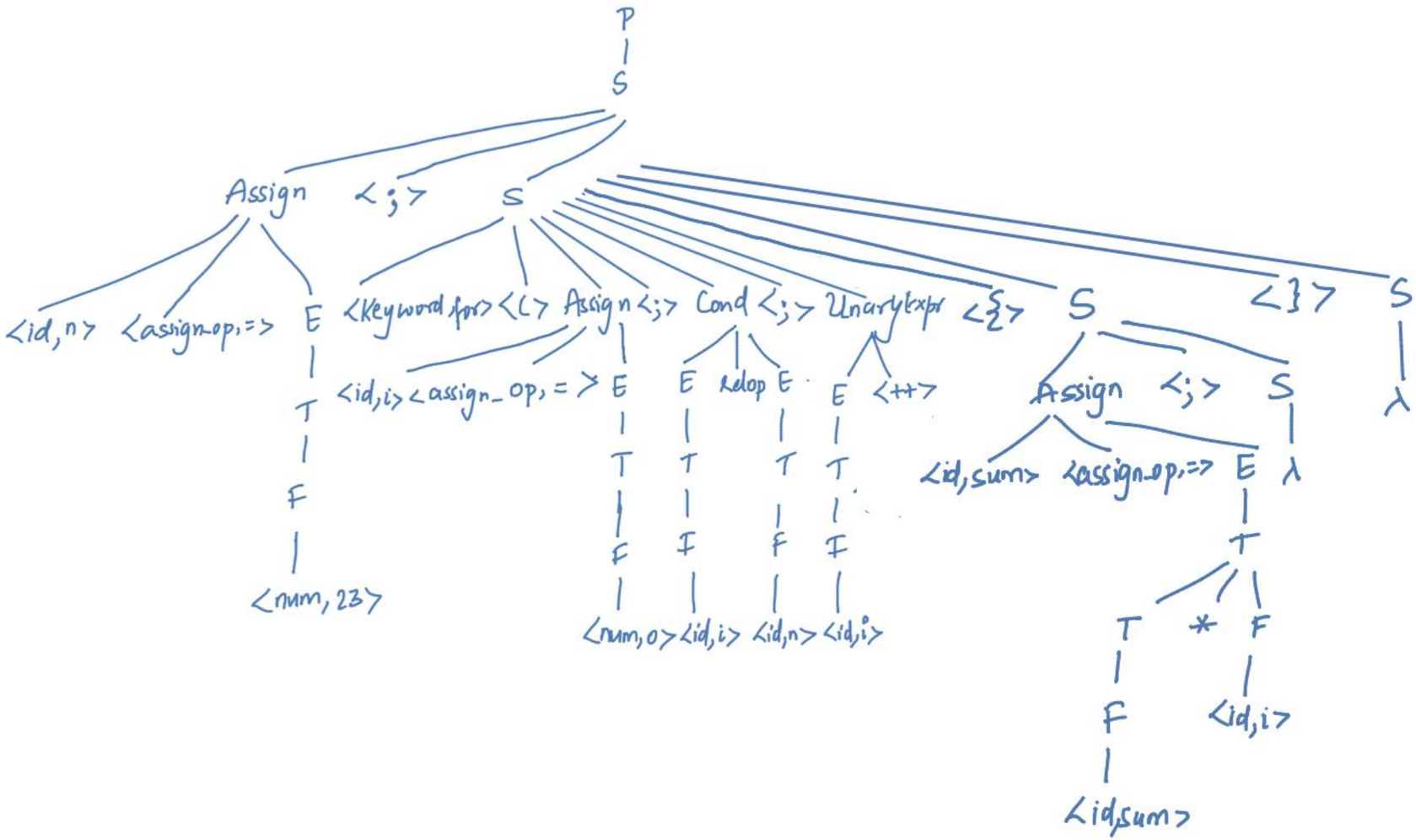
< { >
<ID, 3>
<assign_op, =>
<ID, 3>
<arith_op, *>
<ID, 2>
< ; >
< } >

```
n = 23;
for (i = 0; i < n; i++)
{
sum = sum * i;
}
```

Compiler Design

Example 3 (continued)

Phase 2 : Syntax Analysis or Parser



Assume slightly modified grammar (with functions)

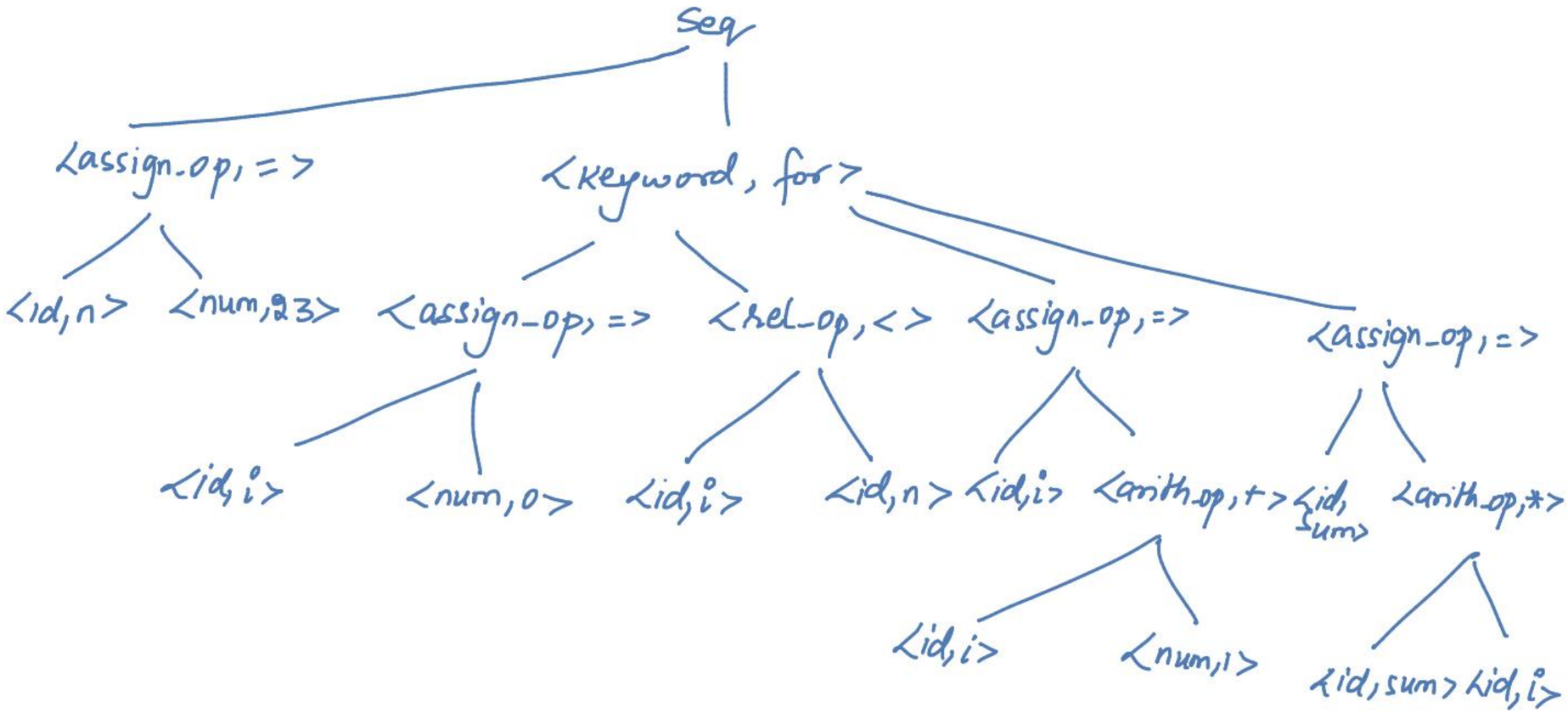
```
n = 23;
for (i = 0; i < n; i++)
{
sum = sum * i;
}
```

Compiler Design

Example 3 (continued)

Phase 3 : Semantic Analyzer

Output: Semantically checked Syntax Tree




```
n = 23;
for (i = 0; i < n; i++)
{
sum = sum * i;
}
```

Compiler Design

Example 3 (continued)

Phase 4 : Intermediate Code Generator

Output: Three Address Code (3AC or TAC)

```
number = 23
i = 0
L0 : If i < n goto L1
 goto L2
 L1 : t1 = sum * i
sum = t1
 t2 = i + 1
 i = t2
 goto L0
 L2 : ...
```

```
n = 23;
for (i = 0; i < n; i++)
{
sum = sum * i;
}
```

Compiler Design

Example 3 (continued)

Phase 5 : Machine Independent Code Optimization

Output: Optimized IR

```
number = 23
i = 0
L0 : If i < n goto L1
goto L2
L1 : sum = sum * i
i = i + 1
goto L0
L2 : …
```

```
n = 23;
for (i = 0; i < n; i++)
{
sum = sum * i;
}
```

Compiler Design

Example 3 (continued)

Phase 6 : Code Generator

Output: Assembly Code

```
    MOV R1, #23    // R1 → n
    MOV R2, #0     // R2 → i
    MOV R4, sum    // R4 = contents(sum)
L0 : SUB R3, R1, R2  // i - n
     BLZ R3, L1
     BR L2
L1 : MUL R4, R4, R2
     BR L0
L2: …
```

```
n = 23;
for (i = 0; i < n; i++)
{
sum = sum * i;
}
```

Compiler Design

Example 4 - Take home exercise

Exercise Problem : Consider the following piece of code:

```
int n = 3;
do
{
    if( (n/2)*2 == n )
    {
        n = n / 2;
    }
    else
    {
        n = 3 * n + 1;
    }
}
while(n != 1 || n != 2 || n != 4);
return n;
```

This problem is for the understanding of the concept by testing you on if-else and do-while constructs.

Note: The Parse Tree may become large, try to do the if-else part separately

Hint: Consider
S -> do {S} while (S); S
in the grammar

THANK YOU

Preet Kanwal

Department of Computer Science & Engineering

preetkanwal@pes.edu

PES UNIVERSITY ONLINE

Compiler Design

Preet Kanwal

Department of Computer Science & Engineering

Teaching Assistant : Kavya P K

Compiler Design

Unit 1

Lexical Analysis

Preet Kanwal

Department of Computer Science & Engineering

Compiler Design

Lecture Overview

PES UNIVERSITY ONLINE

In this lecture, you will learn about:

- The Lexical Analyser
- Role of a lexer
- The need to separate Lexical Analyser and Parser and the interaction between them
- Challenge of Lexical Analyser - Speed
- Input Buffer
  - Buffer Pairs
  - Buffer Pair Algorithm
  - Sentinels
- Lexical Errors

Compiler Design

Some important definitions

1. Token :- It is a pair consisting of a token name and an optional attribute value.

   <token name, attribute value>

   - Token name is an abstract symbol representing a kind of lexical unit - keywords, identifiers etc.
   - Attribute value is pointer to the symbol table entry.

1. Pattern :- It is a description of the form that the Lexemes of a token may take.
   a. Keyword - a sequence of characters
   b. Identifiers - sequence of character with some complex Structure
2. Lexeme :- It is a sequence of characters in the source program that matches the pattern for a token, i.e, and instance of a token

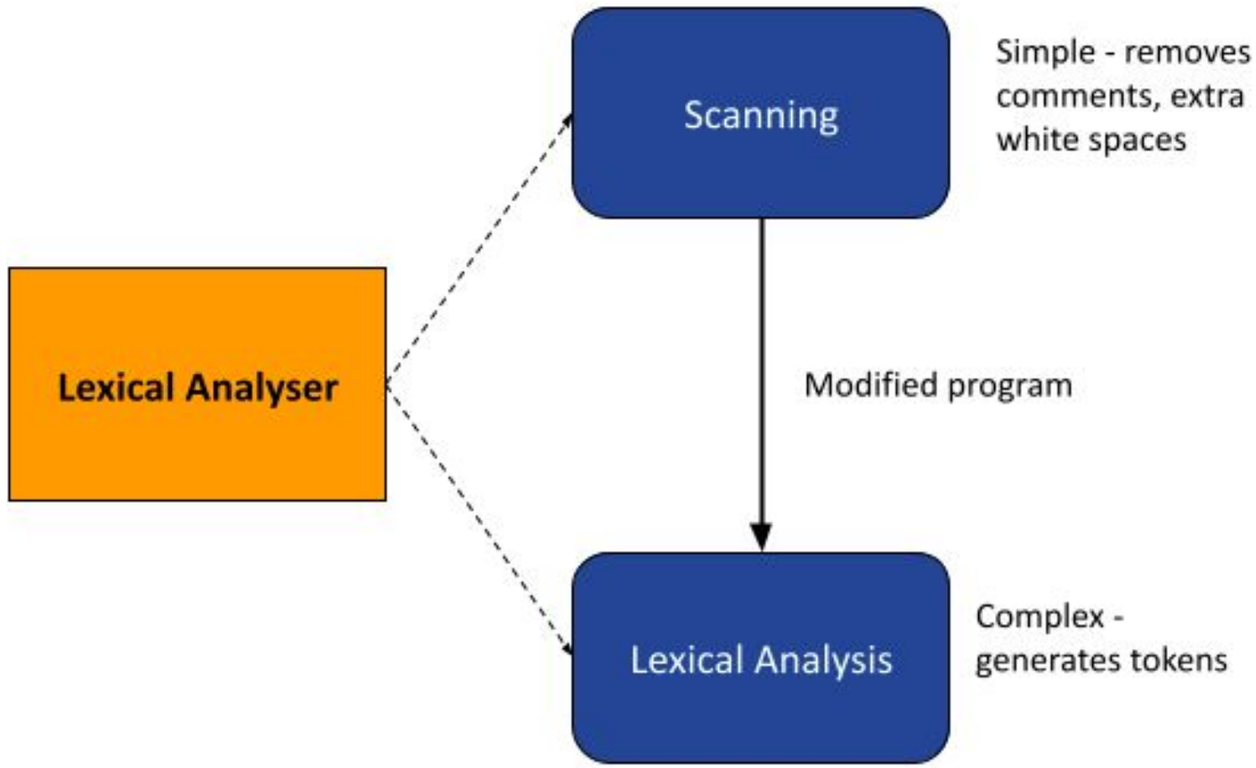
Examples

Token	Informal Description of Pattern	Lexemes
if	Character i,f	if
else	Characters e,l,s,e	else
Comparison	< or > or <= or >= or == or !=	<=,!=
id	Letter followed by letters and digits	Sum, avg1, pi
number	Any numeric constant	10001, 3.14
literal	Anything but “, surrounded by “	“compiler design”

Compiler Design

Introduction to Lexical Analyzer

- The Lexical Analyzer is the first phase of the compiler.
- It is language dependent.
- It performs two main tasks -
  1. Scanning
     a. Reading the input characters of the source program.
     b. Performing simple processes that do not require tokenization of the input - deletion of comments, compaction of consecutive whitespaces.
  2. Lexical Analysis - Producing the sequence of tokens as output.



Compiler Design

Goal of a Lexical Analyzer

1. To convert from physical description of a program into sequence of of tokens.
2. Each token represents one logical piece of the source file - a keyword, the name of a variable etc.
3. Each token is associated with a lexeme, i.e, the actual text of the token: ”137,” ”int,” etc.
4. Each token can have optional attributes - Extra information derived from the text, perhaps a numeric value.
5. The token sequence will be used in the parser to recover the program structure.

Compiler Design

Role of a lexer (1)

The Lexical analyzer reads the input characters of the source program and then performs the following steps:

1. Groups the input characters into Lexemes.
2. Produces a sequence of tokens for each lexeme in the source program as output.
3. Sends the stream of tokens to the parser for Syntax Analysis.
4. When a lexeme representing an identifier is discovered, it is entered into the symbol table.

Compiler Design

Role of a Lexer (2)

The Lexical Analyzer also performs the following secondary tasks:

- Stripping out comments and white space like blank, new line, tab and other characters that are used to separate tokens in the input.
- Correlating error messages generated by the compiler with the source program file.
- Keeping track of the number of the newline characters and associate each error message with its line number.
- Make a copy of the source program with error messages inserted at appropriate positions.
- Expansion of macros

Compiler Design

The need to separate Lexical Analyzer and Parser

- There exists a tight coupling between lexical analyzer and Parser.
- The Parser works as the master program, and directs the lexical analyzer.
- However, there are several reasons to separate lexical analyzer and syntax analysis into different phases -

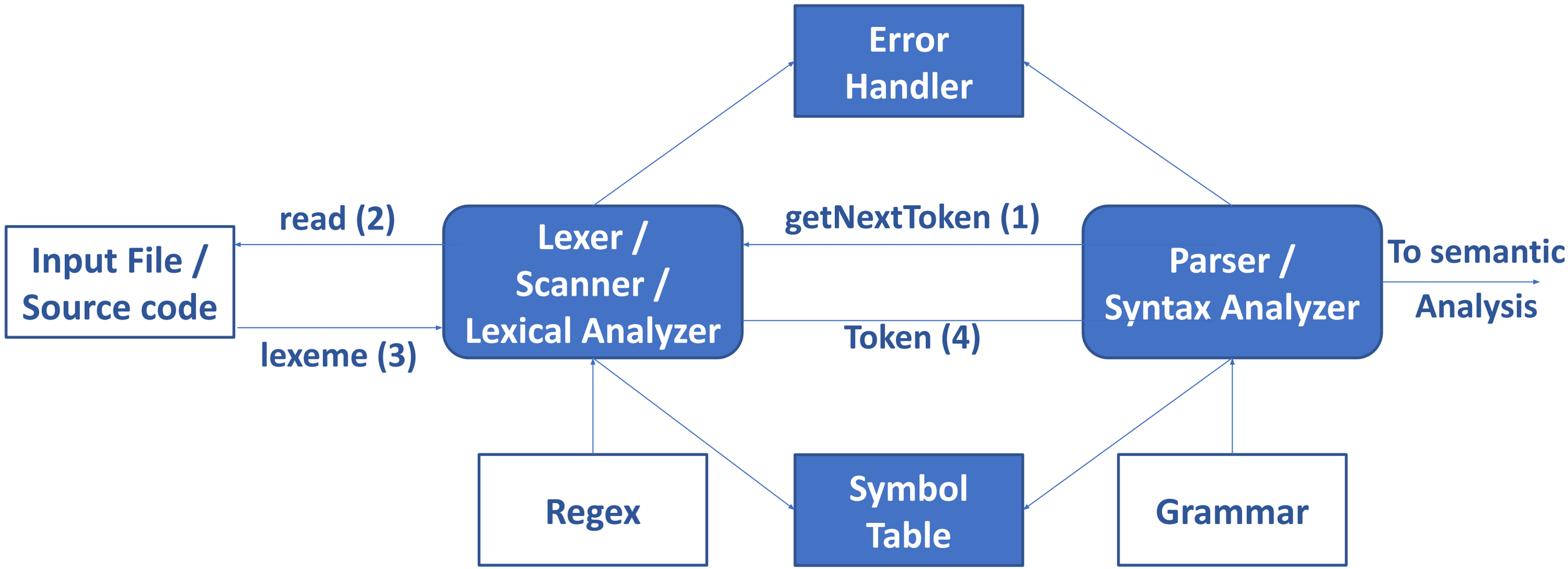
1. Simplicity of design
   - Comments and spaces are removed in the lexical analysis phase.
   - If performed by the parser, parser design becomes very complex.
   - Separating lexical and syntactical concerns leads to cleaner design.
1. Compiler efficiency is improved - specialised buffering schemes can be used for reading characters, which can speed up the compiler.
2. Compiler portability is enhanced - input specific peculiarities can be restricted to the lexical analyzer.

Lexical Analyzer Versus Parser

Lexical Analyser	Parser
Scan Input program	Perform syntax analysis
Identify Tokens	Create an abstract representation of the code
Insert tokens into Symbol Table	Update symbol table entries
It generates lexical errors	It generates a parse tree of the source code

Compiler Design

Interaction Between the Lexical Analyzer and Parser(1)



Compiler Design

Interaction Between the Lexical Analyzer and Parser (2)

- Lexer stops when parser stops since the parser is its master.
- If working only with the lexer, then the lexer must be explicitly stopped lest it will continue running infinitely.
- Comments are eliminated by the preprocessor but even the compiler can remove comments in the lexer phase. If “-E” is used in the gcc command, then comments are not removed in the preprocessing phase. So, the lexer can make sure that comments do not reach the parser.

Compiler Design

Challenge: Speed of Lexical Analyzer

- The lexical analyzer reads characters of the source program one at a time to discover tokens.
- Thus, speed of lexical analysis is a concern.
- In many languages, there are scenarios when the lexer needs to look ahead at least one character before a match can be announced.

Solution - The lexical analyzer reads from Input Buffers

- There are many schemes that can be used to buffer input. This course discusses two schemes -
  - Buffer Pairs
  - Sentinels

Compiler Design

Input Buffers - Buffer Pairs

- This scheme makes use of two buffers of the same size(N).
- Usually, N = size of disk block = 4096 bytes.
- These buffers are alternately reloaded.
- A single system read can be used to read N characters into the buffer instead of one system call per character.
- If fewer than N characters remain in the input file, EOF is read into the buffer, marking the end of the source file.

Compiler Design

Input Buffers - Buffer Pairs

- Two pointers are maintained to the input buffer -
  1. lexeme_begin - marks the beginning of the current lexeme
  2. forward - scans ahead till a pattern is found.
- Initially, both pointers point to the first character of the next lexeme to be found.
- The forward pointer scans ahead until a match for the pattern is found.
- The string of characters between the two pointers is the current lexeme.
- Once the lexeme is determined, the forward pointer is set to the character at the right end of the lexeme.
- After the lexeme is processed, both pointers are set to the character immediately after the lexeme.

Input Buffering - Buffer Pairs Algorithm

```
if forward at end of first half then begin
        reload second half;
        forward := forward +1
end
else if forward at end of second half then begin
        reload first half;
        move forward to beginning of first half
end
else
        forword := forward + 1;
```

Compiler Design

Input Buffering - Buffer Pairs Example

Consider the following statement -

abc = pqr * xyz ;

Compiler Design

Input Buffering - Buffer Pairs Example

abc = pqr * xyz ;

Initially, both pointers are set to the beginning of the buffer.

Compiler Design

Input Buffering - Buffer Pairs Example

abc = pqr * xyz ;

Forward pointer advances till a pattern is matched.

Compiler Design

Input Buffering - Buffer Pairs Example

abc = pqr * xyz ;

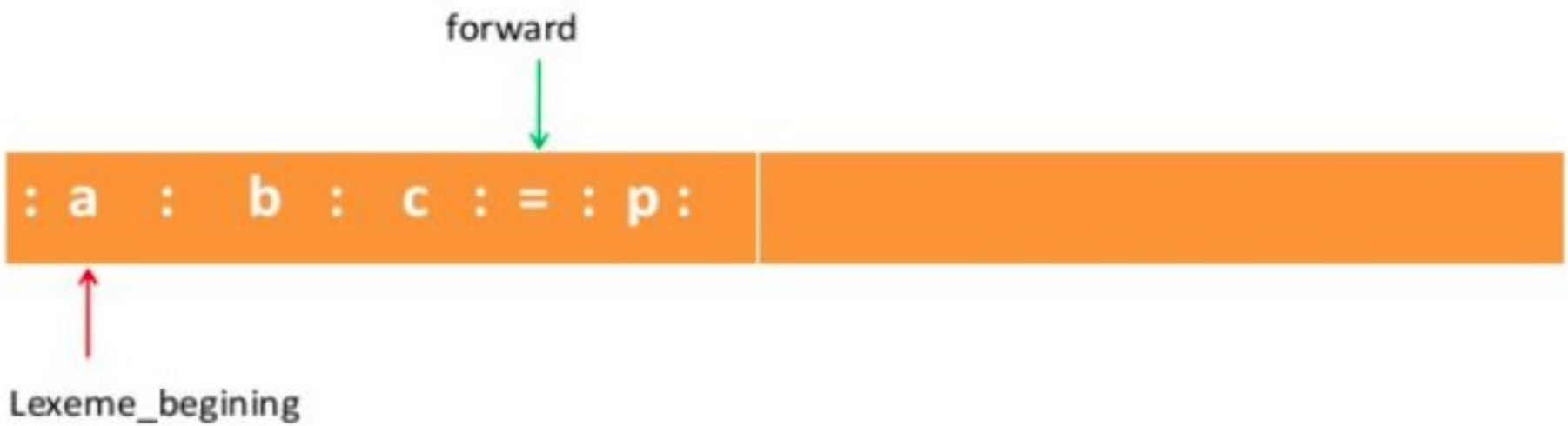
Forward pointer advances till a pattern is matched.

Input Buffering - Buffer Pairs Example

abc = pqr * xyz ;

Here, a pattern is matched. The current lexeme lies between the two pointers.

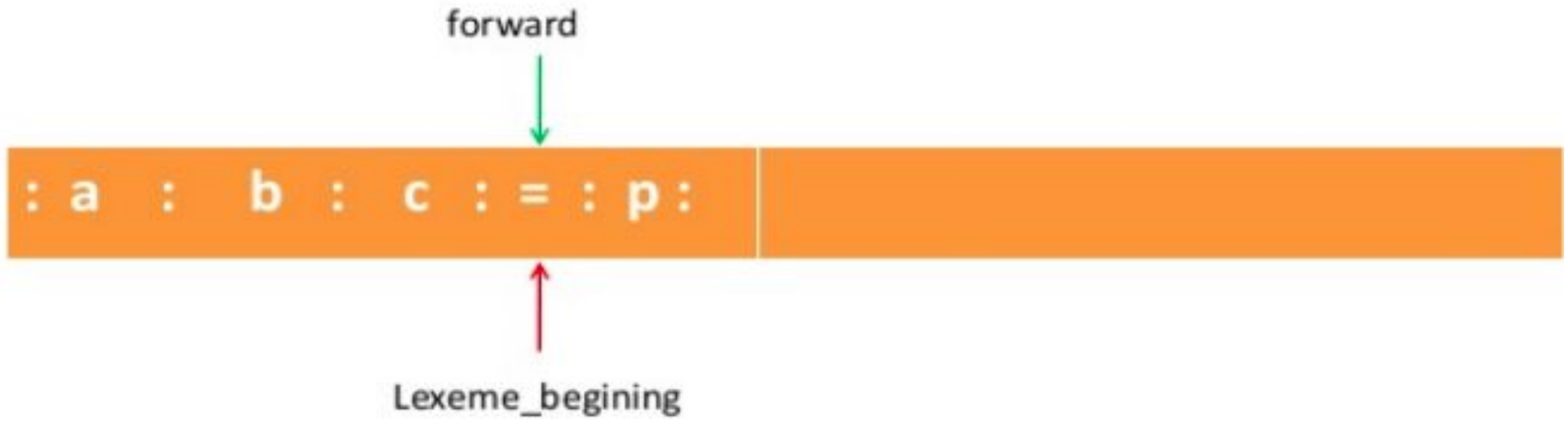
Current lexeme: abc



Input Buffering - Buffer Pairs Example

abc = pqr * xyz ;

Both forward and lexeme_begin are set to the next character after the lexeme.

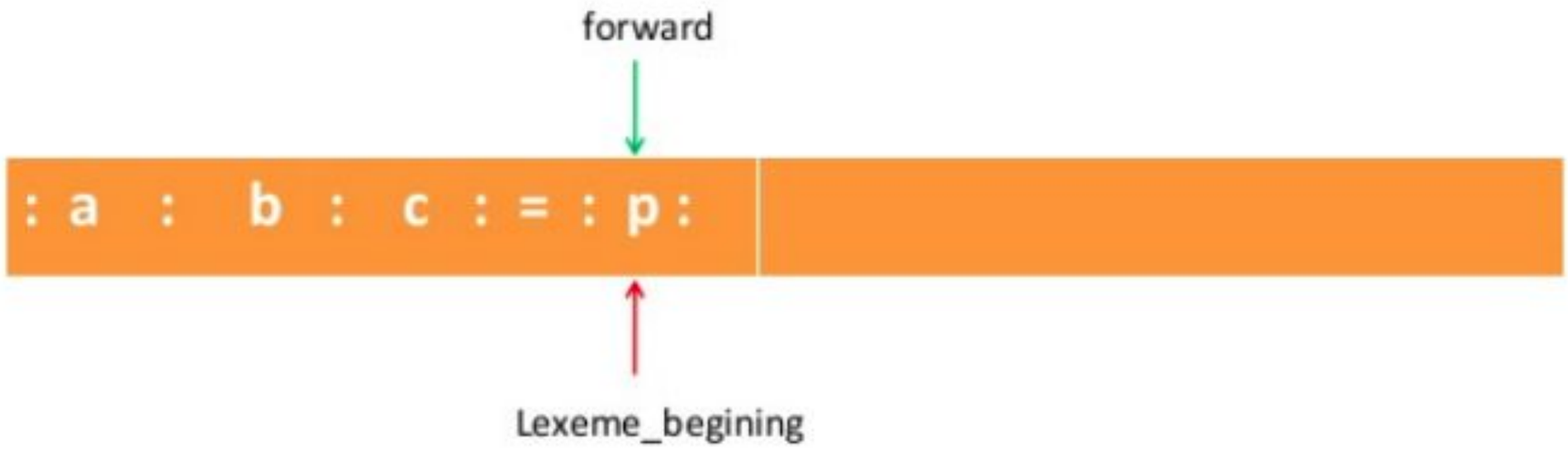


Input Buffering - Buffer Pairs Example

abc = pqr * xyz ;

The current lexeme is =

Both forward and lexeme_begin are moved to the next character after the current lexeme.

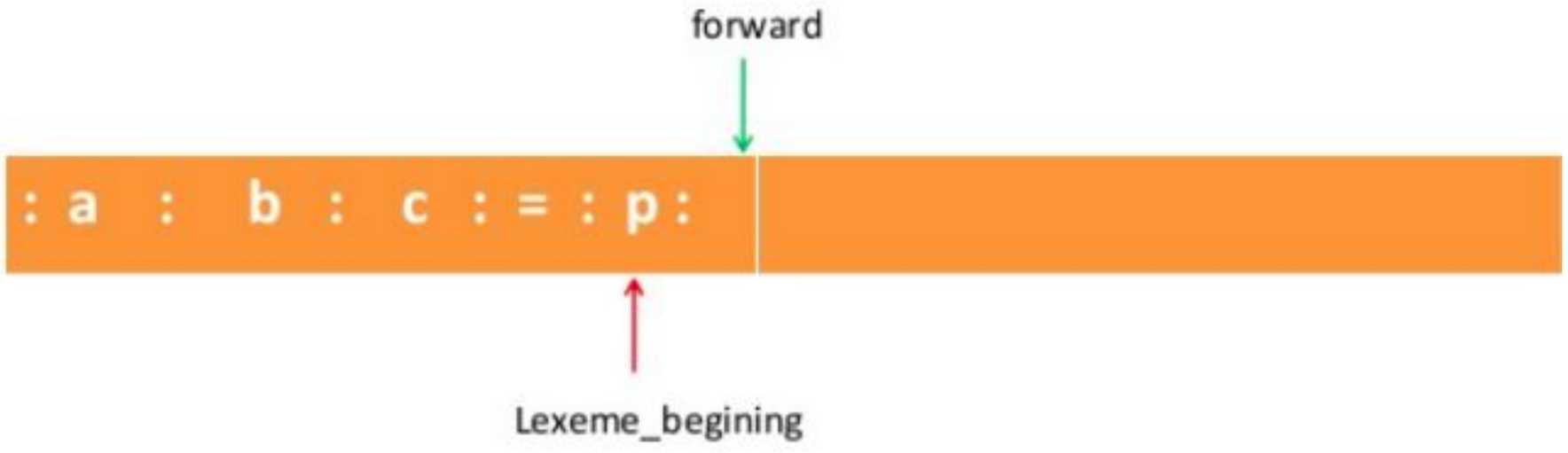


Compiler Design

Input Buffering - Buffer Pairs Example

abc = pqr * xyz ;

The forward pointer has reached the end of the first buffer.

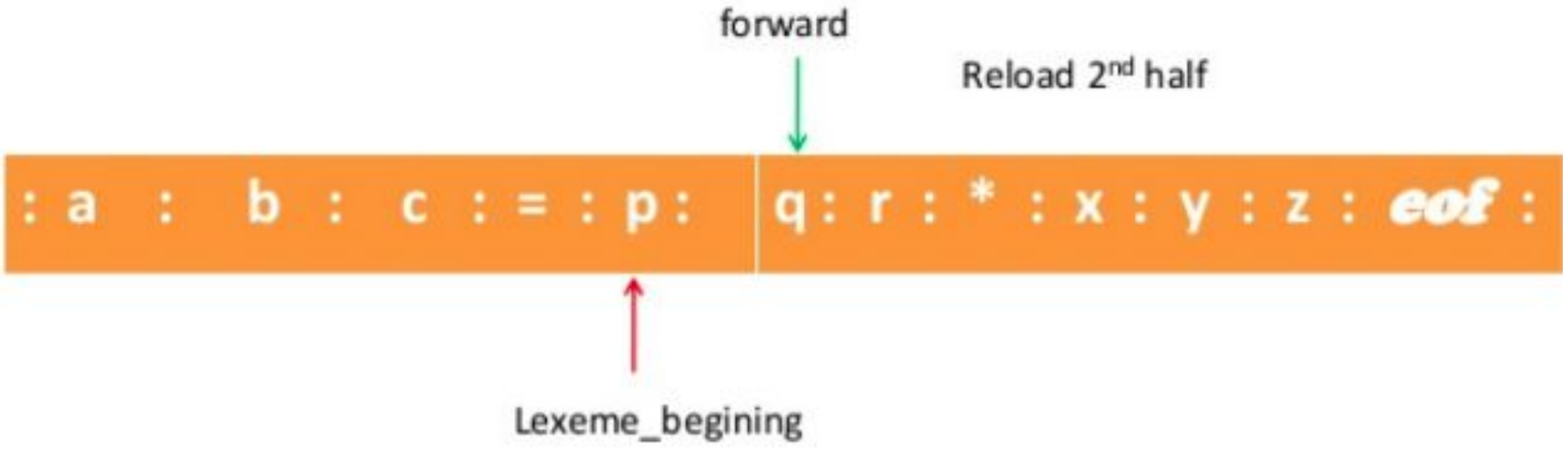


Input Buffering - Buffer Pairs Example

abc = pqr * xyz ;

The second input buffer is reloaded. Forward pointer moves ahead till the next pattern is matched.

This process continues till EOF is reached.



Compiler Design

Drawbacks of Buffer Pairs

The drawbacks of the buffer pairs algorithm is that each time we advance forward, there are two checks that need to be performed -

1. For End Of Buffer
2. To determine which character is read

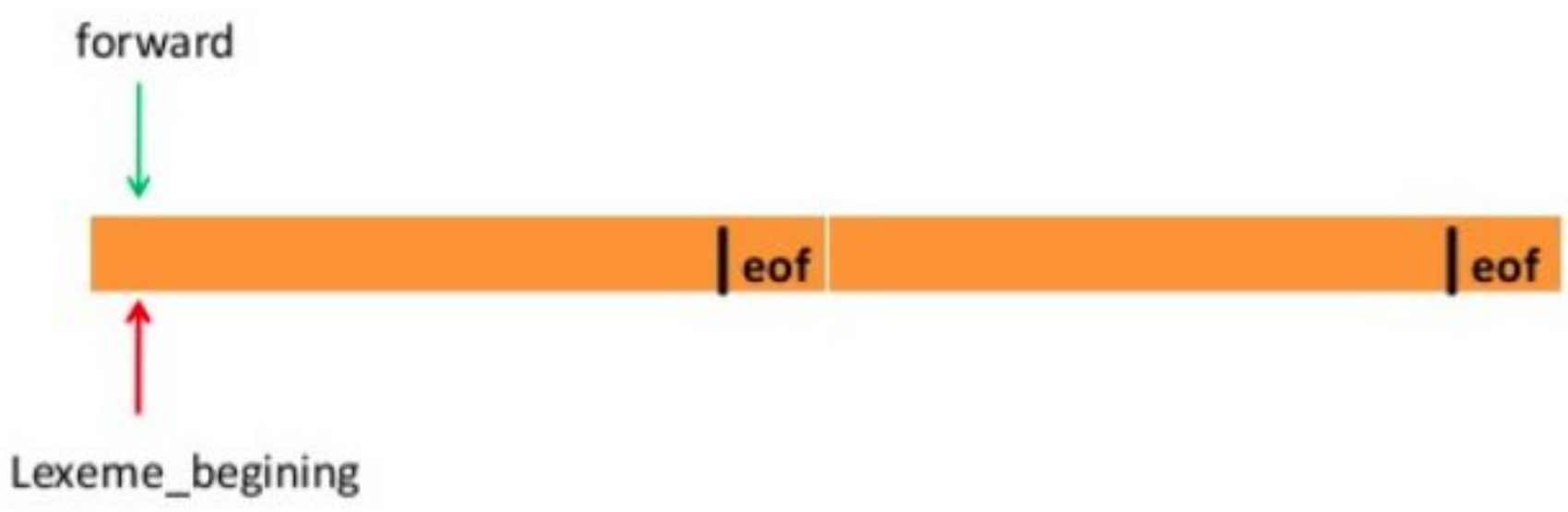
Solution - Combine the buffer-end check with the check for current character.

This can be done by extending each buffer to hold a Sentinel character (EOF) at the end.

Compiler Design

Input Buffering : Buffer pairs with Sentinels

Sentinels (eof) are added to the end of each input buffer.



Input Buffering - Buffer pair Algorithm (with sentinels)

```
forward = forward + 1;
if forward = eof then begin
        if forward at end of first half then begin
                reload second half;
                forward := forward +1
        end
        else if forward at end of second half then begin
                reload first half;
                move forward to beginning of first half
        end
        else
                terminate lexical analysis
                /* eof within buffer, i.e, end of input */
end
```

Compiler Design

Lexical Errors

Suppose a situation arises in which the lexical analyzer is unable to proceed because none of the patterns for tokens matches any prefix of the remaining input.

Some possible scenarios are -

1. Spelling Error
2. Unmatched string
3. Appearance of illegal characters
4. Exceeding length of identifiers

Compiler Design

Lexical Errors

1. Spelling Error
2. Unmatched string
3. Appearance of illegal characters
4. Exceeding length of identifiers

```
di{…
……}
while();
```

Compiler Design

Lexical Errors

1. Spelling Error
2. Unmatched string
3. Appearance of illegal characters
4. Exceeding length of identifiers

```
printf(“hi);

printf(“hi”);
```

Compiler Design

Lexical Errors

1. Spelling Error
2. Unmatched string
3. Appearance of illegal characters
4. Exceeding length of identifiers

```
printf(“hi”)$$;
```

Compiler Design

Lexical Errors

1. Spelling Error
2. Unmatched string
3. Appearance of illegal characters
4. Exceeding length of identifiers

More than 32 characters in an identifier

Error Message

When writing an error message, make sure it satisfies the following properties:

- **It must pinpoint the error correctly**
- **It must be understandable**
- **It must be specific**
- **It must not have any redundancy**

THANK YOU

Preet Kanwal

Department of Computer Science & Engineering

preetkanwal@pes.edu

PES UNIVERSITY ONLINE

Compiler Design

Preet Kanwal

Department of Computer Science & Engineering

Teaching Assistant : Kavya P K

Compiler Design

Lex Tutorial

Preet Kanwal

Department of Computer Science & Engineering

Compiler Design

Lecture Overview

PES UNIVERSITY ONLINE

In this lecture, you will learn about:

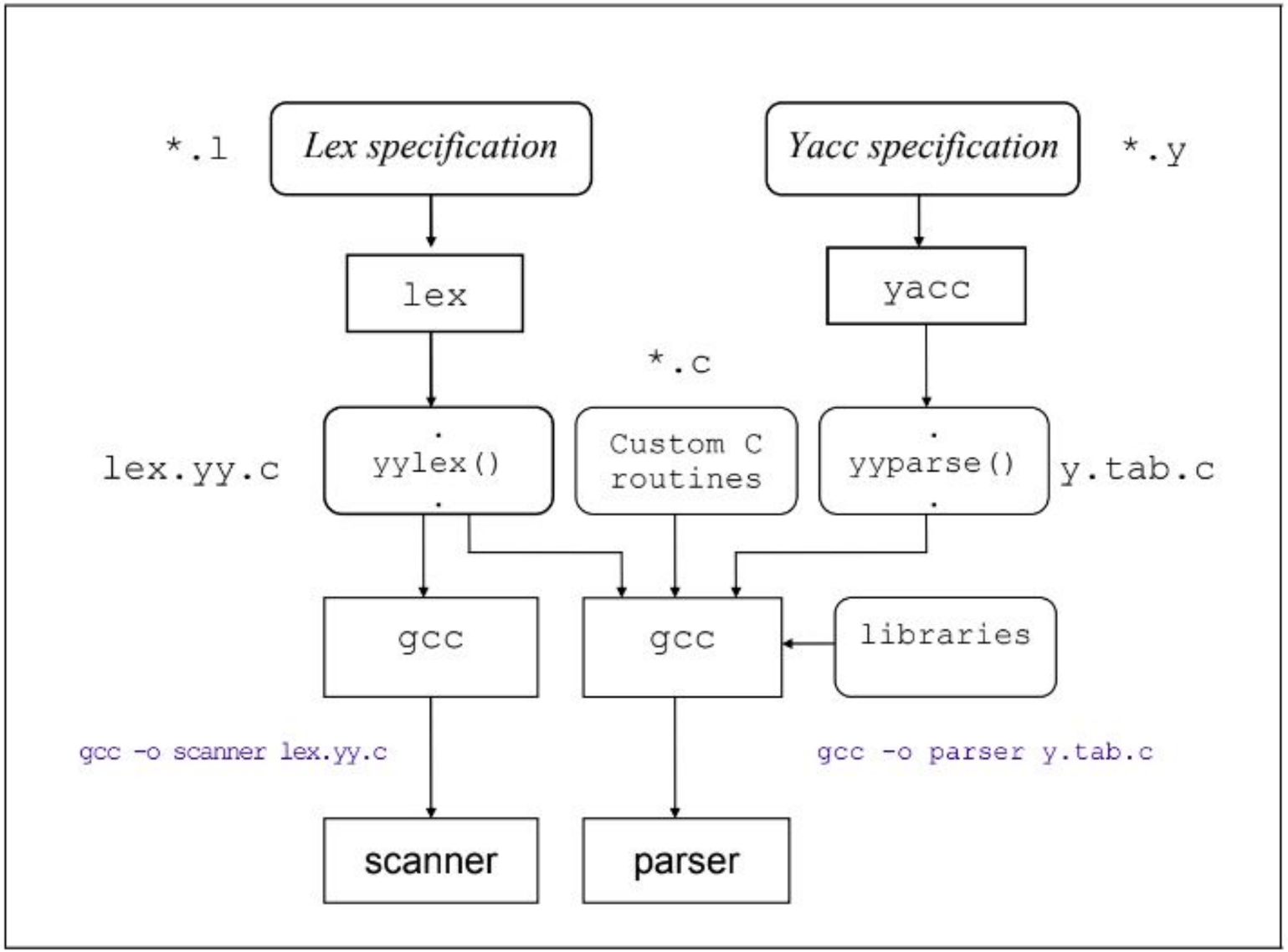
- What is lex and yacc?
- Flow of code
- How to write a Lex Specification file ?
- Example 1 - A simple program
- Predefined variables in lex
- Example 2 - Prepend Line Numbers
- Example 3 - Use of definitions
- Example 4 - Word count
- Example 5 - Removing Comments
- Start conditions
- Commonly used Regular expressions in lex specification file
- Pseudocode for loop-switch implementation

Lex and Yacc

- **Lex** - reads a specification file containing regular expressions and generates a C routine that performs lexical analysis. Matches sequences that identify tokens.
- **Yacc** - reads a specification file that codifies the grammar of a language and generates a parsing routine.

Compiler Design

Flow of code - using lex and yacc tools



How to write a Lex Specification file (*.l) ?

- The Lex specification file is divided into three sections with %% dividing the sections.
- The structure of a lex specification file is as follows -

```
definitions
%%
regular expressions and associated actions (rules)
%%
user routines
```

Compiler Design

Lex Specification file

- Input is copied to output one character at a time.
- The first %% is always required as there must always be a rules section.
- However, if we don't specify any rules, then the default action is to match everything and copy it to output.
- Defaults for input and output are stdin and stdout.
- Any source code not intercepted by lex is copied into the generated program.
  - A line that is not part of a lex rule or action, which begins with a blank or tab, is copied out as above (useful for global declarations)
  - Anything included between lines containing only %{ and %}

    (useful for preprocessor statements that must start in col.1)
  - Anything after the second %% delimiter is copied out after the lex output (useful for local function definitions)

Lex Specification file

- Definitions intended for lex are given before the first %%. Anyline in this section that does not begin with a blank or tab, or is not enclosed by %{...%}, is assumed to be defining a lex substitution string of the form

  name translation , for e.g, letter [a-zA-Z]

- yytext : a character array that contains the actual string that matches a pattern.
- yyleng : the no. of characters matched.
- Values associated with the token are returned by lex in a union variable yylval.
- Local variables can be defined.
- Do not leave extra spaces and/or empty lines at the end of the lex specification file.

```
[0-9]+ { yylval =
atoi(yytext); return
INTEGER;}
```

Compiler Design

Lex Specification file - A Simple Example

- Here, two patterns have been specified in the rules section.
- Each pattern must begin in column one.
- This is followed by whitespace (space, tab or newline) and an optional action associated with the pattern.
- The action may be a single C statement, or multiple C statements enclosed in braces.
- Anything not starting in column one is copied to the generated C file. This behavior can be to specify comments in the lex file.
- In this example there are two patterns, "." and "\n", with an ECHO action associated for each pattern.

```
%%
. ECHO;
\n  ECHO;
%%
int yywrap() {
   return 1;
}
int main(void) {
yylex();
   return 0;
}
```

Lex Specification file - A Simple Example (cont.)

- Several macros and variables are predefined by lex.
  - ECHO is a macro that writes code matched by the pattern. This is the default action for any unmatched strings. Typically, ECHO is defined as:

    #define ECHO fwrite(yytext, yyleng, 1, yyout)
- Function yywrap is called by lex when input is exhausted. Return 1 if done , 0 if more processing is required.
- Every C program requires a main function. In this case we simply call yylex which is the main entry-point for lex.
- Some implementations of lex include copies of main and yywrap in a library thus eliminating the need to code them explicitly.

```
%%
. ECHO;
\n ECHO;
%%
int yywrap() {
   return 1;
}
int main(void) {
yylex();
   return 0;
}
```

Some predefined variables in Lex

Table : Lex Predefined Variables	
Name	**Function**
int yylex(void)	call to invoke lexer, returns token
char *yytext	pointer to matched string
yyleng	length of matched string
yylval	value associated with token
int yywrap(void)	wrapup, return 1 if done, 0 if not done
FILE *yyout	output file
FILE *yyin	input file
INITIAL	initial start condition
BEGIN condition	switch start condition
ECHO	write matched string

Compiler Design

Lex Specification file - Example 2 - Line Numbers

- This example prepends line numbers to each line in a file.
- Some implementations of lex predefine and calculate yylineno.
  - %option yylineno is added to the definitions
  - Check program file : start_conditions.l
- Here, the input file for lex is yyin (which defaults to stdin), which is initialised in the main function.
- Variable yytext is a pointer to the matched string (NULL-terminated) and yyleng is the length of the matched string.
- Variable yyout is the output file and defaults to stdout.

```
%{
   int yylineno;
%}
%%
(.*)\n    printf("%4d\t%s",
yylineno++, yytext);
%%
int yywrap() { return(1); }
int main() {
   yyin = fopen("input.txt", "r");
   yylex();
   fclose(yyin);
}
```

Program file - line_numbers.l

Compiler Design

Lex Specification file - Example 3 - Using definitions

- This example demonstrates the use of the definitions section.
- The definitions section is composed of substitutions, code, and start states.
- Code in the definitions section is simply copied as-is to the top of the generated C file and must be bracketed with %{ and %} markers.
- Substitutions simplify pattern-matching rules.

```
digit     [0-9]
letter    [_A-Za-z]
%{
#include<stdio.h>
%}
%%
{letter}({letter}|{digit})*    printf("Identifier : %s",yytext);
. ;
%%
int yywrap(){return(1);}
int main()
{
   yyin = fopen("input.txt", "r");
   yylex();
   fclose(yyin);
   return 0;
}
```

Program file - identifiers.l

Compiler Design

Lex Specification file - Example 4 - Word count

- This example counts the number of characters, words, and lines in a file (similar to Unix wc)
- Whitespace must separate the defining term and the associated expression.
- References to substitutions in the rules section are surrounded by braces ({letter}) to distinguish them from literals.
- When we have a match in the rules section the associated C code is executed.

```
%{
   int nchar, nword, nline;
%}
%%
\n        { nline++; nchar++; }
[^ \t\n]+  { nword++, nchar += yyleng; }
.         { nchar++; }
%%
int yywrap(){return(1);}
int main(void) {
   yyin = fopen("input.txt", "r");
   yylex();
   printf("%d\t%d\t%d\n", nchar, nword, nline);
   return 0;
}
```

Program file - counts.l

Compiler Design

Lex Specification file - Example 5 - Removing comments

- This example removes single line and double line comments.

```
%%
\/\/(.*) ;
\/\*(.*\n*)*.*\*\/ ;
%%
int yywrap(){return(1);}
int main()
{
yyin=fopen("input.txt","r");
yylex();
return 0;
}
```

Program file - removing_comments.l

Start conditions

- Start conditions are a mechanism for conditionally activating patterns.
- This is useful for handling -
  - Conceptually different components of an input
  - Situations where the lex defaults (e.g., “longest possible match”) don’t work, like in comments or quoted strings.

- Declare a set of start condition names using

  %Start name1 name2 . . .

- If scn is a start condition name, then a pattern prefixed with <scn> will only be active when the scanner is in start condition scn.

Compiler Design

Start conditions

- The scanner begins in start condition INITIAL, of which all non-<scn>-prefixed rules are members.
- Start conditions such as these are inclusive: i.e., being in that start condition adds appropriately prefixed rules to the active rule set.
- flex also allows exclusive start conditions (declared using %x), which are sometimes more convenient.
  - %x scn

Example - Detecting multi-line comments using start conditions

Program File - start_conditions.l

- This program makes use of start conditions to detect the beginning and end of a multiline comment.

Commonly used Regular expressions in lex specification file

Table 1: Special Characters	
Pattern	**Matches**
.	any character except newline
\.	literal .
\n	newline
\t	tab
****	matches character \
^	beginning of line
$	end of line

Commonly used Regular expressions in lex specification file

Table 2: Operators	
Pattern	**Matches**
?	zero or one copy of the preceding expression
*****	zero or more copies of the preceding expression
+	one or more copies of the preceding expression
a|b	**a** or **b** (alternating)
(ab)+	one or more copies of **ab** (grouping)
abc	**abc**
abc*	**ab abc abcc abccc ...**
"abc*"	literal abc*
abc+	**abc abcc abccc abcccc ...**
a(bc)+	**abc abcbc abcbcbc ...**
a(bc)?	**a abc**

Commonly used Regular expressions in lex specification file

Table 3: Character Class	
Pattern	**Matches**
[abc]	one of: **a b c**
[a-z]	any letter a through z
[a\-z]	one of: **a - z**
[-az]	one of: **- a z**
[A-Za-z0-9]+	one or more alphanumeric characters
[\t\n]+	whitespace
[^ab]	anything except: **a b**
[a^b]	one of: **a ^ b**
[a|b]	one of: **a | b**

THANK YOU

Preet Kanwal

Department of Computer Science & Engineering

preetkanwal@pes.edu

PES UNIVERSITY ONLINE

Compiler Design

Preet Kanwal

Department of Computer Science & Engineering

Teaching Assistant : Kavya P K

Compiler Design

Unit 1

Design of a Lexical Analyzer Generator

Preet Kanwal

Department of Computer Science & Engineering

Compiler Design

Lecture Overview

PES UNIVERSITY ONLINE

In this lecture, you will learn about:

- Maximal Munch Rule
- Questions
- Lexical Analyzer with Lex
- The Structure of the Lex Analyzer Generator
- Constructing Automata - RE to NFA to DFA
- Example - Constructing Automata
- Lookahead Operator in Lex (/)
- Program - To identify keywords and identifiers
- Program - Demonstrate Maximal munch rule

Compiler Design

Behaviour of Lexer - Example 1

Given -

abb	printf(“1”)
aba	printf(“2”)
a	printf(“3”)

Questions:

- Provide output for the String : a
- Provide output for the String : ababa

How does a Lexer resolve ambiguities?

Lex follows the principle of **Maximal Munch rule.**

- When more than one pattern can match the input, lex chooses as follows:

1. The longest match is preferred.
2. Among rules that match the same number of characters, the rule that occurs earliest in the list is preferred.

Compiler Design

Example 1 - Solution

Given -

abb	printf(“1”)
aba	printf(“2”)
a	printf(“3”)

Questions:

- Provide output for the String : a
  - Answer - 3
- Provide output for the String : ababa
  - Answer - 2b3
    - aba - 2
    - b
    - a - 3

Recall Panic Mode Recovery

Compiler Design

Example 2

Given -

a*b	printf(“1”)
(a|b)*b	printf(“2”)
c*	printf(“3”)

Questions:

- Provide output for the String : cbabc
- Provide output for the String : cbbbbac
- Provide a String such that the Output is 132

Compiler Design

Example 2 - Solution

a*b printf(“1”)

(a|b)*b printf(“2”)

c* printf(“3”)

- Provide output for the String : cbabc
  - Answer - 323
    - c - 3
    - bab - 2
    - c - 3
- Provide output for the String : cbbbbac
  - Answer - 32a3
    - c - 3
    - bbbb - 2
    - a - a
    - c - 3
- Provide a String such that the Output is 132
  - Answer - abccbb
    - ab - 1
    - cc - 3
    - bb - 2
  - Other answers are possible

Compiler Design

Example 3

Given -

aa	printf(“1”)
b?a+b?	printf(“2”)
b?a*b?	printf(“3”)

Questions:

- Provide output for the String : bbbaabb
- Provide a String such that the Output is 123
- Provide a String such that the Output is 321

Compiler Design

Example 3 - Solution

aa printf(“1”)

b?a+b? printf(“2”)

b?a*b? printf(“3”)

- Provide output for the String : bbbaabb
  - Answer - 323
    - bb - 3
    - baab - 2
    - b - 3
- Provide a String such that the Output is 123
  - Not possible
- Provide a String such that the Output is 321
  - bbbabaa
    - bb - 3
    - bab - 2
    - aa – 1
  - Other answers are possible

Compiler Design

Example 4

Give an Example of such a set of regular expressions and an input string such that :

1. The String can be broken apart into substrings, where each substring matches one of the regular expressions BUT,

2. The longest-prefix algorithm will fail to break the string in a way where each piece matches one of the regex.

Compiler Design

Example 4 - Solution

Answer -

Suppose the patterns are as follows :

a* print(“1”)

ab print(“2”)

bb print(“3”)

Let the pattern be aabbb

Here, a matches 1, ab matches 2 and bb matches 3

However, due to maximal munch rule, the longest match is found, and

aa matches 1, bb matches 3 and last b is left unmatched. So output is 13b

Compiler Design

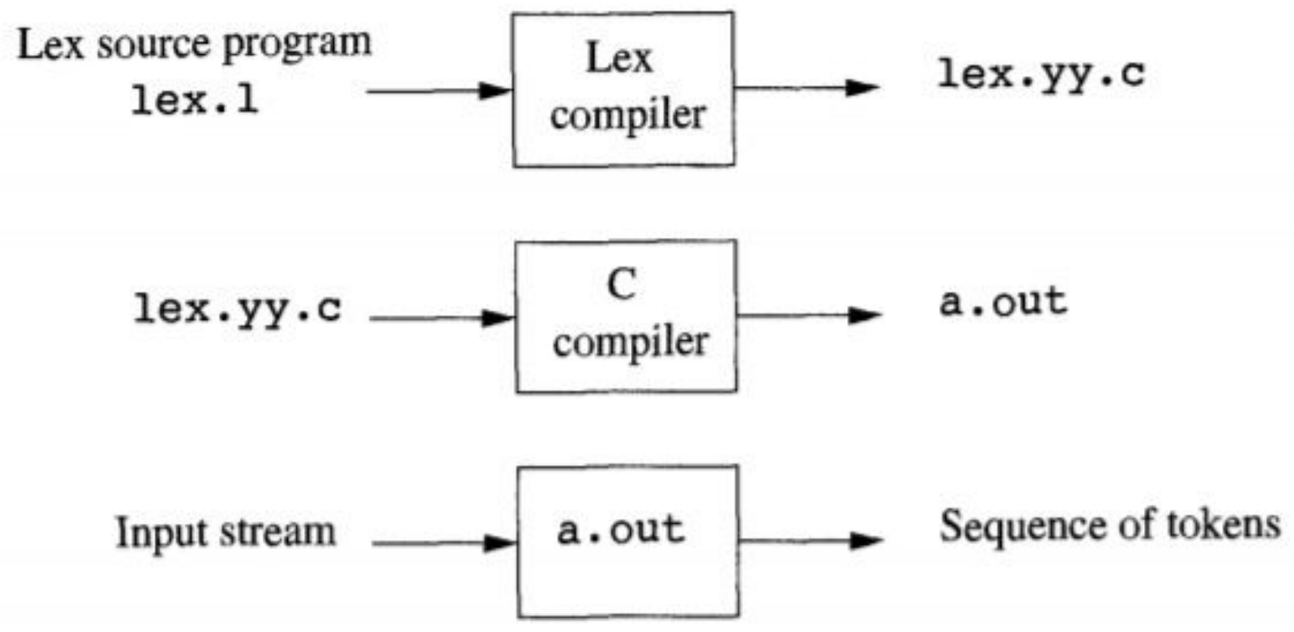
Verify the answers of the Example 1,2,3,4 using lex

PES UNIVERSITY ONLINE

Program Files -

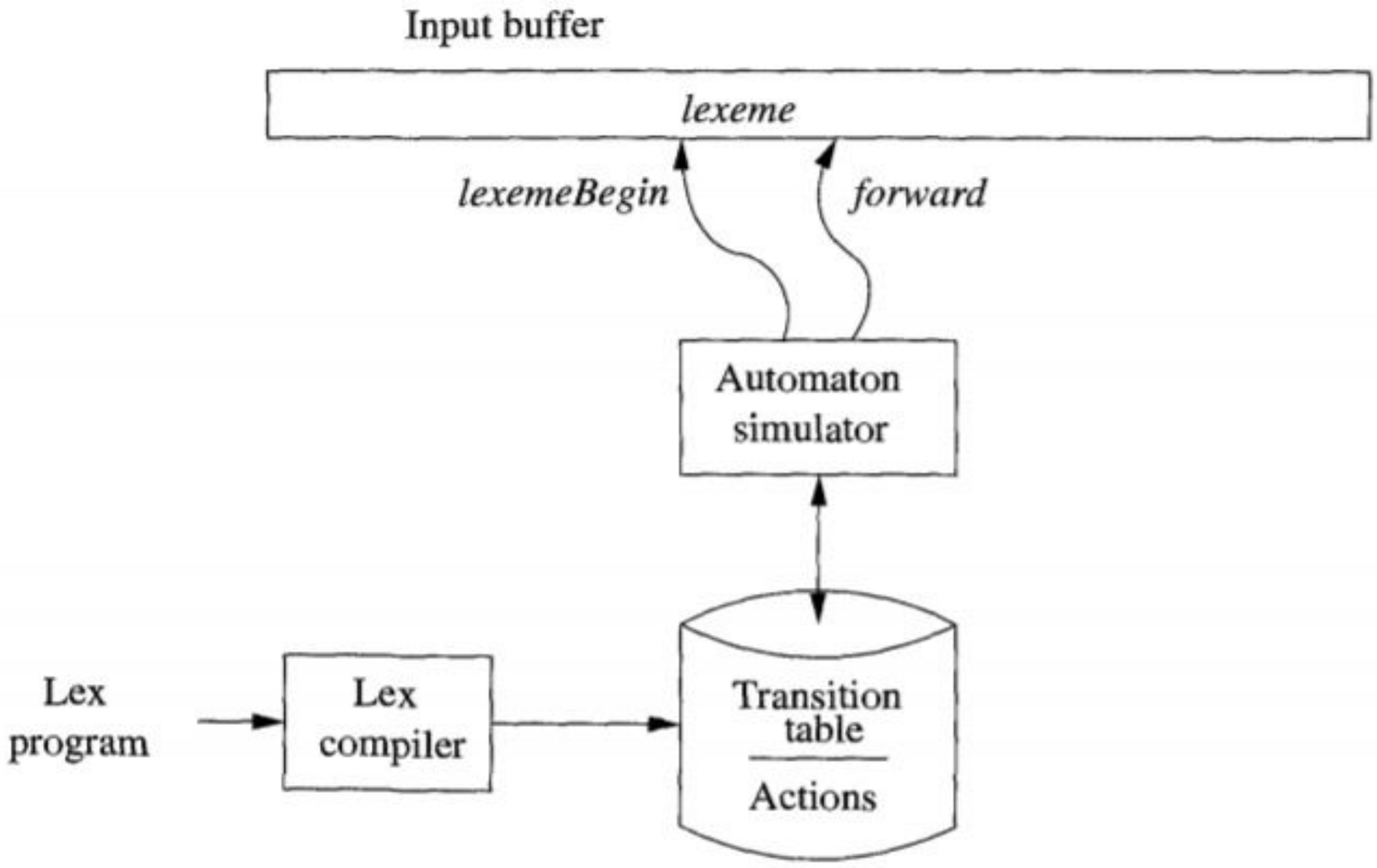
- **Example 1 - example1.l**
- **Example 2 - example2.l**
- **Example 3 - example3.l**
- **Example 4 - example4.l**
  - **In this program, lex shows a warning - "rule cannot be matched", because the first rule always matches the lexemes matched by the other 2 rules.**

Lexical Analyzer with Lex



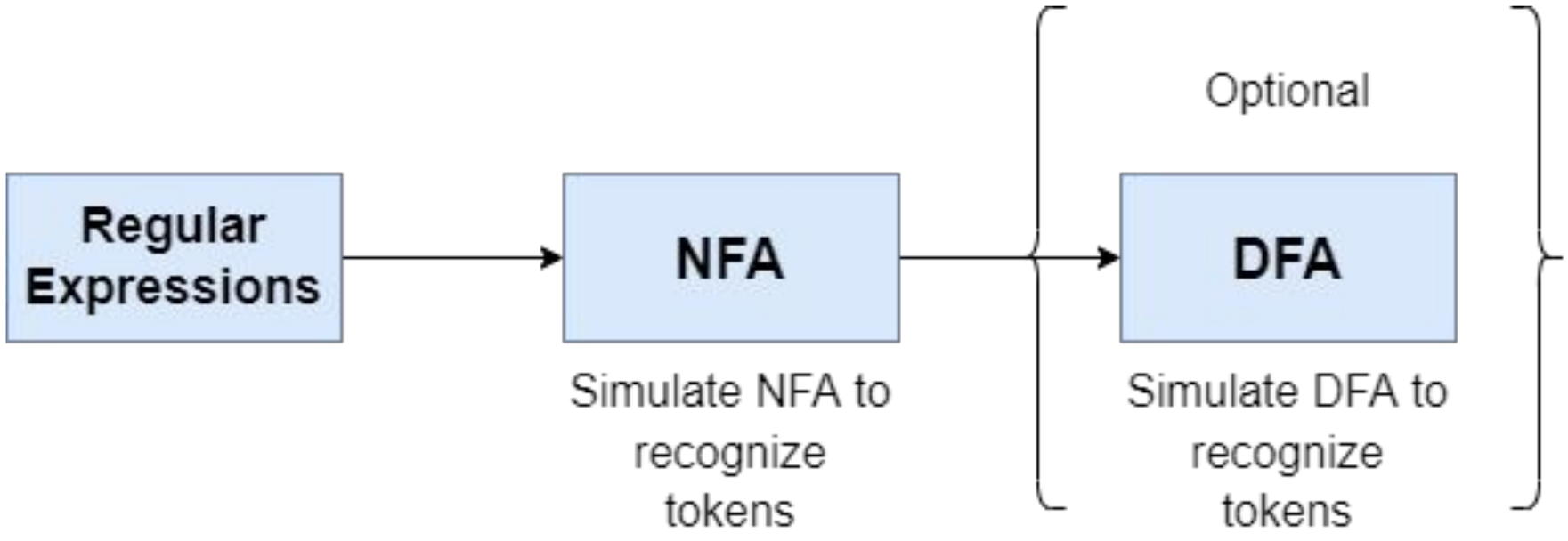
Compiler Design

The Structure of the Lex Analyzer Generator



Compiler Design

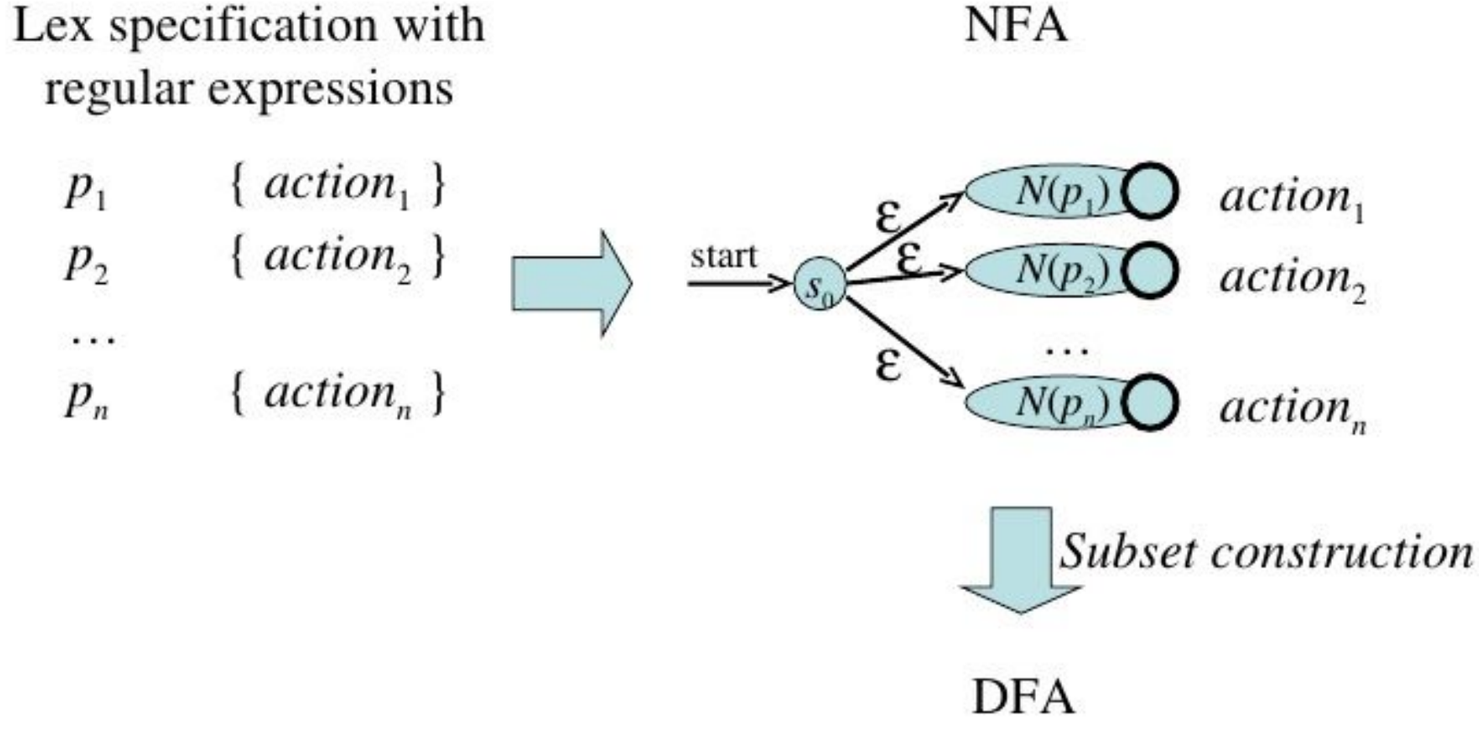
Constructing Automata - RE to NFA to DFA



Compiler Design

Constructing Automata - RE to NFA to DFA

- Convert each pattern (regular expressions) in the lex program to an NFA.
- Combine all NFAs into one using a new start state, with ε transactions to each of the start states of the NFAs.
- Convert the combined NFA to DFA.



Compiler Design

The Structure of the Lex Analyzer Generator

The generated lexical analyzer consists of components that are created from the lex specification file.

These components are -

- **Transition Table** - Created for all patterns defined in the lex program.
- **Actions** - Fragments of code defined to their corresponding patterns.
  - Actions are invoked by the Automata Simulator at the appropriate time.
- **Functions** - Defined in the auxiliary functions section of the lex program, these are passed directly through lex to the output file.
- **Automata Simulator** - The program that serves as the lexical analyzer and uses the components mentioned above.
  - It simulates an automata (NFA or DFA)

Compiler Design

Question 1- Construct an automata for the lex program

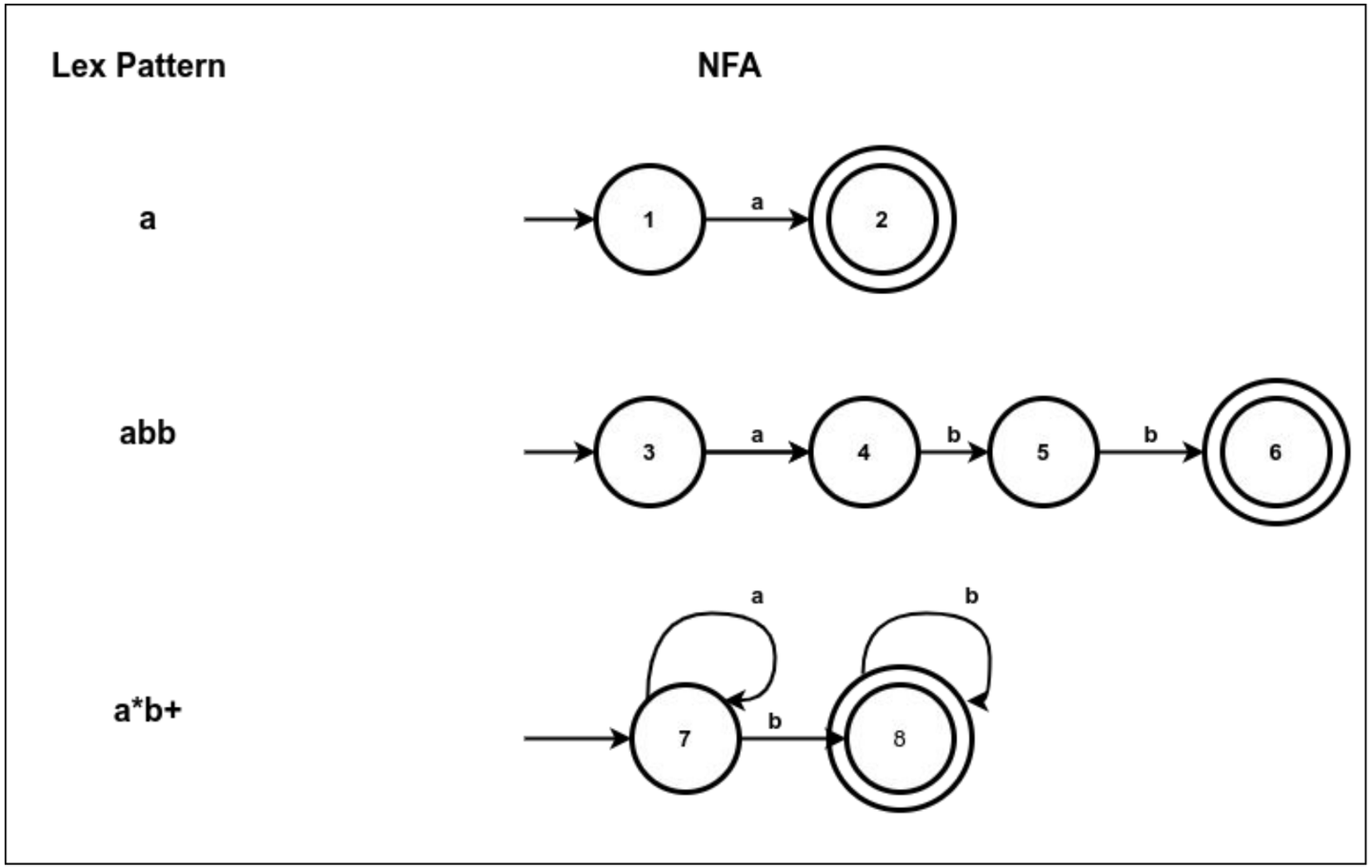
Given patterns and corresponding actions -

a	A1
abb	A2
a*b+	A3

Compiler Design

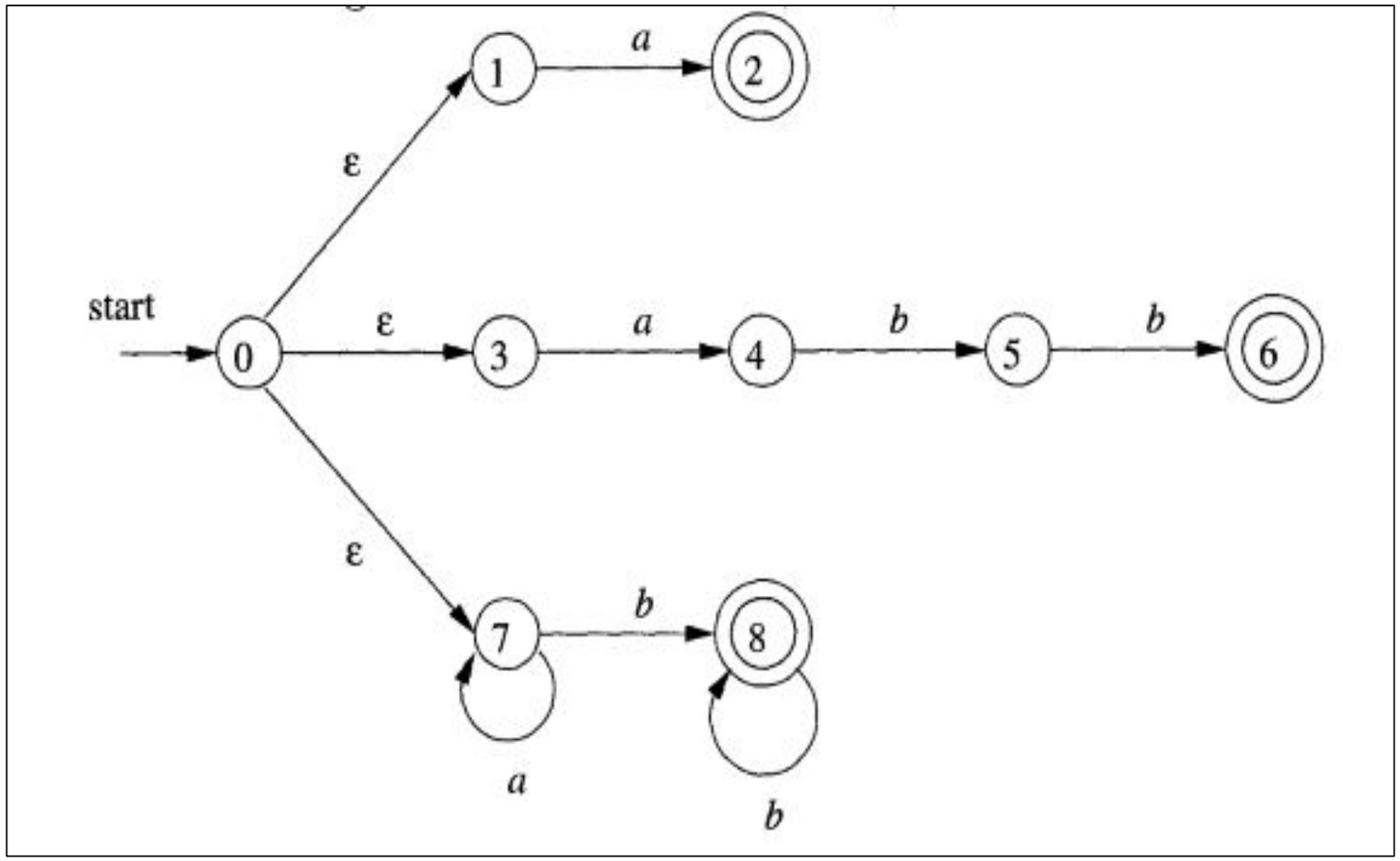
Question 1- Construct an automata for the lex program

Step 1 - Convert each lex pattern to an NFA



Question 1- Construct an automata for the lex program

Step 2 - Construct a combined NFA by adding a new start state, and ε transactions to each start state (1,3,7)

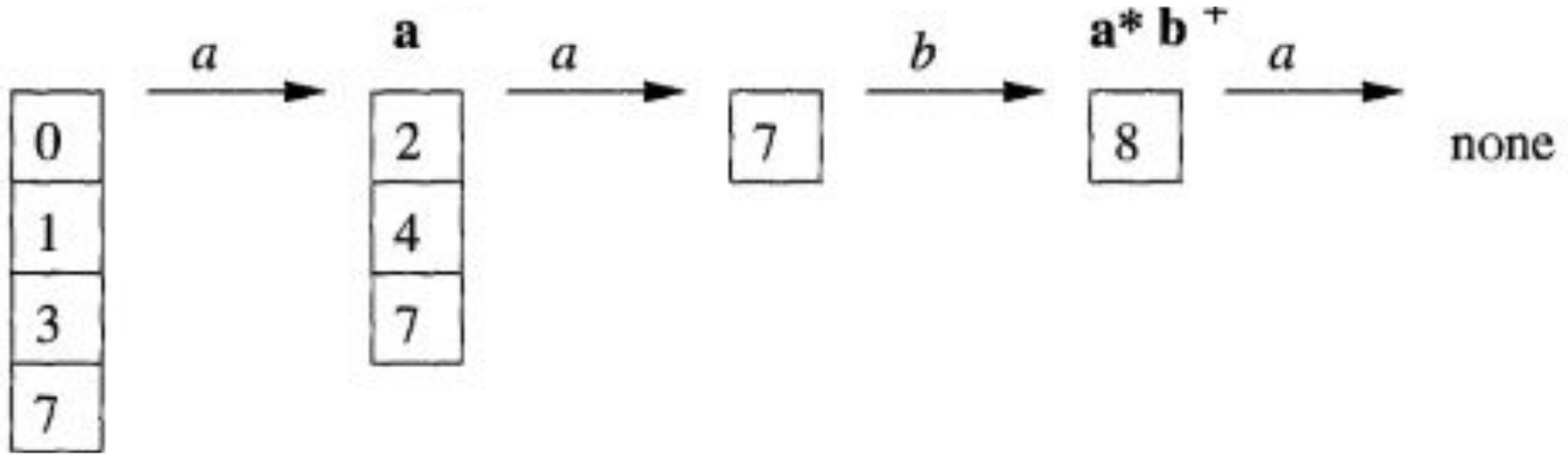


Compiler Design

Question 1- Construct an automata for the lex program

If the Lexical analyzer simulates the above NFA, pattern matching is done as follows -

- Let the input lexeme be aaba
- Initially, the combined NFA reads the input character pointed by lexeme_begin.
- The forward pointer moves ahead. Set of states is calculated at each point.
- When the NFA simulation reaches a point in the input where there are no next states, the set of states is empty (none).

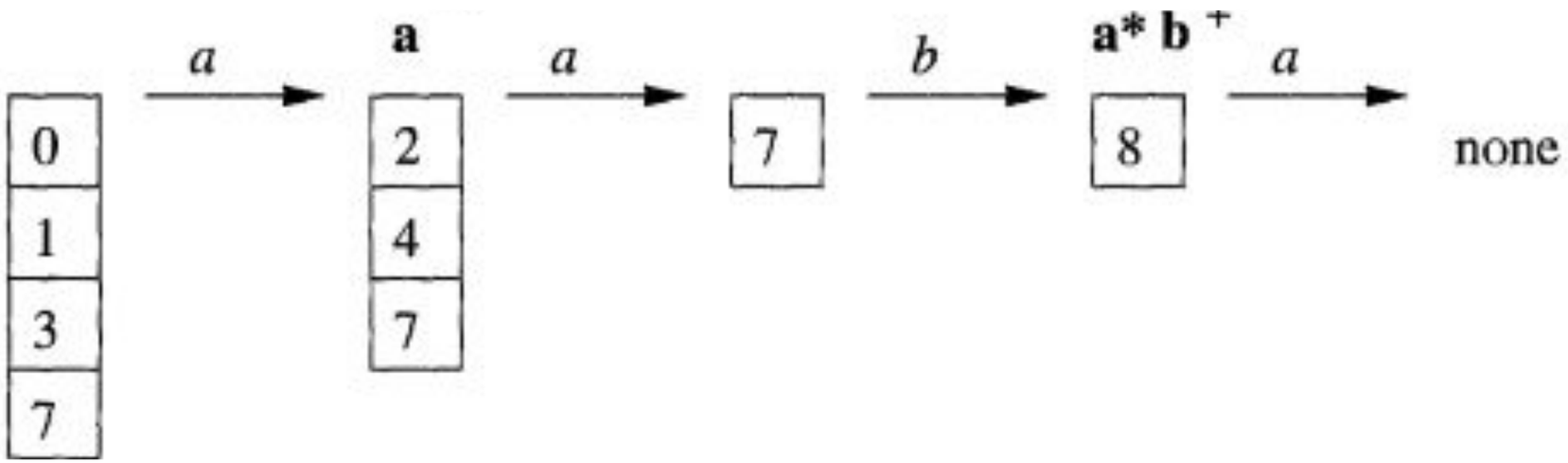


Sequence of states entered while processing input aaba

Compiler Design

Question 1- Construct an automata for the lex program

- At this point, the longest prefix that matches the NFA can be identified,
- As indicated in the diagram, in state 8, **aab** matches the pattern **a*b**.
- Prefix **aab** is the longest prefix, and so it executes action **A3**.
- The forward pointer is movies to the end of the lexeme.
- Note - If there are several accepting states in the set, the action associated with the earliest pattern is executed.

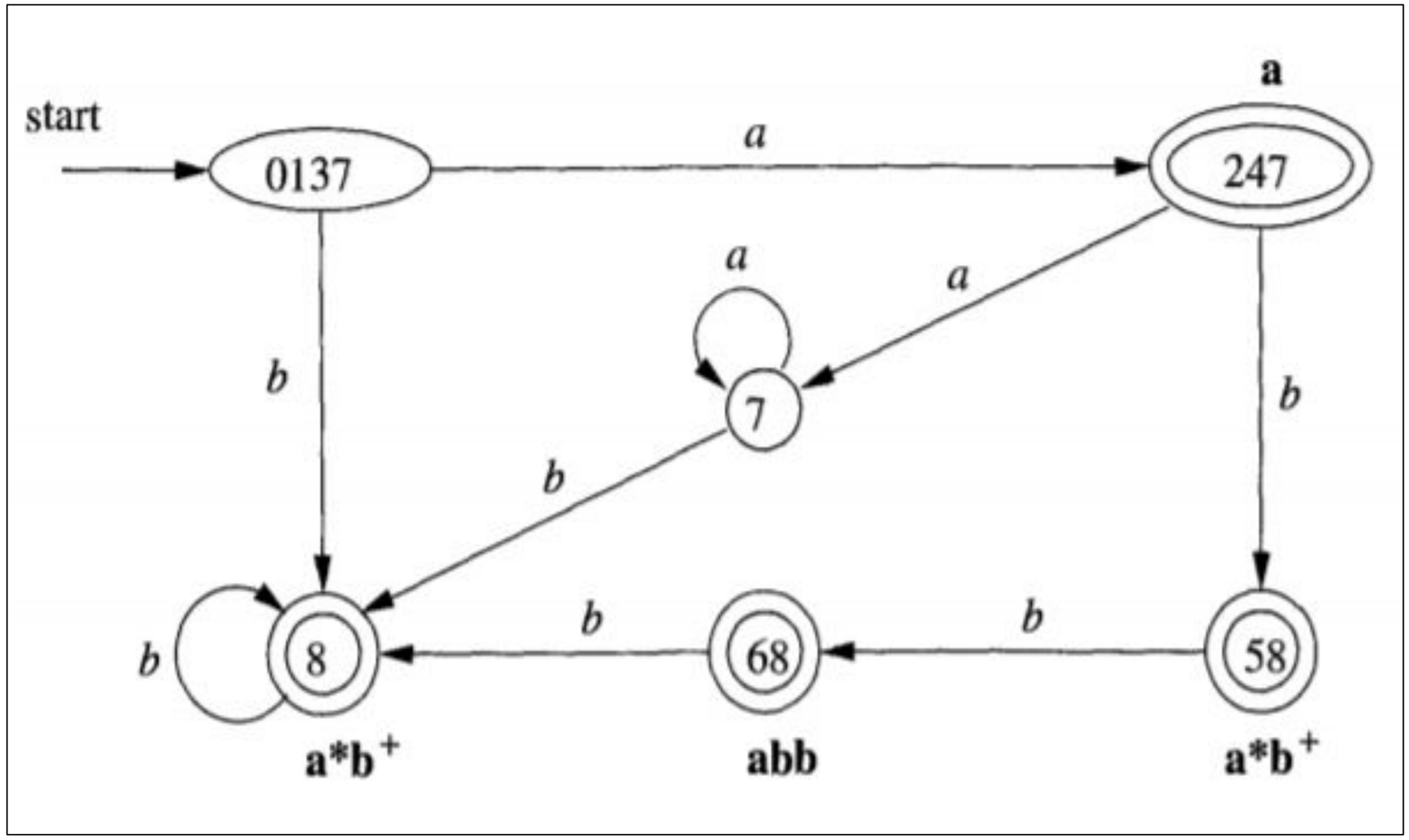


Sequence of states entered while processing input aaba

Compiler Design

Question 1- Construct an automata for the lex program

Step 3 - Convert NFA to DFA

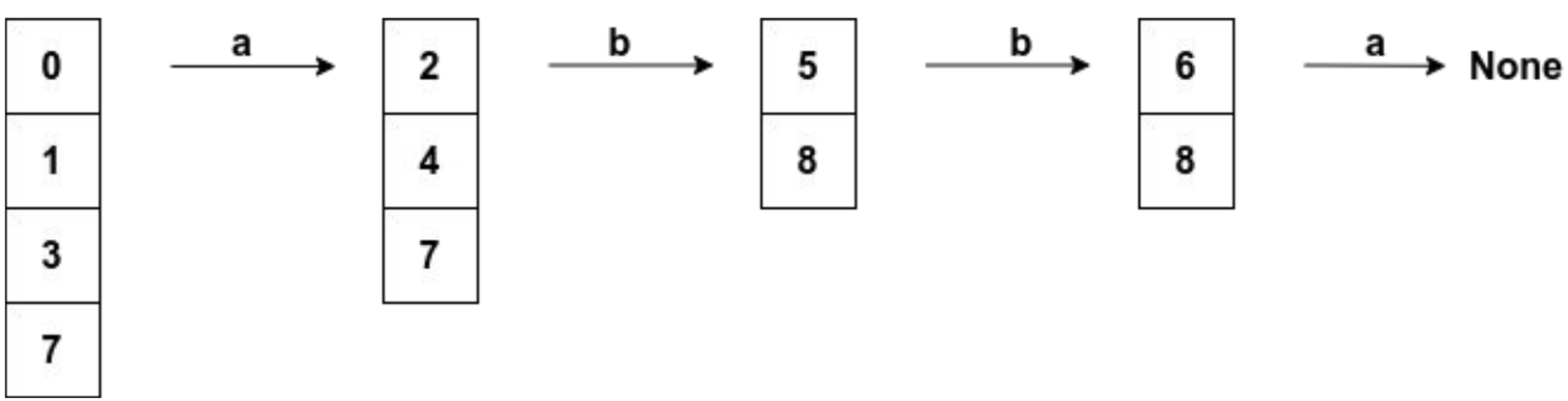


Compiler Design

Question 1- Construct an automata for the lex program

If the Lexical analyzer simulates the above DFA, pattern matching is done as follows -

- Let the input lexeme be abba
- Set of states is calculated at each point.
- When the DFA simulation reaches a point in the input where there are no next states, the action associated with the last accepting state is executed.
- The lexeme is abb, and the accepting state is 68
- This matches pattern abb, and action A2 is executed



Sequence of states entered while processing input abba

Compiler Design

Question 2- Construct an automata for the lex program

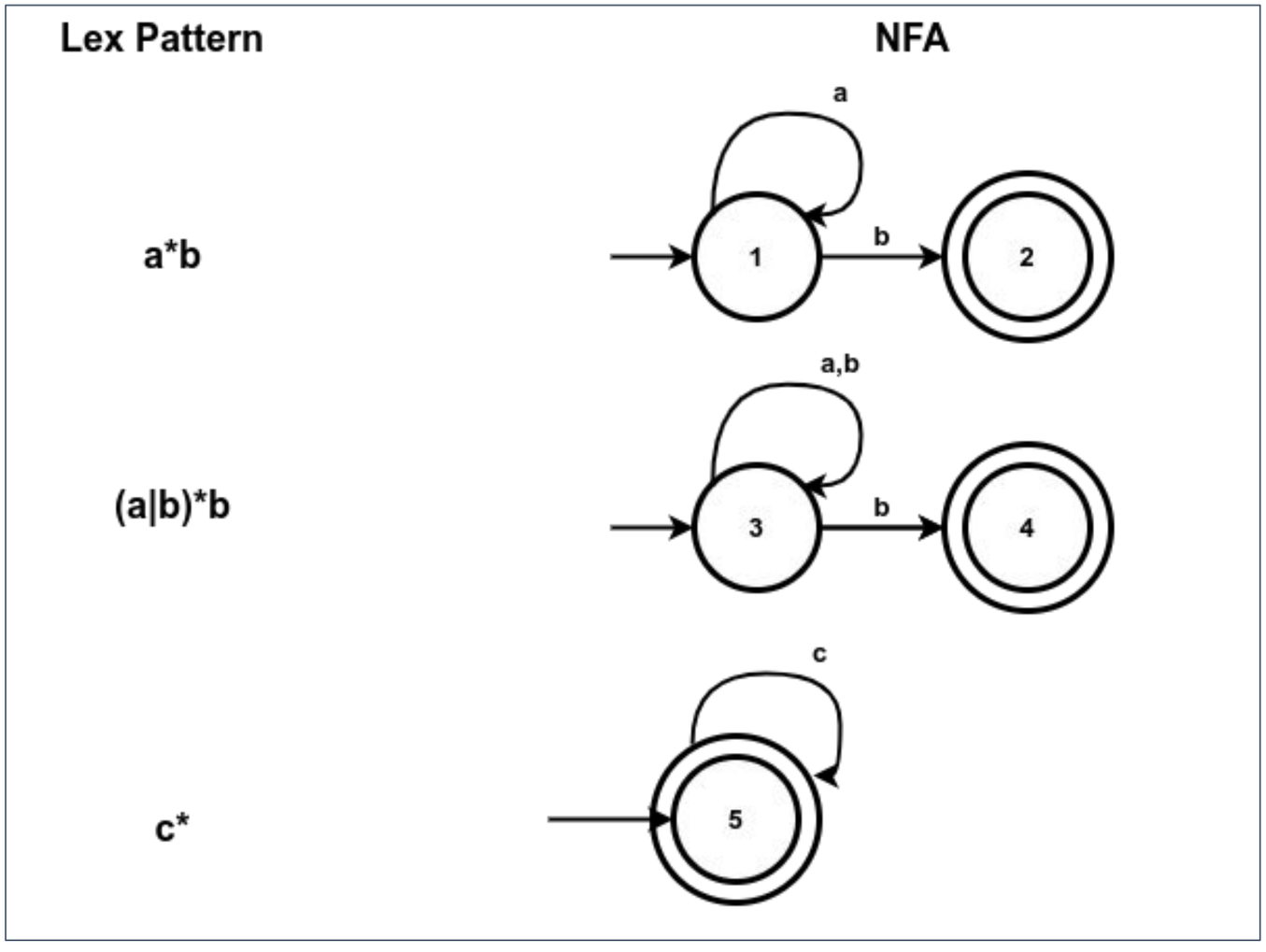
Given patterns and corresponding actions -

a*b	A1
(a|b)*b	A2
c*	A3

Compiler Design

Question 2- Construct an automata for the lex program

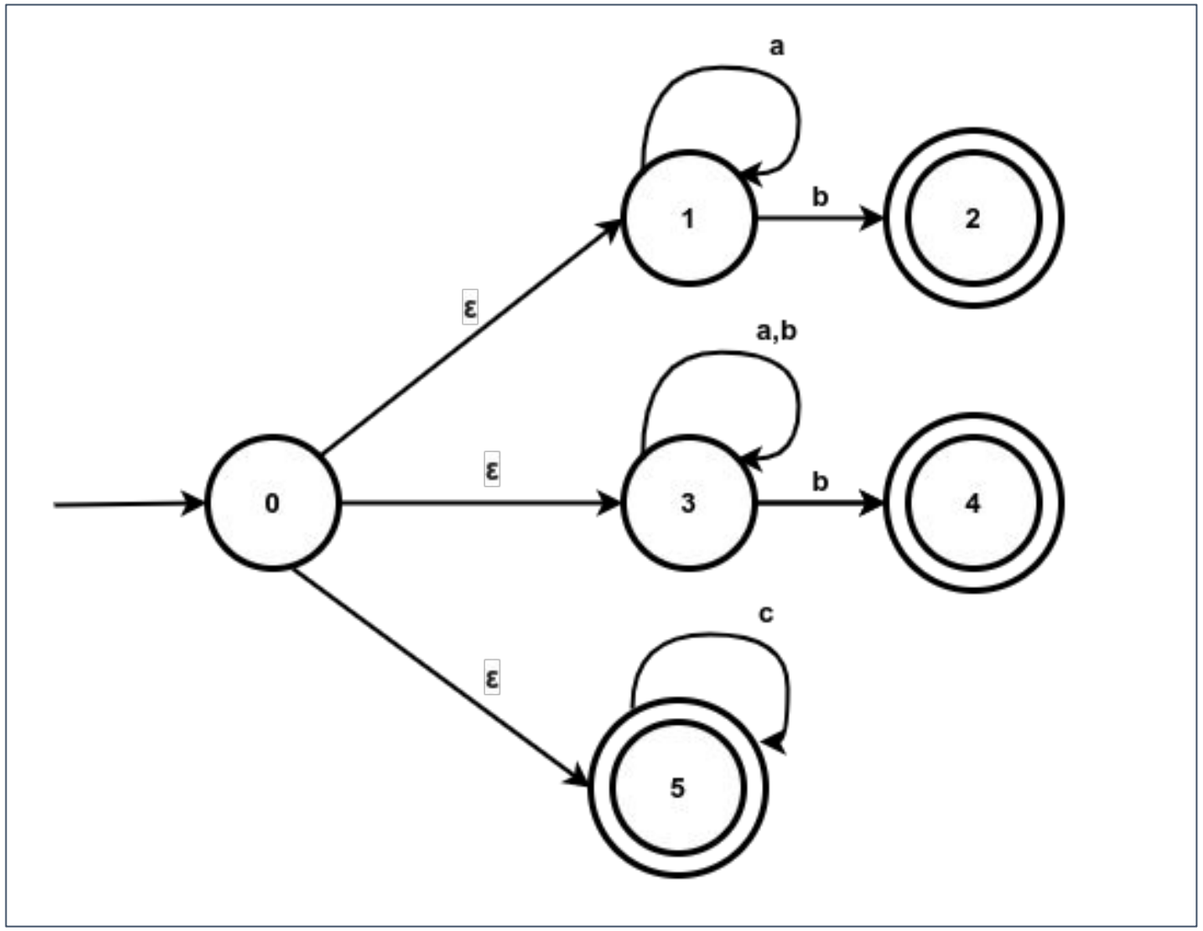
Step 1 - Convert each lex pattern to an NFA



Compiler Design

Question 2- Construct an automata for the lex program

Step 2 - Construct a combined NFA by adding a new start state, and ε transactions to each start state (1,3,5)

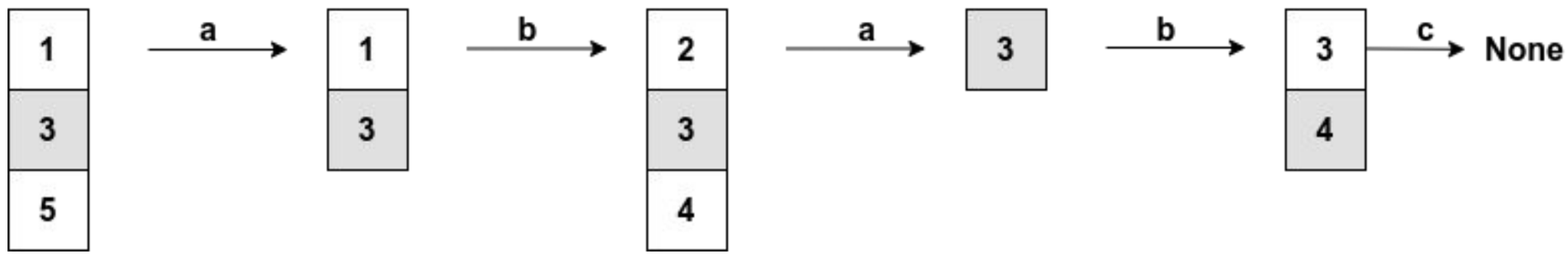


Compiler Design

Question 1- Construct an automata for the lex program

If the Lexical analyzer simulates the above NFA, pattern matching is done as follows -

- Let the input lexeme be **ababc**
- Initially, the combined NFA reads the input character pointed by **lexeme_begin**.
- The **forward** pointer moves ahead. Set of states is calculated at each point.
- When the NFA simulation reaches a point in the input where there are no next states, the set of states is empty (none).

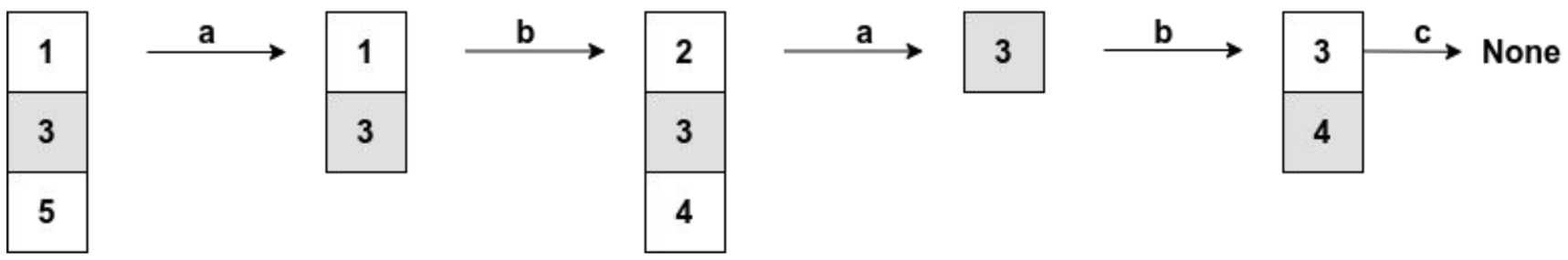


Sequence of states entered while processing input ababc

Compiler Design

Question 1- Construct an automata for the lex program

- At this point, the longest prefix that matches the NFA can be identified,
- As indicated in the diagram, in state 8, **abab** matches the pattern **(a|b)*b**.
- Prefix **abab** is the longest prefix, and so it executes action **A2**.
- The forward pointer is movies to the end of the lexeme.
- Note - If there are several accepting states in the set, the action associated with the earliest pattern is executed.

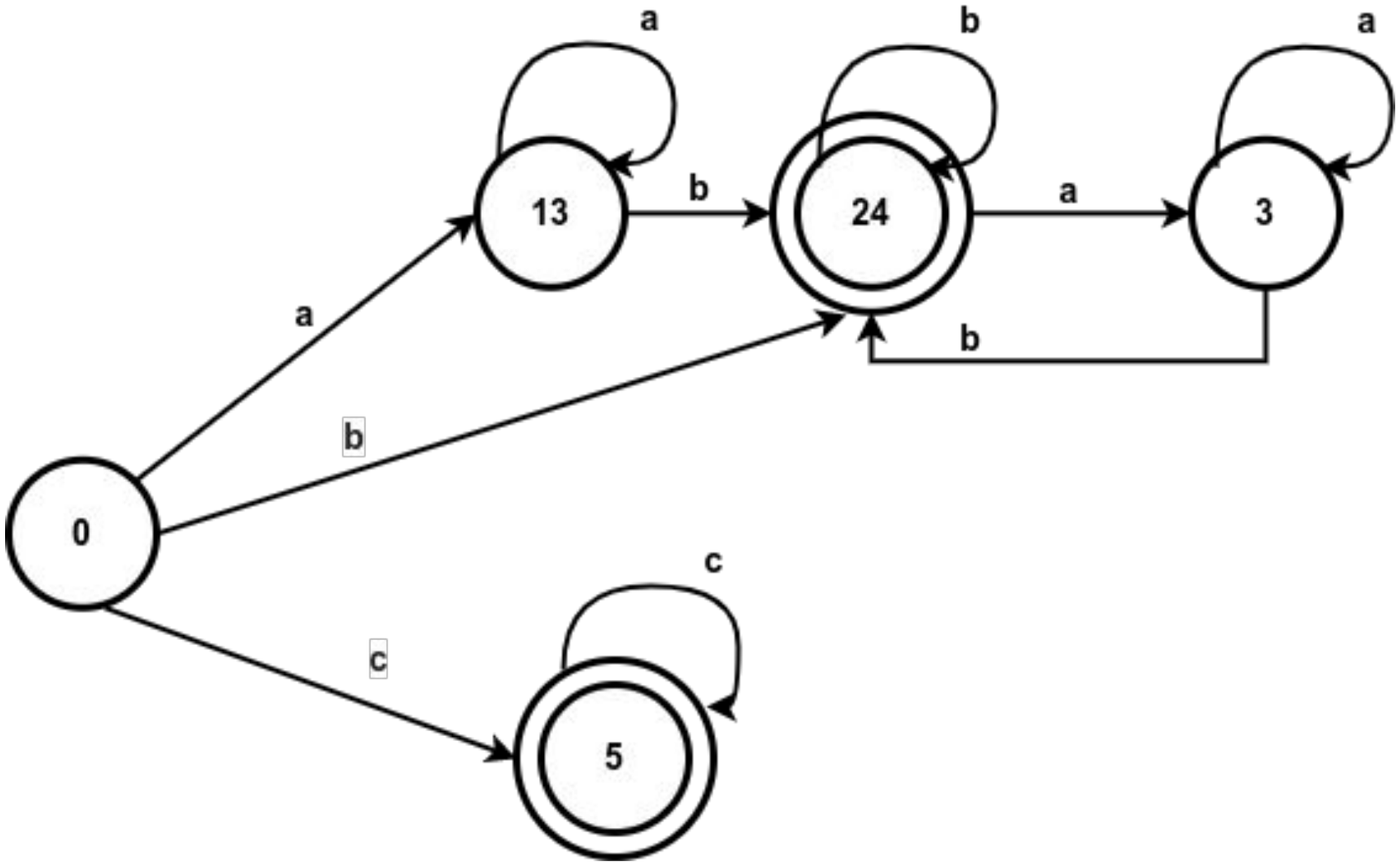


Sequence of states entered while processing input aabac

Compiler Design

Question 1- Construct an automata for the lex program

Step 3 - Convert NFA to DFA

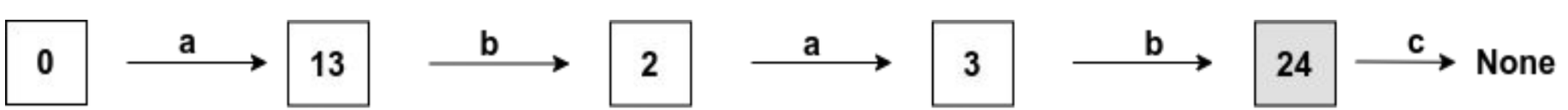


Compiler Design

Question 1- Construct an automata for the lex program

If the Lexical analyzer simulates the above DFA, pattern matching is done as follows -

- Let the input lexeme be **ababc**
- Set of states is calculated at each point.
- When the DFA simulation reaches a point in the input where there are no next states, the action associated with the last accepting state is executed.
- The lexeme is **abab**, and the accepting state is 24
- This matches pattern **(a|b)*b**, and action **A2** is executed



Sequence of states entered while processing input ababc

Compiler Design

Lookahead Operator in Lex (/)

- When a certain pattern is required to be matched only when it is followed by certain other characters, the lookahead operator is / used
- r1 / r2
  - r1 - the pattern that matches the lexeme
  - r2 - the additional pattern that must be matched before deciding the token of the lexeme. Note - the characters that match r2 is NOT part of the lexeme.
- Example - FORTRAN IF statement of the from IF(condition) THEN …
- In FORTRAN, keywords are not reserved

Compiler Design

Lookahead Operator in Lex (/)

- IF is always followed by
  - Left parenthesis (
  - Some text
  - Right parenthesis)
  - Any letter
- The lex rule can be written as

  IF/\(.*\){letter}
- This lex code demonstrates use of / to detect a FORTRAN IF

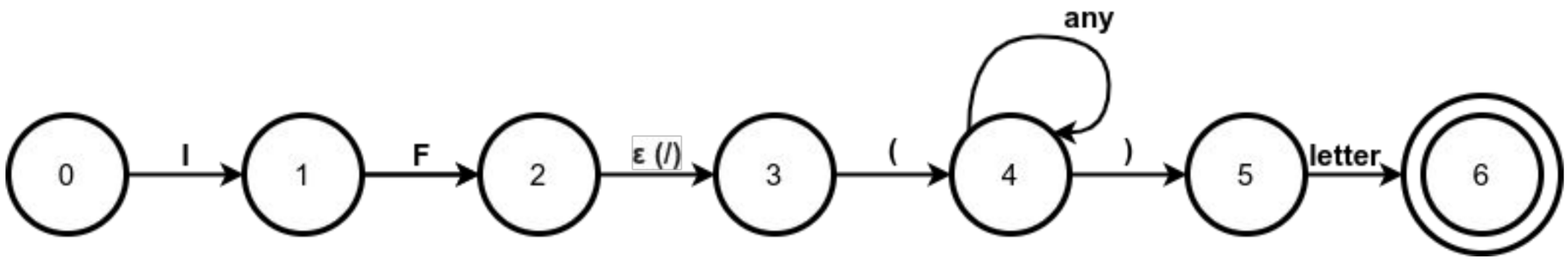
```
letter    [A-Za-z]
%{
#include<stdio.h>
%}
%%
IF/\(.*\){letter}   printf("FORTRAN IF");
. ;
%%
int yywrap(){return(1);}
int main()
{
   yyin = fopen("input.txt", "r");
   yylex();
   fclose(yyin);
   return 0;
}
```

Program file : lookahead.l

Compiler Design

Implementation of Lookahead Operator

- When converting to NFA, the lookahead operator (/) is treated as an ε transaction.
- The below diagram represents NFA for FORTRAN IF using lookahead
- The ε transaction from state 2 to state 3 represents the lookahead operator.
- State 6 indicates presence of IF
- The lexeme IF is found by scanning backwards to the last occurrence of state 2 whenever state 6 is reached.
- Note - If the NFA has more than one ε transaction on / , then the problem of finding the correct state (e.g, state 2 in this example) is difficult.



Compiler Design

Write the lex file to identify identifiers and keywords

```
digit    [0-9]
letter   [A-Za-z]
%{
#include <stdio.h>
}%
%%
if|else|int|char|float    {printf("Keyword :\t %s",yytext);
{letter}({letter}|{digit})*    printf("Valid Identifier:\t %s",yytext);
. ;
%%
int yywrap(){return(1);}
int main() {
        yyin = fopen(“input.txt”, "r");
        yylex();
        fclose(yyin);
}
```

Program file : keywords_and_identifiers.l

The rule
. ;
is used to ignore all other characters.

THANK YOU

Preet Kanwal

Department of Computer Science & Engineering

preetkanwal@pes.edu

PES UNIVERSITY ONLINE

Compiler Design

Preet Kanwal

Department of Computer Science & Engineering

Teaching Assistant : Kavya P K

Compiler Design

Unit 1

Specification and Recognition of Tokens

Preet Kanwal

Department of Computer Science & Engineering

Compiler Design

Lecture Overview

In this lecture, you will learn about:

- Revision :
  - What are tokens?
  - Classes of tokens
  - Attributes
- Implementation of Lexer:
  - Hand-written Implementation
  - Using a tool like: lex, PLY, CUP(in Java)
- Handwritten Implementation:
  - Specification of Tokens
  - Recap - Regular expressions and Transition diagrams
  - Implementation of Transition Diagrams
  - How does the Lexical Analyzer implement all transition diagrams?
- Questions

Some important definitions

Recall that

Token :- It is a pair consisting of a token name and an optional attribute value.

<token name, attribute value>

- **Token name is an abstract symbol representing a kind of lexical unit - keywords, identifiers etc.**
- **Attribute value is pointer to the symbol table entry.**

Examples

Token	Informal Description of Pattern	Lexemes
if	Character i,f	If
else	Characters e,l,s,e	else
Comparison	< or > or <= or >= or == or !=	<=,!=
id	Letter followed by letters and digits	Sum, avg1, pi
number	Any numeric constant	10001, 3.14
literal	Anything but “, surrounded by “	“compiler design”

Compiler Design

Tokens

- Tokens are required to classify substrings of source program according to their role.
- The parser relies on token distinctions.
- get next token is the command sent from the Parser to the Lexical analyzer.
- On receipt of the command, the lexical analyzer scans the input until it determines the next token, and returns it.
- Input is read left to right.
- Lookahead is required to decide where one token ends.

Compiler Design

Classes of Tokens

Choosing good tokens is important, and is mostly language specific.

The most common classes of tokens are listed below:

- **Keywords** - One token for each keyword. The pattern for a keyword is the same as the keyword itself.
- **Operators** - Tokens either individually, or in classes for operators.
- **Identifiers** - One token representing all identifiers.
- **Constants** - One or more tokens representing constants, such as numbers and literal strings.
- **Punctuation Symbols** - Tokens for each punctuation symbol, such as left and right parentheses, comma and semicolon.

Compiler Design

Attributes of Tokens

- When more than one lexeme can match a pattern, lexical analyzer must provide additional information about each lexeme matched.
- This additional information is stored in the attribute-value field of the token.
- The attribute-value field can include several pieces of information.
- Consider the example : token ID (an identifier)
  - Its attributes are the lexeme, the type and the location at which it is first found. This is added to the symbol table.
  - The attribute value is the pointer to the symbol table entry for that identifier.
- Lexical analyzer returns token name and attribute value to parser.
- The token name influences parsing decisions, whereas the attribute value influences translation of tokens after parsing.

Compiler Design

Implementation of Lexer

There are two ways :

- **Hand-written Implementation** : One will write the lexer from scratch as a program. For example : The C compiler is hand-written. It is basically called loop-switch implementation.
- **Using a tool**: lex, PLY

Hand-written Implementation

Compiler Design

Specification of Tokens

The specification of tokens is done using formal notations, i.e, regular expressions.

Some important definitions -

- Symbol : A letter, digit or punctuation
- Alphabet : A finite set of symbols, for e.g, {0,1} is a Binary alphabet
- String over an alphabet : A finite set of symbols drawn from the alphabet
- If **s** is the string, then **|s|** denotes the length of the string
- Language - A set of strings over a fixed alphabet
- Abstract language - An empty set, {e}

Compiler Design

Recap - Regular Expressions

- Regular expressions are special text strings that describe a search pattern.
- It is mainly used for pattern matching.
- It is a metalanguage - a form of language used for description or analysis of another language.
- It uses metacharacters - characters that have special meaning. For eg -
  - * matches any number of characters
  - . matches single characters except newline
  - \ drops the special meaning of the next character
- A Regular language is a formal language that can be expressed using a regular expression.
- Regular expressions are the grammatical notations used for describing structure of tokens, i.e, patterns.

Compiler Design

Regular Definitions

- For notational convenience, names can be given to regular expressions, and these names can be used to define other regular expressions.
- This is called a regular definition.
- Example 1 : Regular definition for Identifiers

  letter -> [a-zA-Z_]

  digit -> [0-9]

  id -> letter(letter|digit)*
- Example 2: Regular definition for whitespace

  blank -> " "

  tab -> \t

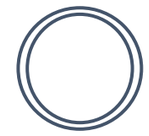
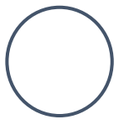
  newline -> \n

  ws -> (blank|tab|newline)+

Compiler Design

Recap : Transition Diagrams

- Patterns are converted to stylized flowcharts called Transition Diagrams to understand how a pattern traverses a given input.
- Transition diagrams consist of a collection of nodes called states.
- Each state represents a condition that summarises the characters seen.
- Two types of states -

1. Final or Acception State 
   a. Indicates that a lexeme has been found.
   b. An action is attached to this state - returning the token and attribute value to the parser.
   c. If necessary to retract the forward pointer by one position, a * is placed next to the state. 
2. Start State 

Compiler Design

Implementing Transition Diagrams

- A variable state holds current state number
- A switch based on the value of state executes the code for that state.
- The following functions are used -
  - getc() or nextChar() - reads the next character from input.
  - install_id() - places the lexeme (identifier) in the symbol table if it's not present and returns a pointer to the same.
  - install_num() - similar to install_id(), enters the number in the table of numbers if not present, and returns a pointer to the same.
  - retract() - If the accepting state has a * , this function is used to retract forward pointer.

Implementing Transition Diagrams

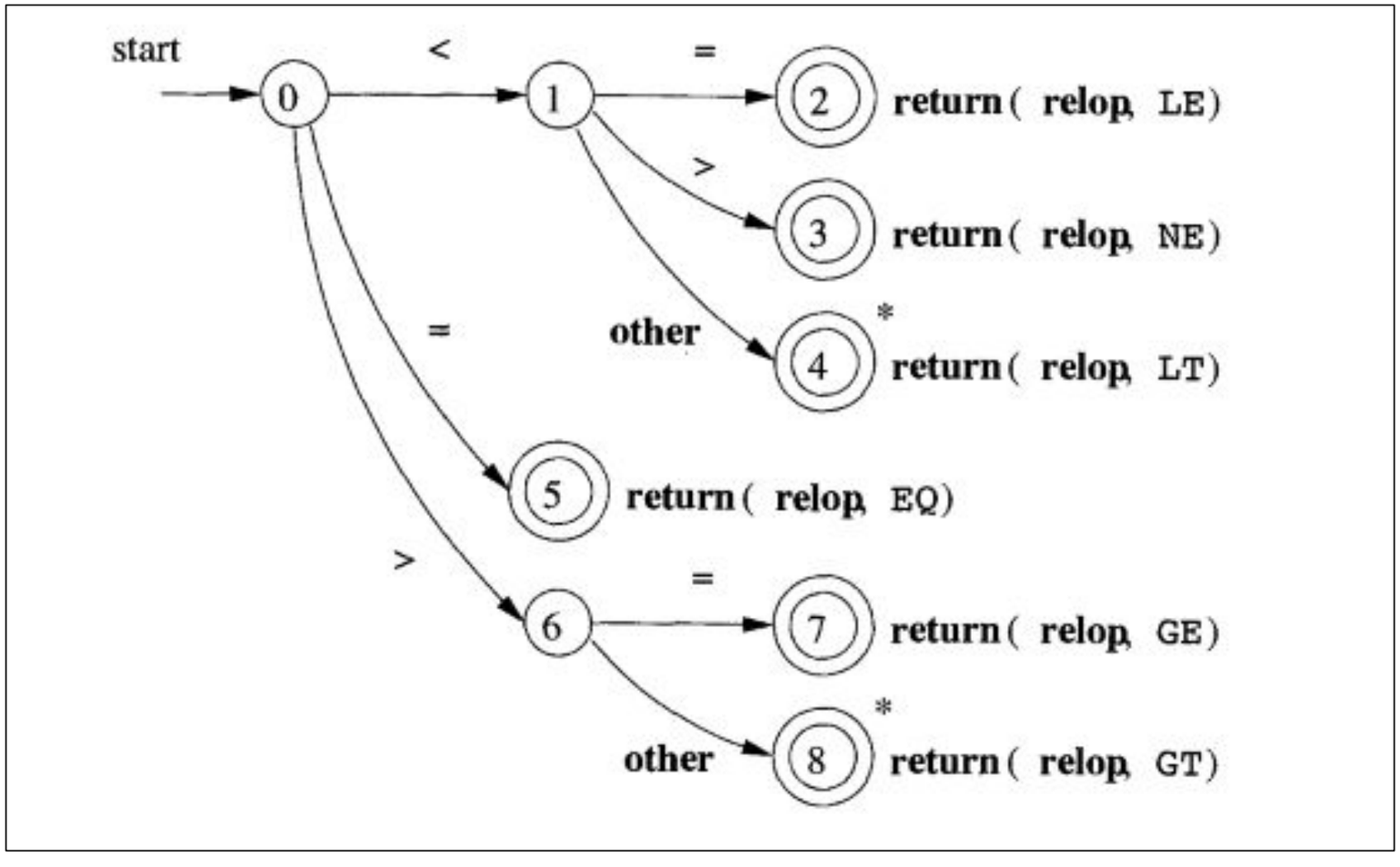
- **getToken()** - examines the symbol table entry for the lexeme found and returns the corresponding token name.
- **fail()** - resets forward ptr to lexeme_begin to try another transition diagram.
  - What the **fail()** function does depends on the global error recovery strategy of the Lexical Analyzer
- **isalpha(c)** - returns true iff **c** is an alphabet
- **isdigit(c)** - returns true iff **c** is a digit
- **isalnum(c)** - returns true iff **c** is an alphabet/digit
- **isdelim(c)** - returns true iff **c** is a delimiter

Compiler Design

Transition Diagram for Relational operators

Regular Expression -

relop < | < = | > | > = | <> | ==



Compiler Design

Implementation - Relational Operators

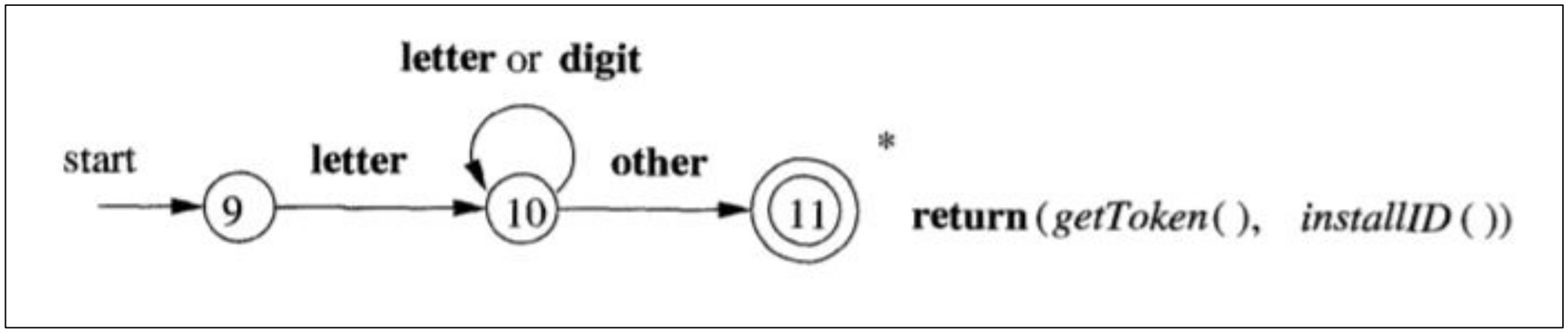
```
TOKEN getRelop()
{
    TOKEN retToken = new(RELOP);
    while(1) { /* repeat character processing until a return
                  or failure occurs */
        switch(state) {
            case 0: c = nextChar();
                    if ( c == '<' ) state = 1;
                    else if ( c == '=' ) state = 5;
                    else if ( c == '>' ) state = 6;
                    else fail(); /* lexeme is not a relop */
                    break;
            case 1: ...
            ...
            case 8: retract();
                    retToken.attribute = GT;
                    return(retToken);
        }
    }
}
```

Compiler Design

Transition Diagram for Identifiers

Regular Expression -

letter [a-zA-Z]
digit [0-9]
letter ([letter|digit] | ”_”)*



Compiler Design

Implementation for Identifiers

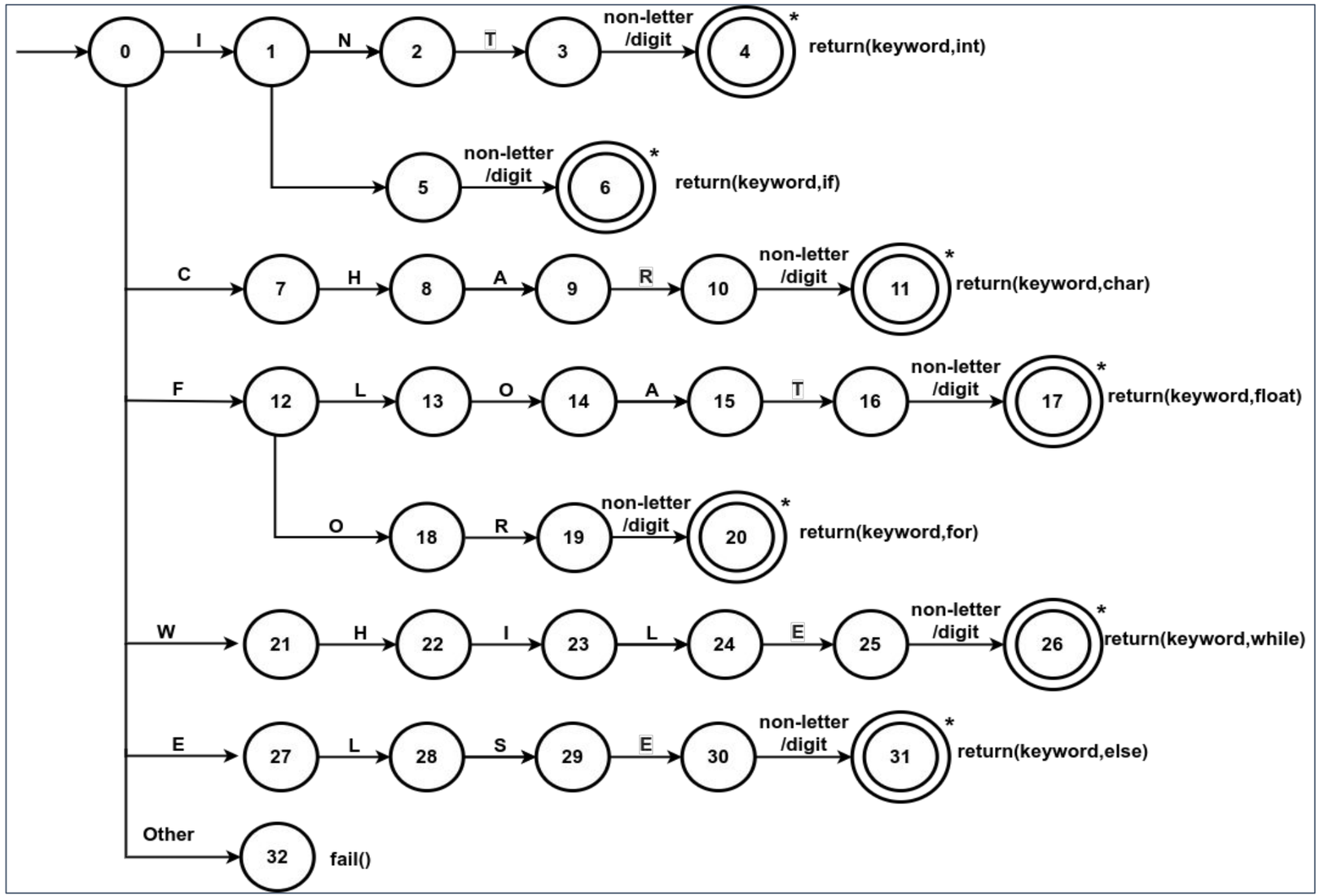
For pseudo-code, please refer to the file:
identifier_loopswitch_pseudocode.c.txt

Note: This file is only a conversion of the transition diagram from the previous slide and is not executable without a proper interface and implementations of other functions like retract(), fail(), nextChar(), etc.

Compiler Design

Transition Diagram for Keywords

int | char | float | if | while | else | for



non-letter/digit simply means that the transition should be over a non-letter, non-digit character

Compiler Design

Implementation for Keywords

- For pseudo-code, please refer to the file:
  keyword_loopswitch_pseudocode.c

  Note: This file is only a conversion of the transition diagram from the previous slide and is not executable without a proper interface and implementations of other functions like retract(), fail(), nextChar(), etc.

- For code that combines identification of keywords and identifiers, please refer to the file:
  loopswitch.c

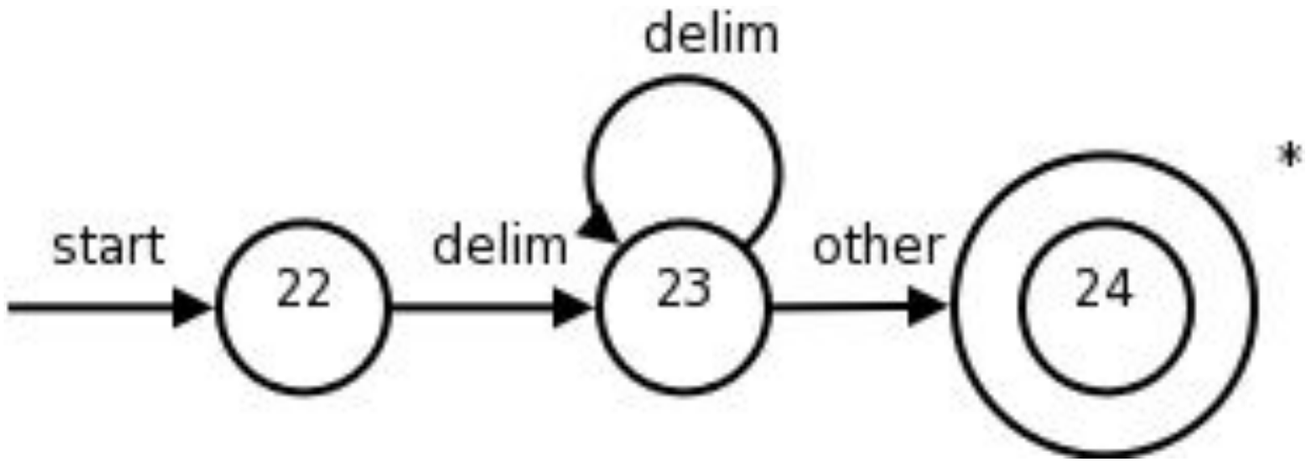
  This code is an altered version of the code from:
  https://gist.github.com/luckyshq/6749155

Compiler Design

Transition Diagram for Whitespaces

Regular Expression -

blank " "
tab "\t"
newline "\n"
ws -> (blank | tab | newline)+



Here, the input is retracted to begin at a non-whitespace

There is no return to parser.

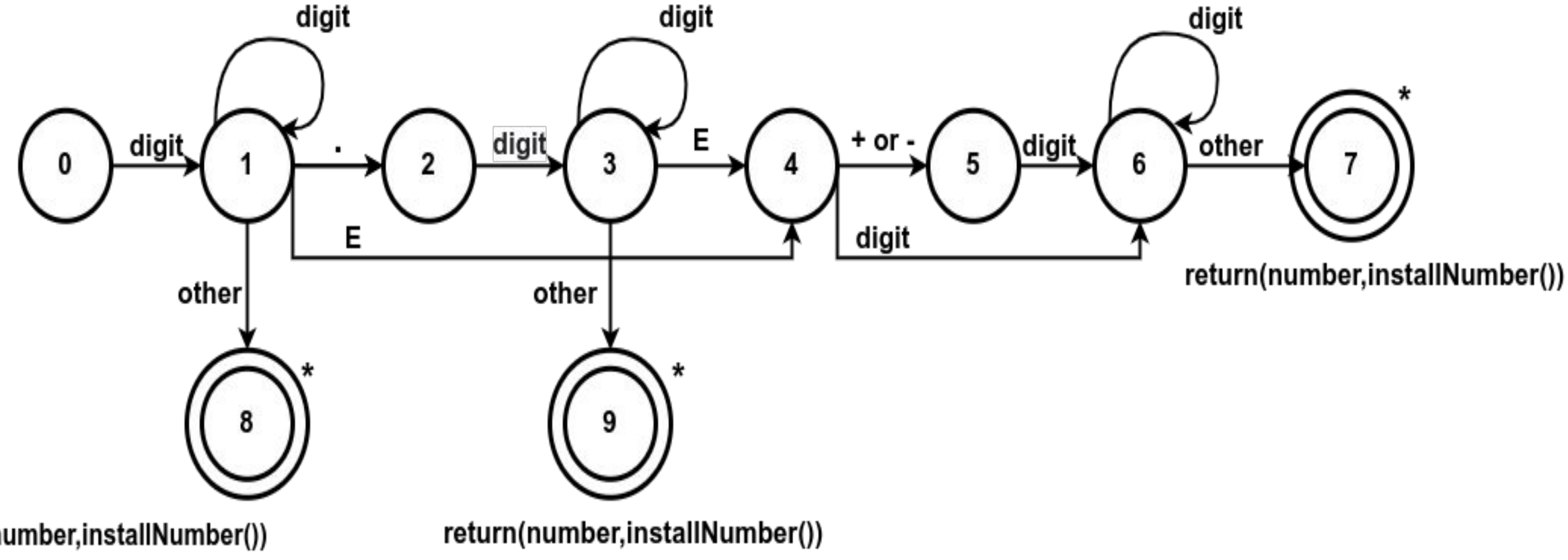
The process of lexical analysis is restarted after whitespace

Compiler Design

Transition Diagram for Unsigned numbers

Regular Expression -

digit [0-9]+
number digit(.digit)?(E[+-]? digit)?

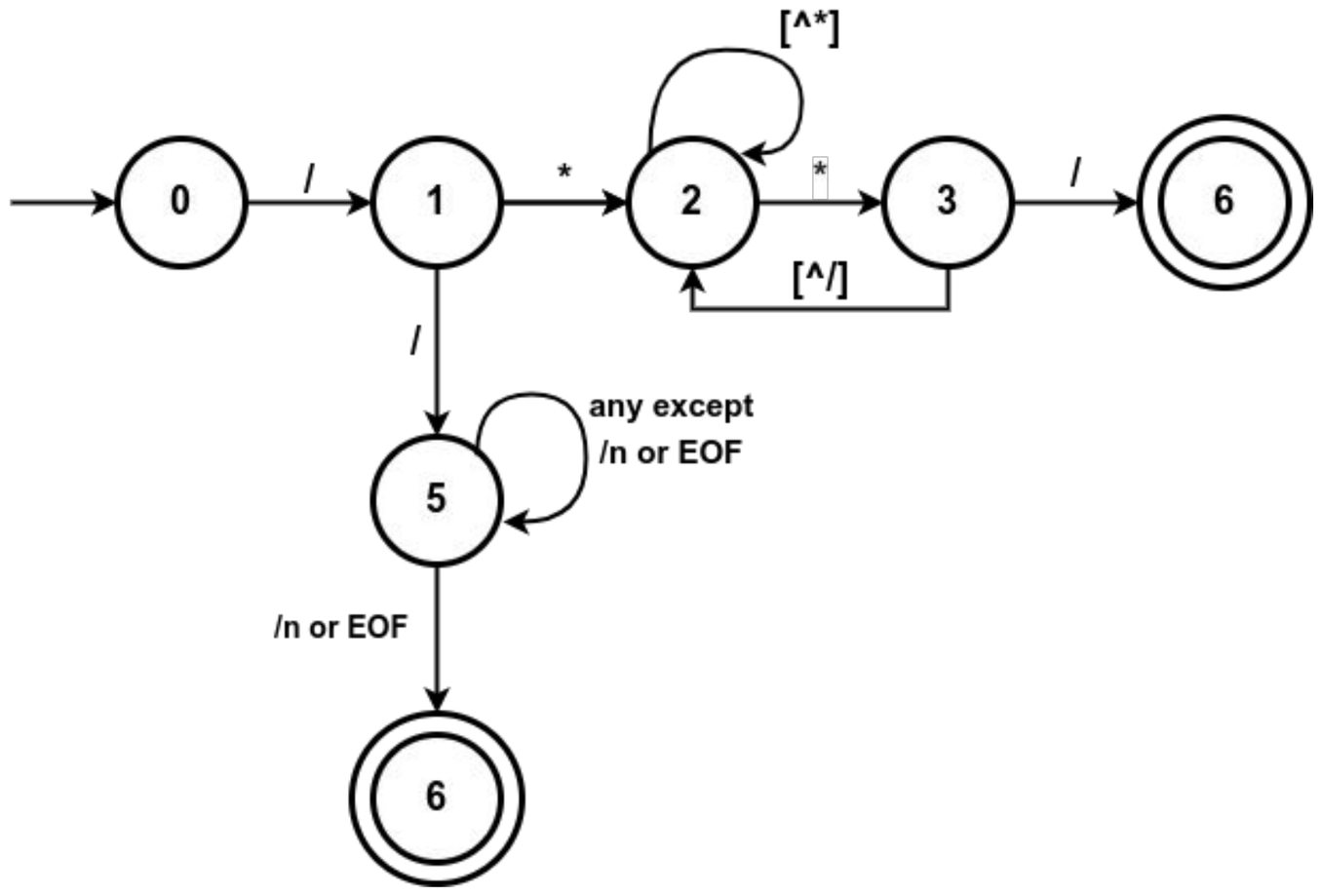


Compiler Design

Transition Diagram for Single and Multi-Line Comments in C

Regular Expression -

single-line comments: \/\/(.*)
multiline comments: \/*(.*\n)*.**\/



Compiler Design

Keywords vs Identifiers - How does the lexer handle keywords?

The transition diagram for identifiers will also recognise Keywords.

There are two ways to handle keywords that look like identifiers -

1. Install the reserved words in the symbol table beforehand.
   a. One field of the symbol table entry can indicate that these are reserved words (e.g. token_name = keyword)
   b. Thus, when an identifier is found
      i. installID() is called to place the identifier in the symbol table.
      ii. Since keywords were preloaded in the symbol table, installID() returns the pointer to the keyword found.
   c. getToken() can be used to return the token_name

Keywords vs Identifiers - How does the lexer handle keywords?

2. Create a separate transition diagram for each keyword.
   a. In this approach, each pattern for the keywords should be defined before the pattern for identifiers.
   b. This enables reserved word tokens to be recognised when the lexeme matches both patterns.

Compiler Design

How does the Lexical Analyzer implement all transition diagrams?

There are three possible approaches -

- **Approach 1** - Arrange transition diagrams for each token to be tried sequentially.
- **Approach 2** - Run various transition diagrams in parallel
- **Approach 3** - Combine all transition diagrams into one.

How does the Lexical Analyzer implement all transition diagrams?

- **Approach 1** - Arrange transition diagrams for each token to be tried sequentially.
  - In this case, each time the function fail() is called, it resets the forward pointer to lexeme_begin, and then tries the next transition diagram.
  - In this method, each keyword can have it's own transition diagram, but it must be defined before the transition diagram for identifiers.

How does the Lexical Analyzer implement all transition diagrams?

- **Approach 2** - Run various transition diagrams in parallel
  - This involves feeding the net input character to all transition diagrams
  - If more than one transition diagram matches the lexeme, the usual strategy is to take the longest prefix.

- **Approach 3** - Combine all transition diagrams into one.
  - The transition diagram reads input until there is no possible next state.
  - The longest lexeme that matches a pattern is taken.
  - This is the preferred approach. However, the problem of combining several transition diagrams is more complex.

Compiler Design

Question 1: Find the number of tokens

```
int a = 20;
```

Tokens		Count
int	keyword	1
a	Identifier	1
=	Operator	1
20	Constant	1
;	Special Symbol	1
Total		5

Question 2: Find the number of tokens

```
int main()
{
    //Compiler Design
    int a, b;
    a = 10;
    b = 20;
    return 0;
}
```

Tokens		Count
int main	Keyword	2
()	Special symbol	2
{	Special symbol	1
//Compiler Design	Comment	0
int	Keyword	1
a	Identifier	1
,	Separator	1
b;	Identifier, symbol	2
a=10;	…	4
b=10;	…	4
return 0;	..	3
}	Special symbol	1
Total		22

COMPILER DESIGN

Question 3: Write the tokens and associated attribute

PES UNIVERSITY ONLINE

E = M * C ** 2

- <id, Pointer to symbol table entry for E >
- <assign_op>
- <id , Pointer to symbol table entry for M>
- <mult-op>
- <id, Pointer to symbol table entry for C>
- <exp-op>
- <number, integer value 2>

COMPILER DESIGN

Question 4: List all the tokens in the following code snippet

```
if(a > b)
{
    a = a + 1;
}
else
{
    b = b + 1;
}
```

Pattern	Lexemes
Keyword	if, else
Identifier	a, b
Relop	>
Assignop	=
Arithop	+
Number	1
Punctuation	( ) { ; } { ; }
Number	0

COMPILER DESIGN

Question 4: List all the tokens in the following code snippet

```
if(a > b)
{
    a = a + 1;
}
else
{
    b = b + 1;
}
```

< keyword, if >
< punctuation, (>
<identifier, a >
< relop, > >
< identifier, b >
< punctuation,) >
< punctuation, { >
<identifier, a>
< Assignop, = >
<identifier, a>
< Arithop, + >
< number, 1 >
< punctuation, ; >
< punctuation, } >

< keyword, else >
< punctuation, { >
<identifier, b>
< Assignop, = >
<identifier, b>
< Arithop, + >
< number, 1 >
< punctuation, ; >
< punctuation, } >

THANK YOU

Preet Kanwal

Department of Computer Science & Engineering

preetkanwal@pes.edu