

Presidency University, Bengaluru

CSE 217 - Compiler Design



Introduction



Instructors

Instructor-in-charge : **Dr. Islabudeen. M**

Instructors : **Mr. Sukruth Gowda M.A,**

Mr. Prasad P.S,

Mr. Rahul Biswas,

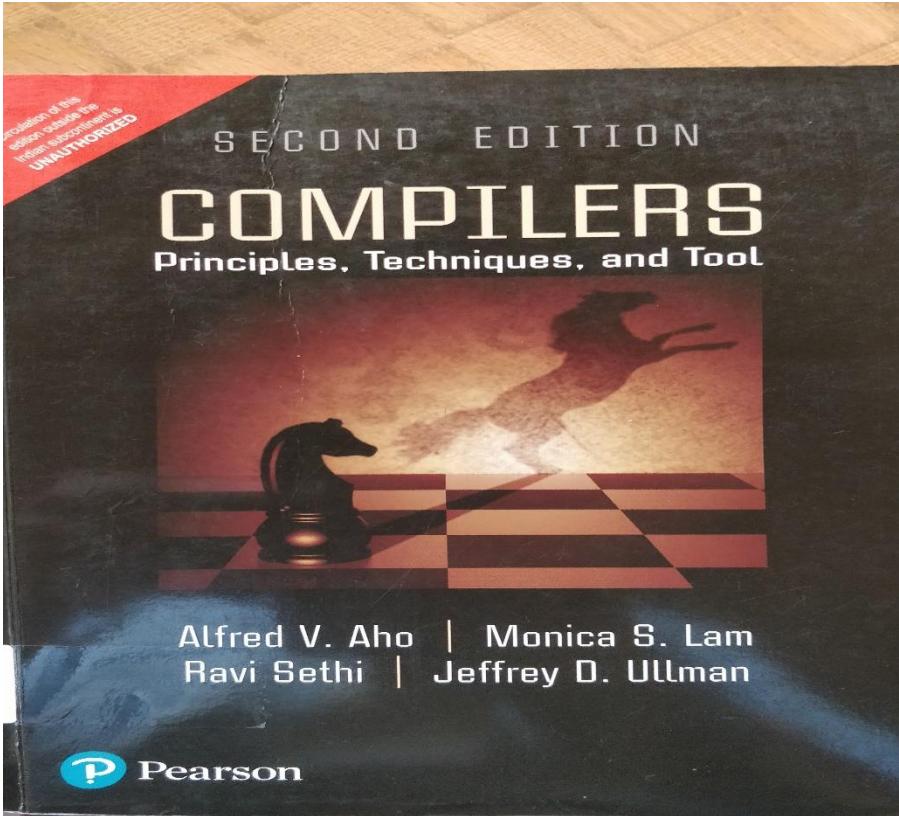
Ms. Shivali Shukya

Mr. Shine V.J



Evaluation Components

	Duration (minutes)	% Weightage	Marks	Date & Time
Test 1	60 min	20	40	To be announced by COE
Test 2	60 min	20	40	To be announced by COE
Quiz / Class Test	60 min	20	40	To be announced by Later
Comprehensive exam	180 min	40	100	To be announced by COE



Text Books:

1. Alfred V. Aho, Jeffrey D Ullman, "Compilers: Principles, Techniques and Tools", Pearson second Edition, 2013.

Reference Books:

1. Jean Paul Tremblay, Paul G Serenson, "The Theory and Practice of Compiler Writing", BS Publications, 2005.
2. C. N. Fischer and R. J. LeBlanc, "Crafting a compiler with C", Benjamin Cummings, 2003.
3. HenkAlblas and Albert Nymeyer, "Practice and Principles of Compiler Building with C", PHI, 2001.
4. Kenneth C. Louden, "Compiler Construction: Principles and Practice", Thompson Learning, 2003.
5. Dhamdhere, D. M., "Compiler Construction Principles and Practice", Macmillan India Ltd, 2008

Course Content (Syllabus):



Module I: INTRODUCTION AND LEXICAL ANALYSIS (13 hours) [COMPREHENSION]

Compilers – Cousins of the Compiler - Phases of a compiler - Analysis of the source program - Grouping of phases – Compiler construction tools – Lexical Analysis – Role of the Lexical Analyzer – Input buffering – Specification of tokens – Recognizer - Introduction to LEX Programming.

Module II: SYNTAX ANALYSIS (15 hours) [APPLICATION]

Role of the parser - Top-down parsing - Recursive decent parser - Predictive parser - Bottom-up parsing – Shift reduce parser - LR parser – SLR parser – Canonical parser – LALR parser - YACC programming.

Module III: SEMANTIC ANALYSIS AND INTERMEDIATE CODE GENERATION (14 hours) [APPLICATION]

Introduction to syntax directed translation - Synthesis and inherited attributes - Type Checking - Type Conversions - Intermediate languages – Three address statements - Declarations – Assignment Statements – Boolean Expressions – Case Statements – Back patching – Looping statements - Procedure calls.

Module IV: CODE OPTIMIZATION AND CODE GENERATION (12 hours) [COMPREHENSION]

Basic Blocks and Flow Graphs – Principal sources of optimization – Peephole optimization - Optimization of basic Blocks - DAG representation of Basic Blocks - Issues in the design of code generator – A simple code generator.



Course Outcomes:

On successful completion of the course the students shall be able to:

CO1: Explain the various phases of compiler (COMPREHENSION)

CO2: Apply parsing techniques to check the syntax of given statement. (APPLICATION)

CO3: Produce intermediate code for the given statement. (APPLICATION)

CO4: Discuss how to optimize the given problem for back end of the compiler (COMPREHENSION)

Program Outcomes



PO1 Engineering Knowledge:

Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems. **[High]**

PO2 Problem Analysis:

Identify, formulate, research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences. **[High]**

PO3 Design/development of Solutions:

Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations. **(High)**

PO4 Conduct Investigations of Complex Problems:

Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions. **[High]**

PO5 Modern Tool usage:

Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modelling to complex engineering activities with an understanding of the limitations. **[Moderate]**

PO10 Communication:

Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions. **[Low]**



Mapping of CO with PO

CO / PO N0.	PO1	PO2	PO3	PO4	PO5	PO10
CO1	H	M	M	M	M	L
CO2	H	H	H	H	M	L
CO3	H	H	H	H	-	L
CO4	H	H	M	M	-	L

Course Prerequisites



- Finite Automata, Context Free Grammar
(CSE208 - Theory of Computation – CSE 5th Semester Course)
- System software



```
#include<stdio
.h>
#define N 10
main()
{
    int a;
    a = N + 20;

    printf("%d",a)
;
}
```

Monitor

30

```
Editor
.h>
#define N 10
main()
{
    int a;
    a = N + 20;

printf("%d", a)
;
}
```

Operating System

Pre-processor

Compiler

Assembler

Interpreter

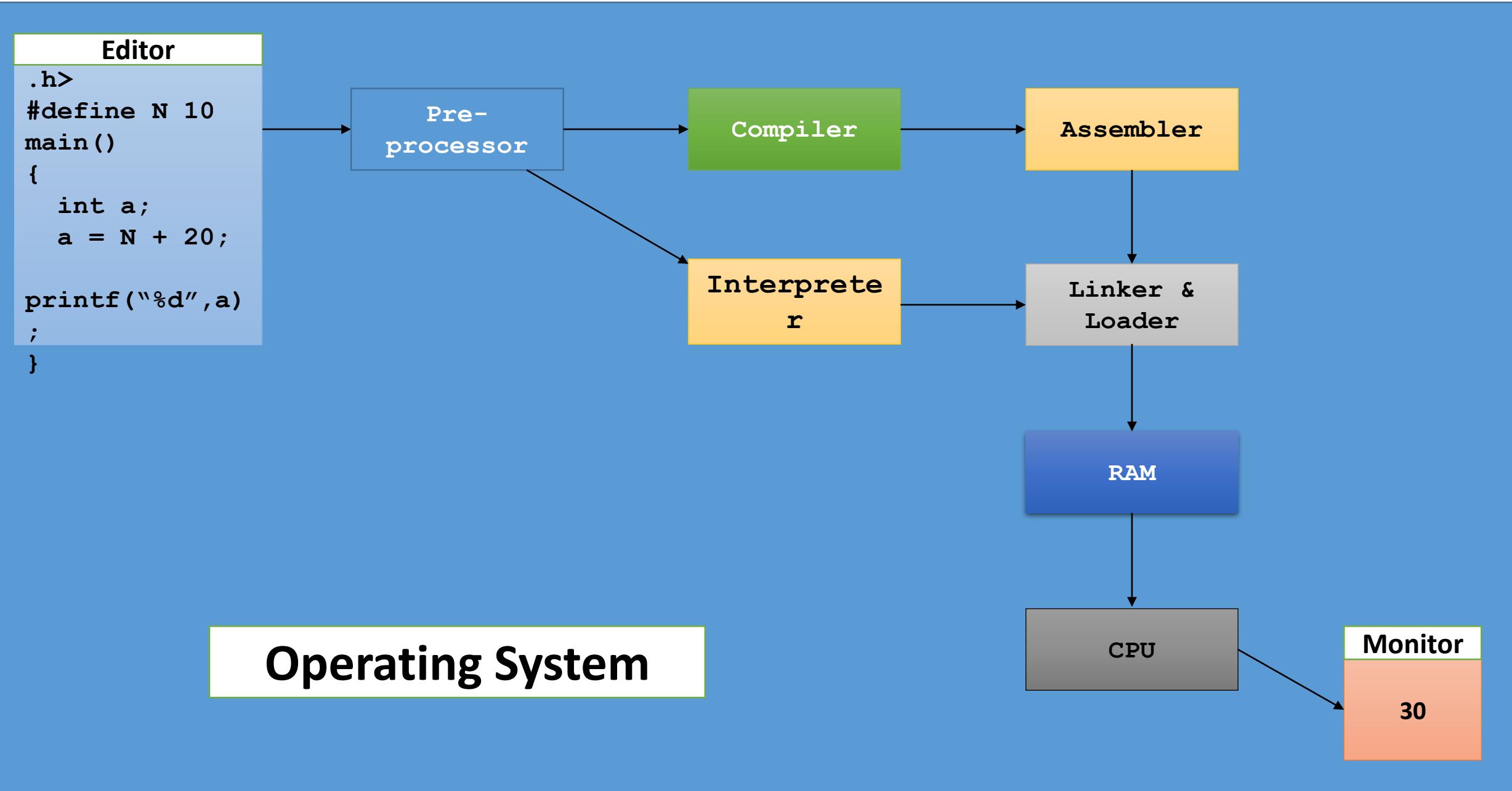
Linker & Loader

RAM

CPU

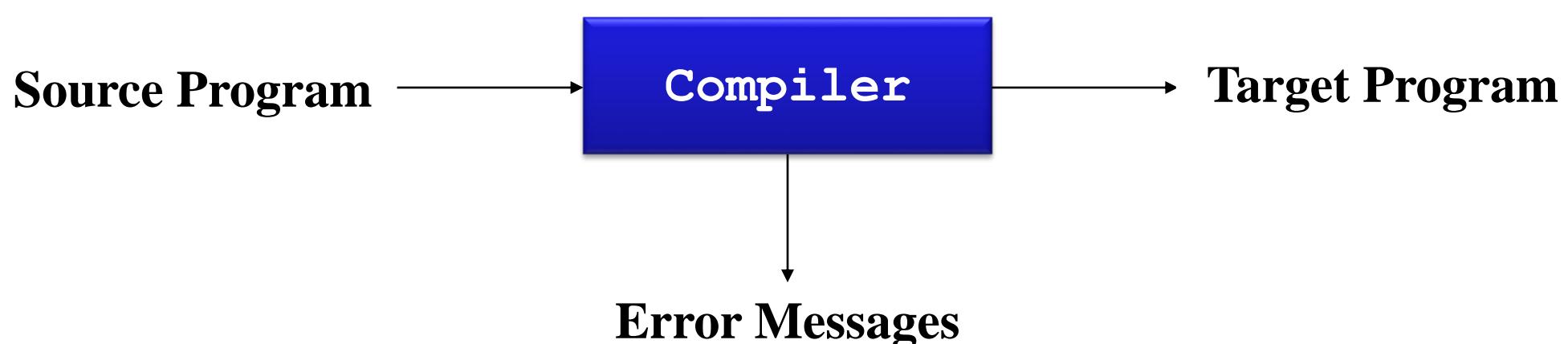
Monitor

30





- Compiler is a program that reads a program written in one language (source language) and translates it into an equivalent program in another language (the target language).
- As an important part of this translation process, the compiler reports to its user the presence of errors in the source program.





- Source code
 - written in a high level programming language
- Target code
 - Assembly language which in turn is translated to machine code

```
//simple example
while (sum < total)
{
    sum = sum + x*10;
}
```

L1: MOV total, R0
CMP sum, R0
CJ< L2
GOTO L3

L2: MOV #10, R0
MUL x, R0
ADD sum, R0
MOV R0, sum
GOTO L1

L3: First Instruction following the while statement

Why build compilers?



- Programming languages are notations for describing computations to people and to machines. The world as we know it depends on programming languages, because all the software running on all the computers was written in some programming language. But, before a program can be run, it first must be translated into a form in which it can be executed by a computer.
- Compilers provide an essential interface between applications and architectures
- Compilers efficiently bridge the gap and shield the application developers from low level machine details

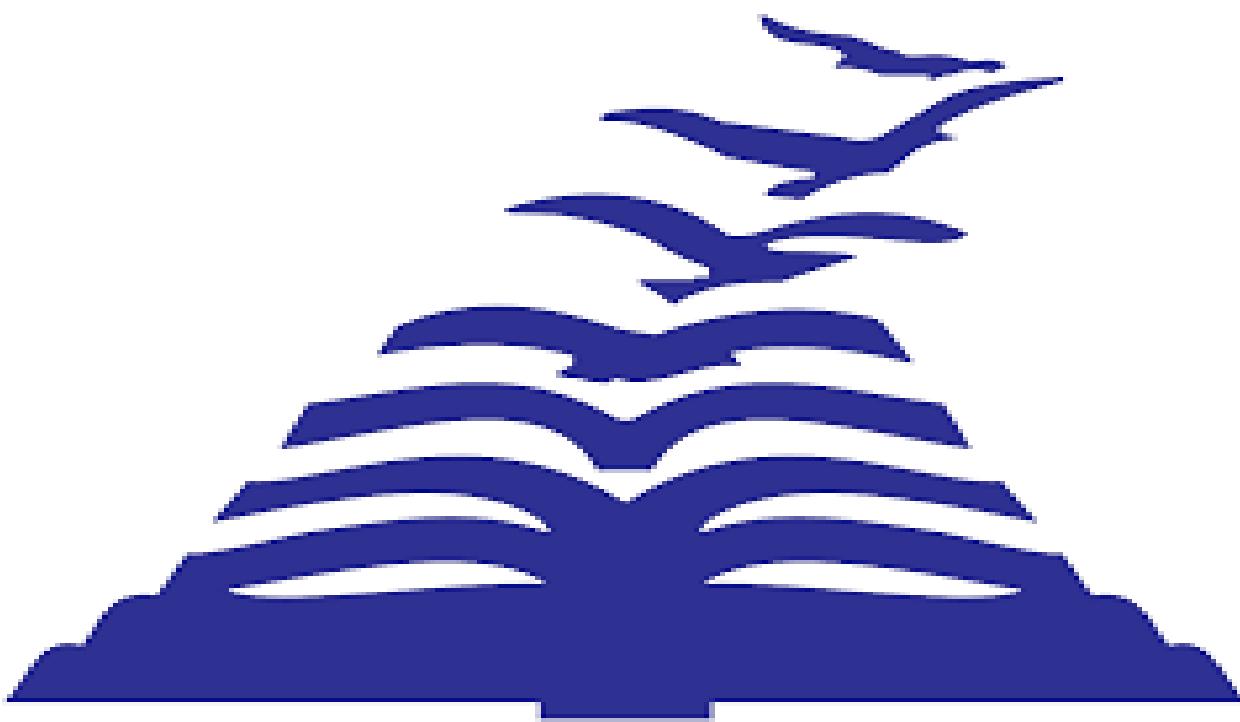
Why study compilers?



- Compilers embody a wide range of theoretical techniques and their application to practice
 - DFAs, PDAs, formal languages, formal grammars, fix points algorithms, lattice theory, etc...
- Compiler construction teaches programming and software engineering skills
- Compiler construction involves a variety of areas
 - theory, algorithms, systems, architecture
- **Is compiler construction a solved problem?**
 - No! New developments in programming languages and machine architectures (processors) present new challenges



- **Compiler must generate a correct executable**
 - The input program and the output program must be equivalent, the compiler should preserve the meaning of the input program
- **Output program should run fast**
 - For optimizing compilers we expect the output program to be more efficient than the input program
- **Compiler should provide good diagnostics for programming errors**
- **Compiler should work well with debuggers**
- **Compile time should be proportional to code size**



Presidency University, Bengaluru

COUSINS OF COMPILER

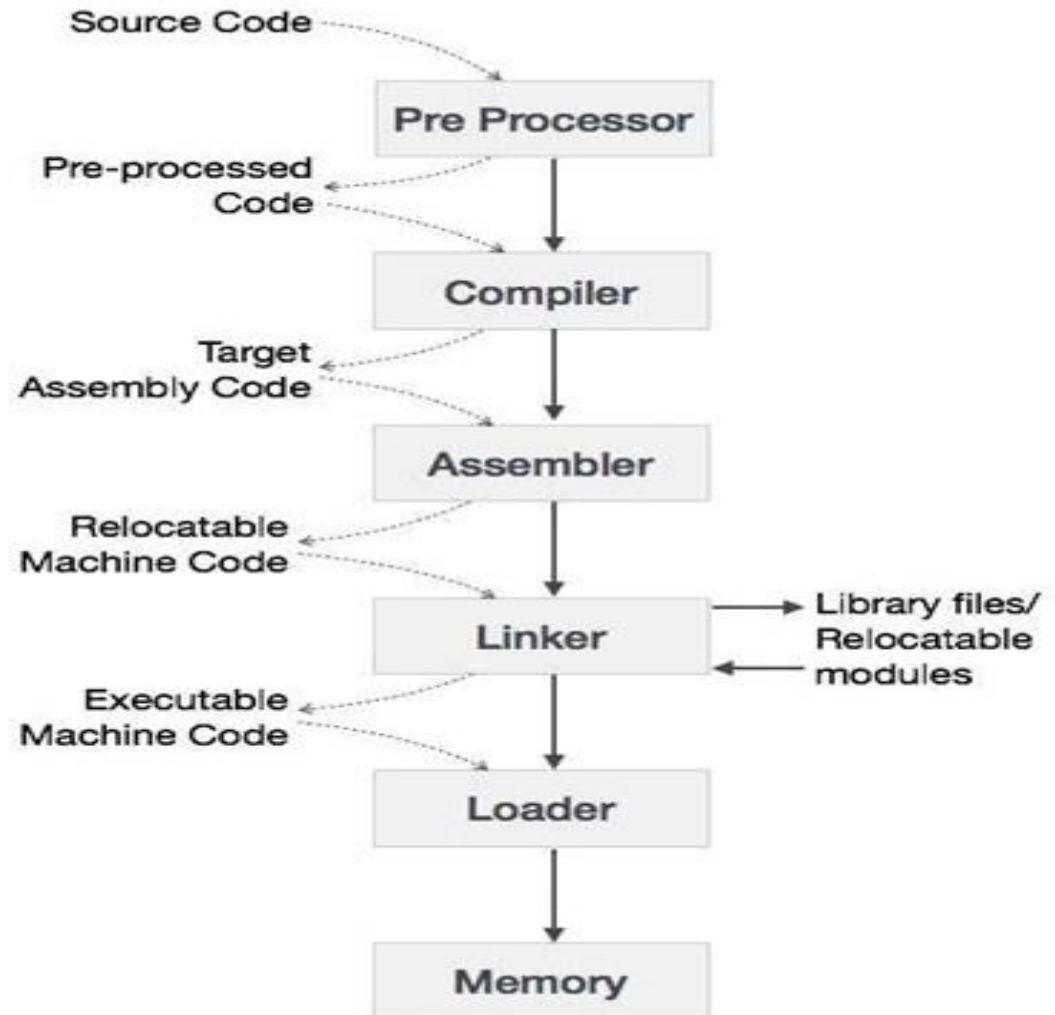
COUSINS OF COMPILER



The input to a compiler may be processed by one or more preprocessors, and further processing of the compiler's output may be needed before running machine code is obtained.

Cousins of Compiler are

1. Preprocessor
2. Assembler
3. Loader and Link-editor





Preprocessor

- A preprocessor is a program that processes its input data (**Source Program with Preprocessing Statements**) to produce output (**Source Program without Preprocessing Statements**) that is used as input to another program (**Compilers**).

They may perform the following functions :

1. **Macro processing**
2. **File Inclusion**
3. **Rational Preprocessors**
4. **Language extension**



1. Macro processing:

A Preprocessor may allow a user to define macros are shorthands for longer constructs.

2. File Inclusion:

Preprocessor includes header files into the program text. When the preprocessor finds an `#include` directive it replaces it by the entire content of the specified file.

3. Rational Preprocessors:

These processors change older languages with more modern flow-of-control and data- structuring facilities.

4. Language extension :

These processors attempt to add capabilities to the language by what amounts to built-in macros. For example, the language Equel is a database query language embedded in C.



Assembler creates object code by translating assembly instruction mnemonics into machine code.

There are two types of assemblers:

- One-pass assemblers go through the source code once and assume that all symbols will be defined before any instruction that references them.
- Two-pass assemblers create a table with all symbols and their values in the first pass, and then use the table in a second pass to generate code



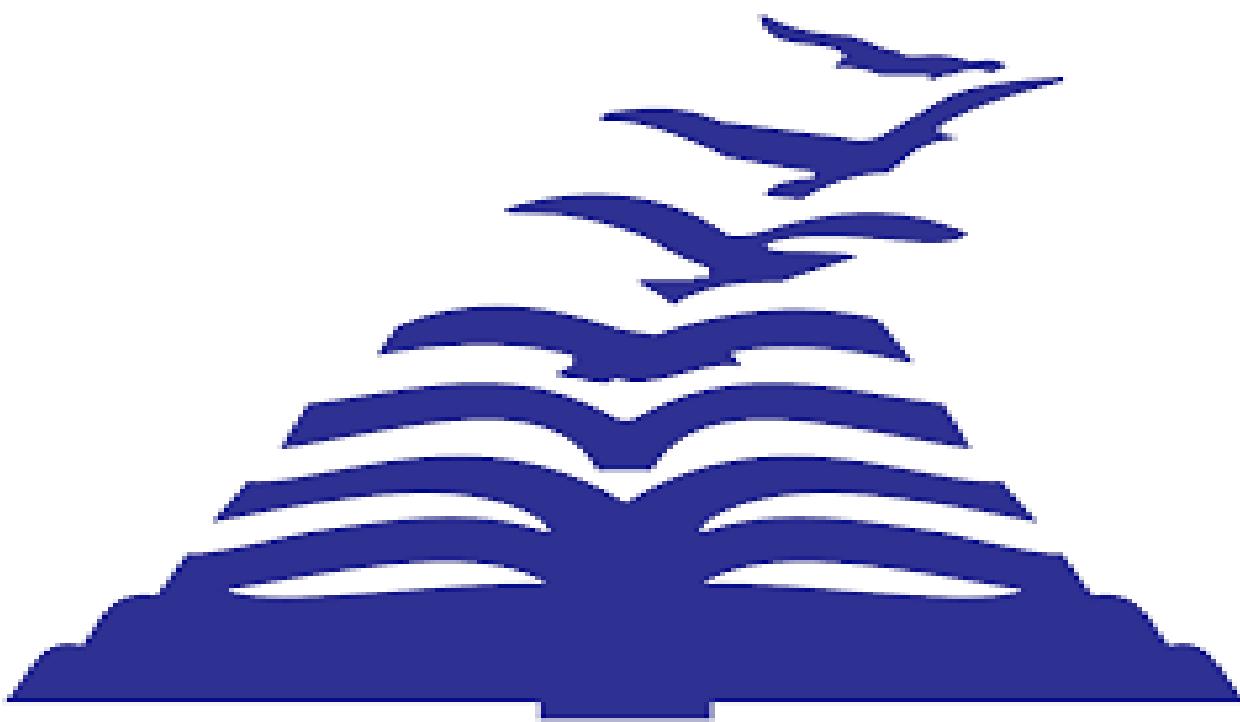
Linker and Loader

A **linker** or **link editor** is a program that takes one or more objects generated by a compiler and combines them into a single executable program.

Three tasks of the linker are

1. Searches the program to find library routines used by program, e.g. printf(), math routines.
2. Determines the memory locations that code from each module will occupy and relocates its instructions by adjusting absolute references
3. Resolves references among files.

A **loader** is the part of an operating system that is responsible for loading executable programs into permanent memory for execution.

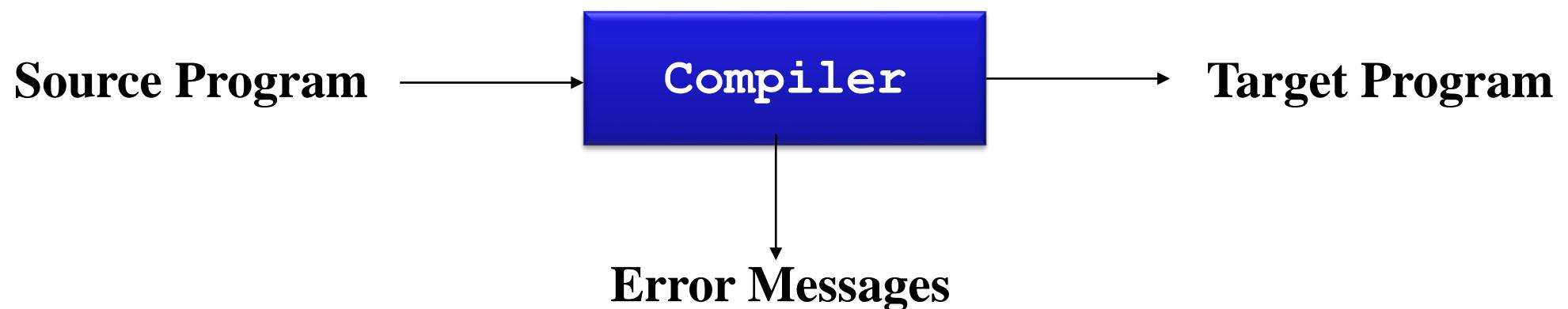


Presidency University, Bengaluru

PHASES OF COMPILER



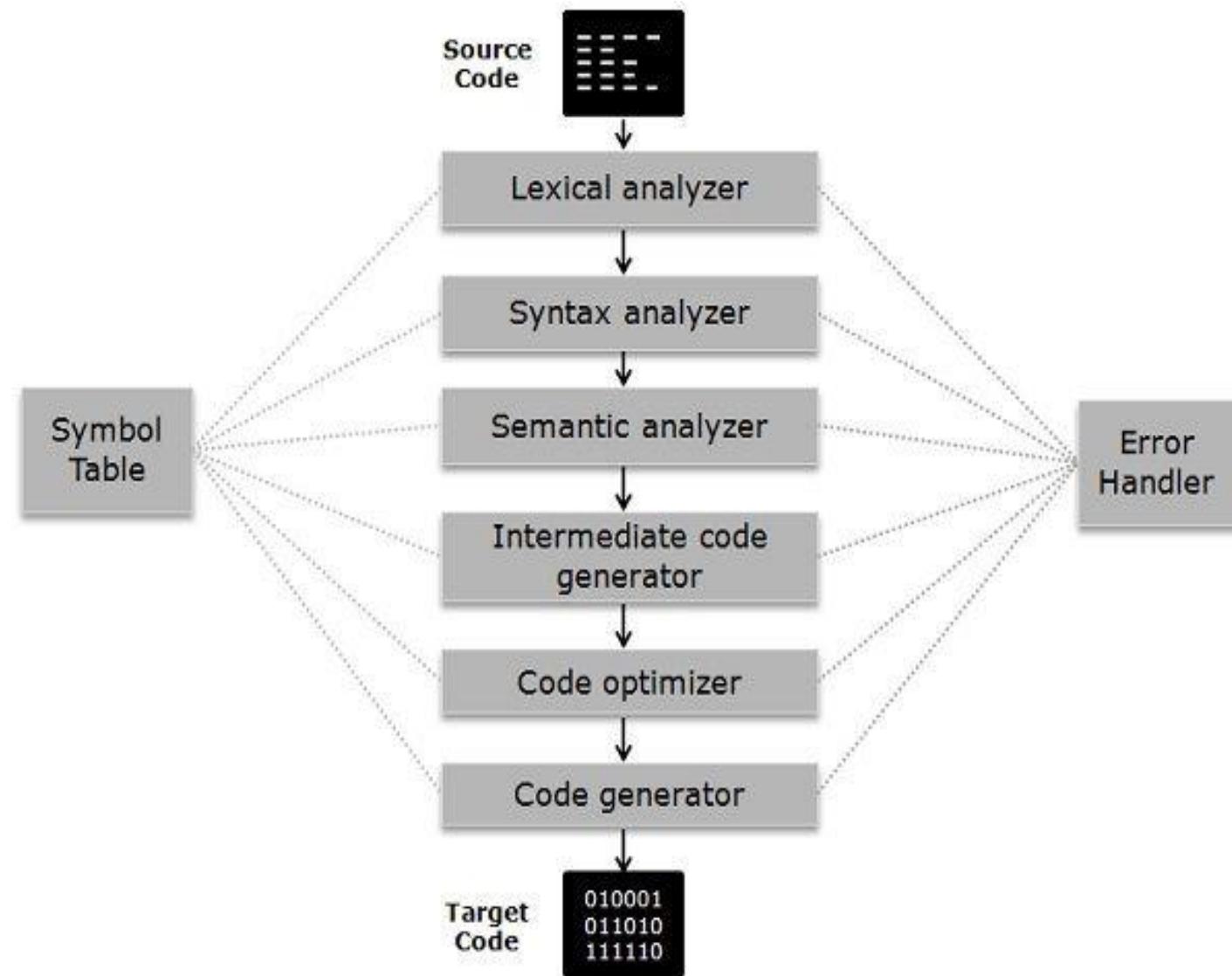
- Compiler is a program that reads a program written in one language (source language) and translates it into an equivalent program in another language (the target language).
- As an important part of this translation process, the compiler reports to its user the presence of errors in the source program.





Phases of a Compiler

Conceptually, a compiler operates in **phases**, each of which transforms the source program from one representation to another.



Lexical Analysis



- The first phase of a compiler is called lexical analysis or **scanning**. It is also called as **linear analysis**.
- The lexical analyzer reads the stream of characters making up the source program and groups the characters into meaningful sequences called lexemes.
- For each lexeme, the lexical analyzer produces as output a token of the form **(token-name, attribute-value)** that it passes on to the subsequent phase, syntax analysis.
- In the token, the first component token-name is an abstract symbol that is used during syntax analysis, and the second component attribute-value points to an entry in the symbol table for this token.
- Information from the symbol-table entry is needed for semantic analysis and code generation.

Lexical Analysis



For example, suppose a source program contains the assignment statement

position = initial + rate * 60

1. **position** is a lexeme that would be mapped into a **token (id, 1)**, where id is an abstract symbol standing for identifier and 1 points to the symbol table entry for position. The symbol-table entry for an identifier holds information about the identifier, such as its name and type.
2. The **assignment symbol =** is a lexeme that is mapped into the token (=). Since this token needs **no attribute-value**.
3. **initial** is a lexeme that is mapped into the token **(id, 2)**, where 2 points to the symbol-table entry for initial .
4. **+** is a lexeme that is mapped into the **token (+)**.
5. **rate** is a lexeme that is mapped into the **token (id, 3)**, where 3 points to the symbol-table entry for rate .
6. ***** is a lexeme that is mapped into the **token (*)**.
7. **60** is a lexeme that is mapped into the token (number type,value). ➔ (Integer, 60)

Lexical Analysis



position = initial + rate * 60



(id,1) = (id,2) + (id,3) * (number,60)

For understanding easier and for our convenience, We can write this as

id1 = id2 + id3 * 60

Syntax Analysis

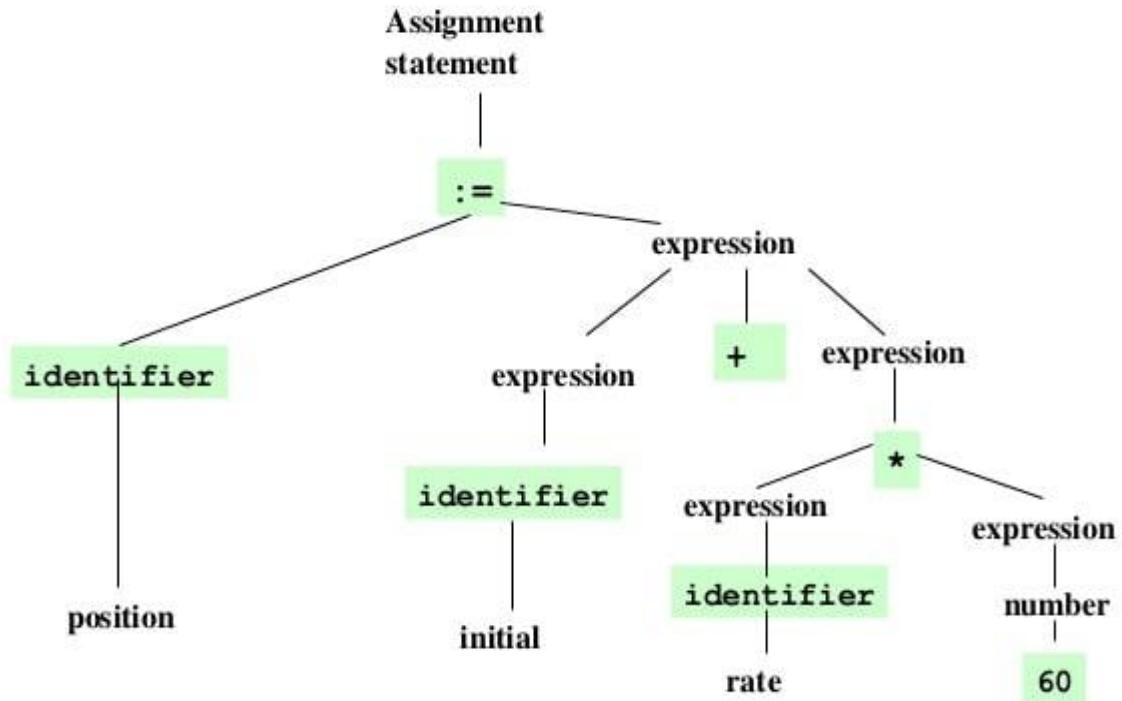


- The second phase of the compiler is **syntax analysis or parsing**.
- The parser uses the first components of the tokens produced by the lexical analyzer to create a **tree-like intermediate representation that depicts the grammatical structure of the token stream**.
- A typical representation is a **syntax tree** in which each interior node represents an operation and the children of the node represent the arguments of the operation.

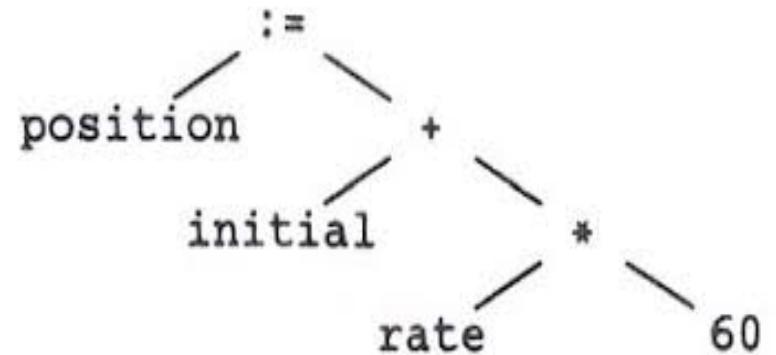


Parse tree

position := initial + rate * 60



Syntax Tree



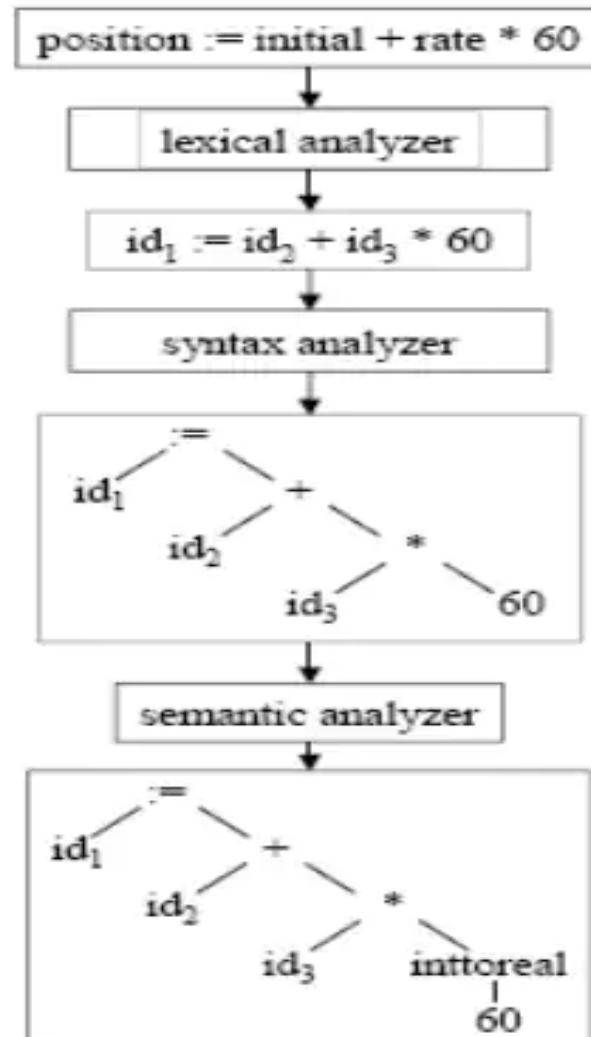
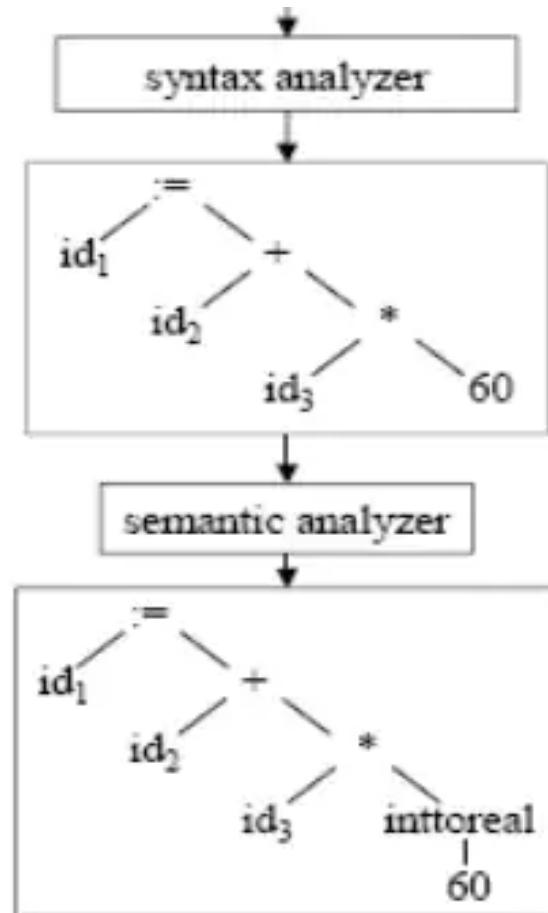
Semantic Analysis



- The semantic analyzer uses the syntax tree and the information in the symbol table to check the source program for semantic consistency with the language definition.
- It also gathers type information and saves it in either the syntax tree or the symbol table, for subsequent use during intermediate-code generation.
- An **important part of semantic analysis is type checking**, where the compiler checks that each operator has matching operands.
- For example, many programming language definitions require an array index to be an integer; the compiler must report an error if a floating-point number is used to index an array.



Semantic Analysis



Semantic Analysis



- The language specification may permit some type conversions called coercions. (Conversion from one type to another is said to be implicit if it is to be done automatically by the compiler. Implicit type conversions, also called Coercions (no information is lost in principle).

```
int b;  
float a;  
a = b;
```

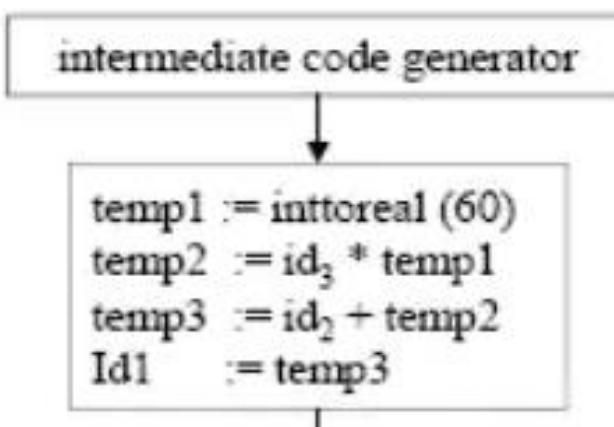
- Conversion is said to be explicit if the programmer must write something to cause the conversion.

```
int b;  
float a;  
a = (int) b;
```

Intermediate Code Generation



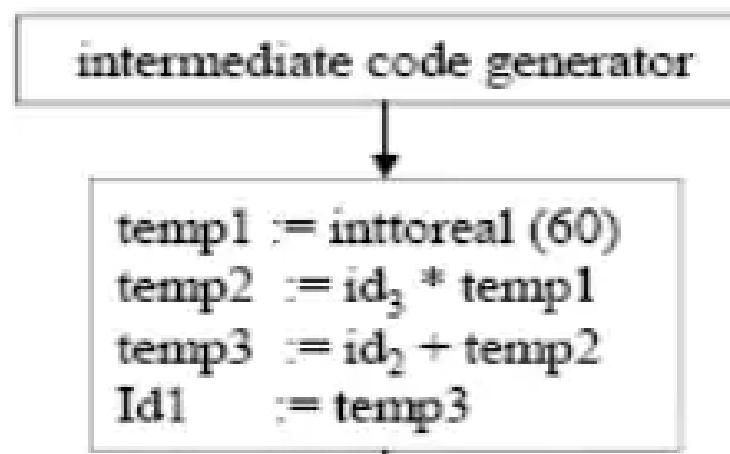
- After syntax and semantic analysis of the source program, many compilers generate an explicit low-level or machine-like intermediate representation.
- In the process of translating a source program into target code, a compiler may construct one or more intermediate representations, which can have a variety of forms.
- This intermediate representation should have two important properties: it should be easy to produce and it should be easy to translate into the target machine



Intermediate Code Generation



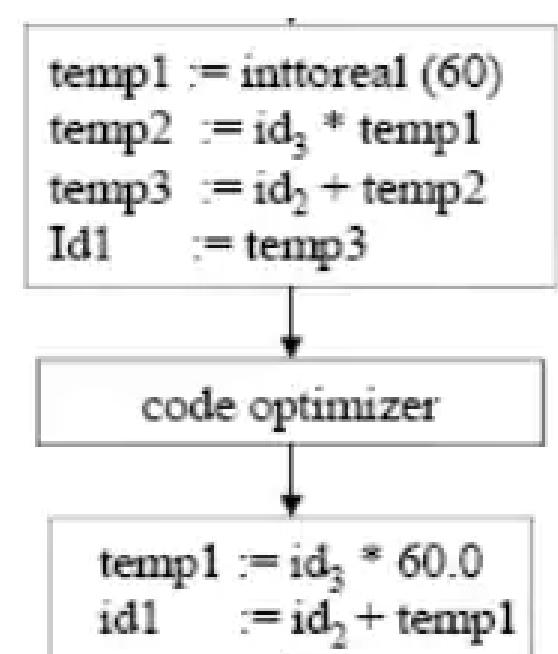
- One of the mostly used intermediate form called **three-address code**, which consists of a sequence of assembly-like instructions with three operands per instruction.
- Three-address instruction has several important properties:
 - First, each three-address assignment instruction has at most one operator on the right side.
 - Second, the compiler must generate a temporary name to hold the value computed by a three-address instruction.
 - Third, some three-address instructions have fewer than three operands.





Code Optimization

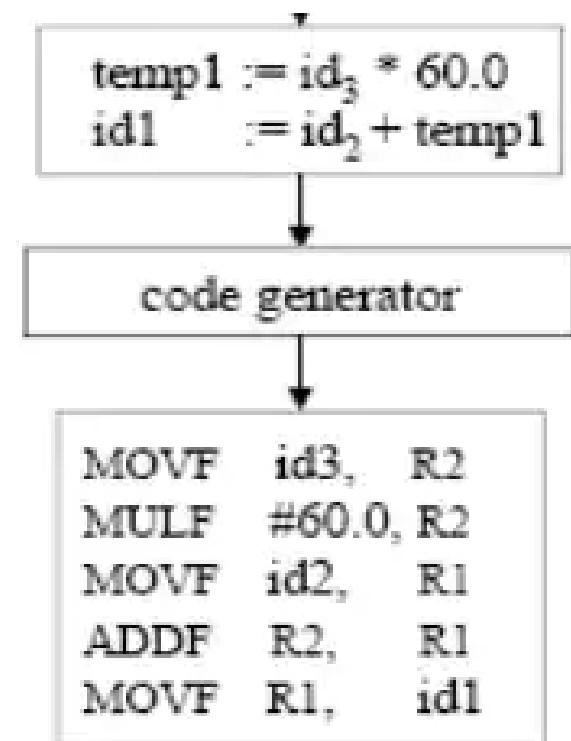
- The machine-independent code-optimization phase **attempts to improve the intermediate code so that better target code will result**. Usually better means **faster**, but other objectives may be desired, such as shorter code, or target code that **consumes less power**.
- There is a great variation in the amount of code optimization different compilers perform.
- Optimizing compilers** spent a significant amount of time is spent on this phase. There are simple optimizations that significantly improve the running time of the target program **without slowing down compilation too much**.





Code Generation

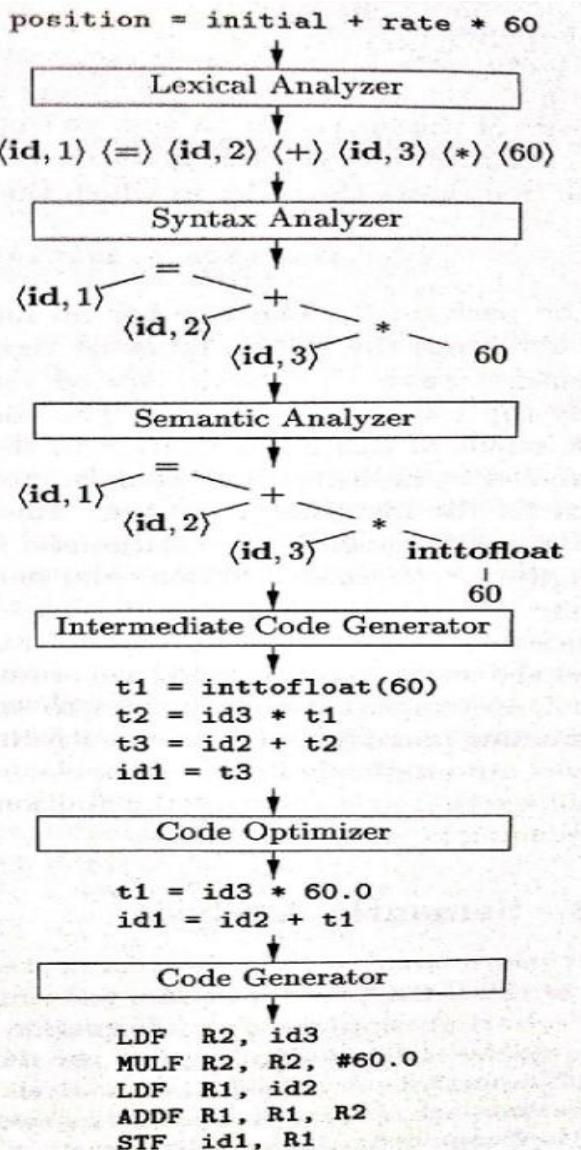
- Final phase of the compiler is generation of target code, consisting normally of **relocatable machine code or assembly code**.
- Memory locations are selected for each of the **variables** used by the program.
- Then, the **intermediate instructions** are translated into sequences of machine instructions that perform the same task.
- A **crucial aspect** of code generation is the judicious assignment of registers to hold variables.





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

SYMBOL TABLE





Symbol Table Management

- An essential function of a compiler is to record the variable names used in the source program and collect information about various attributes of each name. These attributes may provide information about the storage allocated for a **name, its type, its scope** (where in the program its value may be used), and in the case of **procedure names**, such things as the **number and types of its arguments**, the **method of passing each argument** (for example, by value or by reference), and the **type returned**.
- The **symbol table is a data structure** containing a record for each variable name, with fields for the attributes of the name.
- The data structure should be designed to allow the compiler to **find the record for each name quickly and to store or retrieve data from that record quickly**.

Symbol Table Management



Symbol Table for Identifiers (Variables)

Name of the Symbol	Location	Type	Scope	Value	Size	...

Symbol Table for Identifiers (Functions)

Name of the Symbol	No. of Arguments	Type of Arguments	Scope	Return type	...

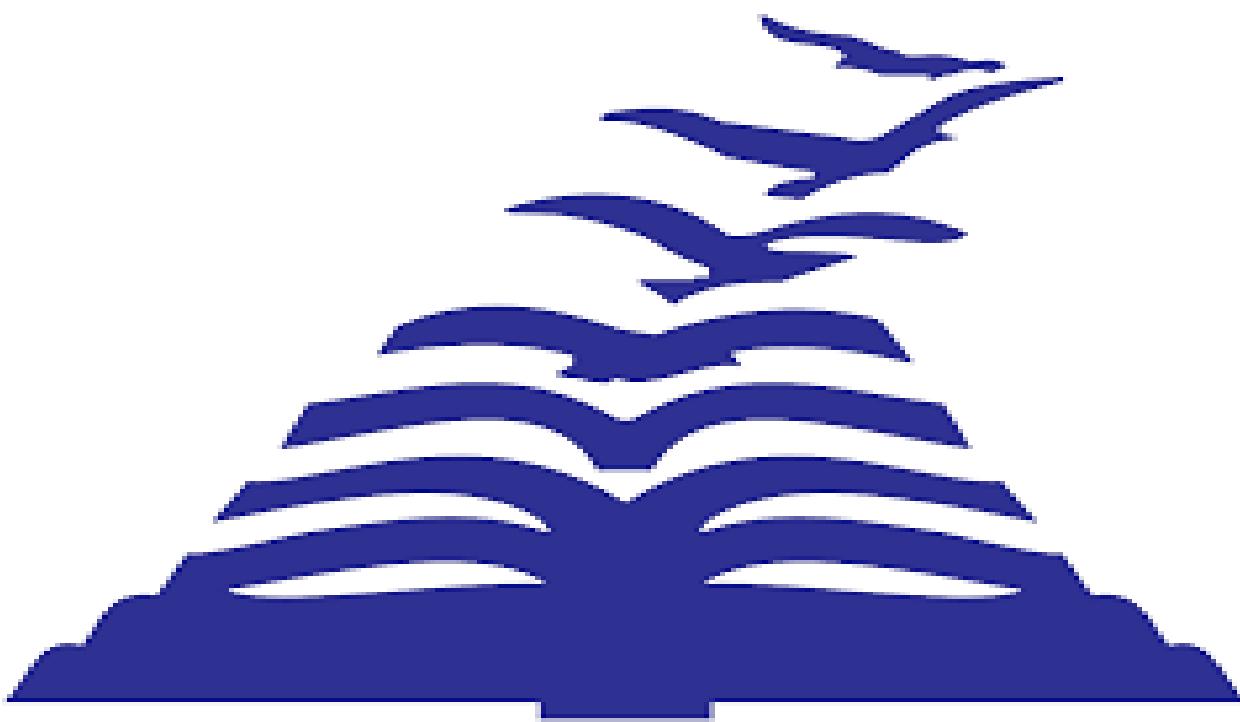


Error Detection and Reporting

- Each phase can encounter Errors. However, after detecting an error, a phase must somehow deal with that error, so that compilation can proceed, allowing further errors in the source program to be detected.
- The syntax and semantic analysis phases usually handle a large fraction of the errors detectable by the compiler.

Some Errors in Phases of Compiler

- Lexical analyzer: Wrongly spelled tokens
- Syntax analyzer: Missing parenthesis
- Semantic analyzer: Mismatched operands for an operator
- Intermediate code generator: When the memory is full when generating temporary variables
- Code Optimizer: When the statement is not reachable
- Code Generator: When the memory is full or proper registers are not allocated



Presidency University, Bengaluru

Analysis of the Phases and Grouping of Phases

Analysis of the Phases



Compilation Process is divided into two parts:

- **Analysis and**
- **Synthesis**

- The analysis part breaks up the source program into constituent pieces and imposes a grammatical structure on them.
- It then uses this structure to create an intermediate representation of the source program.
- If the analysis part detects that the source program is either syntactically ill formed or semantically unsound, then it must provide informative messages, so the user can take corrective action.
- The analysis part also collects information about the source program and stores it in a data structure called a symbol table, which is passed along with the intermediate representation to the synthesis part.



- The synthesis part constructs the desired target program from the intermediate representation and the information in the symbol table.
- The **analysis part** is often called the **front end of the compiler**; the **synthesis part** is the **back end**.



Grouping of Phases into Passes



- The **analysis part** is often called the **front end of the compiler**; the **synthesis part** is the **back end**.
- The discussion of **phases** deals with the **logical organization** of a compiler.
- In an implementation, activities from **several phases may be grouped together into a pass** that reads an input file and writes an output file.

Front End

- For example, the **front-end phases of lexical analysis, syntax analysis, semantic analysis, and intermediate code generation** might be grouped together into one pass.
- **Front ends** are **depends primarily** on **source language** and are largely independent of the target machine.



Back End

- The **back end** includes those portions of the compiler that **depend on the target machine**, and generally, these portions do **not depend on the source language, just the intermediate language**.
- **Code optimization** might be an **optional pass**. Then there could be a back-end pass consisting of **code generation** for a particular target machine.



Some compiler collections have been created around carefully designed **intermediate representations** that allow the front end for a particular language to interface with the back end for a certain target machine.

With these collections, **we can produce compilers for different source languages for one target machine** by combining different front ends with the back end for that target machine.

Similarly, **we can produce compilers for different target machines**, by combining a front end with back ends for different target machines.



Several phases of compilation are implemented in a single pass consisting of reading an input file and writing an output file.

Reducing the number of passes

- Takes time to read and write intermediate files.
- Grouping of several phases into one pass, may force the entire program in memory, because one phase may need information in a different order than previous phase produces it.
- Intermediate code and code generation are often merged into one pass using a technique called backpatching.



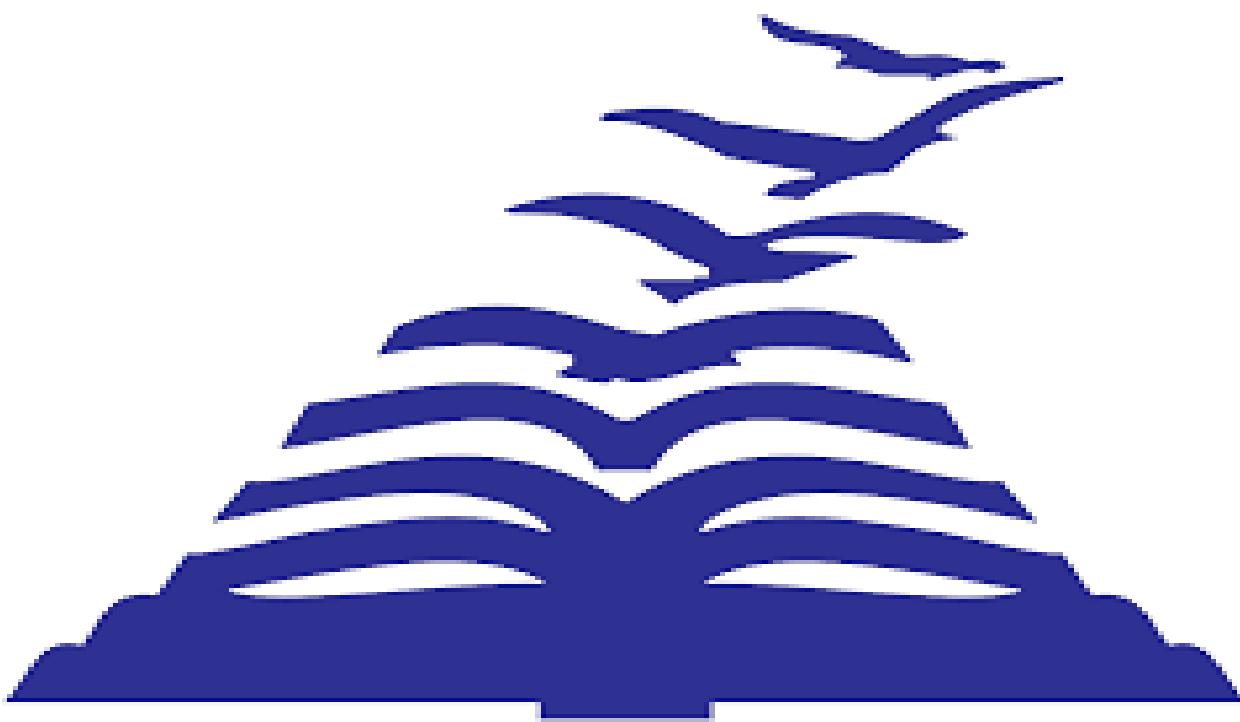
For some phases, grouping into single pass presents few problems.

For example, the interface between lexical and syntax analyzers can often limited into **single token**.

On the other hand, it is often **very hard** to perform **code generation** until the **intermediate representation** has been **completely generation**.

To avoid these issues, **Backpatching** is used.

In some cases, it is possible to leave a blank slot for missing information, and fill in the slot when the information become available. This is called as “**Backpatching**”.



Presidency University, Bengaluru

Compiler Construction Tools

Compiler Construction Tools



The compiler writer, like any **software developer**, can profitably use modern software development environments containing tools such as **language editors, debuggers, version managers, profilers, test harnesses**, and so on.

Shortly after the **first compilers were written**, systems to help with the compiler-writing process appeared. These systems have often been referred to as **Compiler-Compilers, Compiler-Generators** or **Translator-Writing Systems**.

Some general tools have been created for the **automatic design of specific compiler** components. These tools use specialized languages for specifying and implementing the component, and also many use sophisticated algorithms.

The most successful tools produce components that can be easily integrated into the remainder of a compiler.

Compiler Construction Tools



The following is a list of some useful compiler-construction tools:

- **Parser Generators**
- **Scanner Generators**
- **Syntax-directed Translation Engines**
- **Automatic Code Generators**
- **Data-flow Analysis Engines**



- **Scanner Generators**

This tool **automatically generate lexical analyzers**, normally from a specification based on **regular expressions**. The basic organization of the resulting lexical analyzers is in effect a **finite automaton**.

- **Parser Generators**

These tools produce **syntax analyzers**, normally from input that is based on a **Context-Free Grammar (CFG)**. In early compilers, syntax analysis consumed not only a large fraction of the compilation time, but a large fraction of the intellectual effort of writing a compiler. **Many parser generator utilize powerful parsing algorithm that are too complex.**



- **Syntax-directed Translation Engines**

These tools produce **collections of routines** for walking a parse tree and **generating intermediate code**. The basic idea is that one or more “translation” are associated with each node of the parse tree, and each translation is defined in terms of translations at its neighbor nodes in the tree.

- **Automatic Code Generators**

These tools **produce a code generator** from a **collection of rules** for translating each operation of the **intermediate language into the machine language for a target machine**. The rules must include sufficient details that we can handle the different possible access methods for data. For example **variable may be in register or fixed (static) location or may be in stack**.

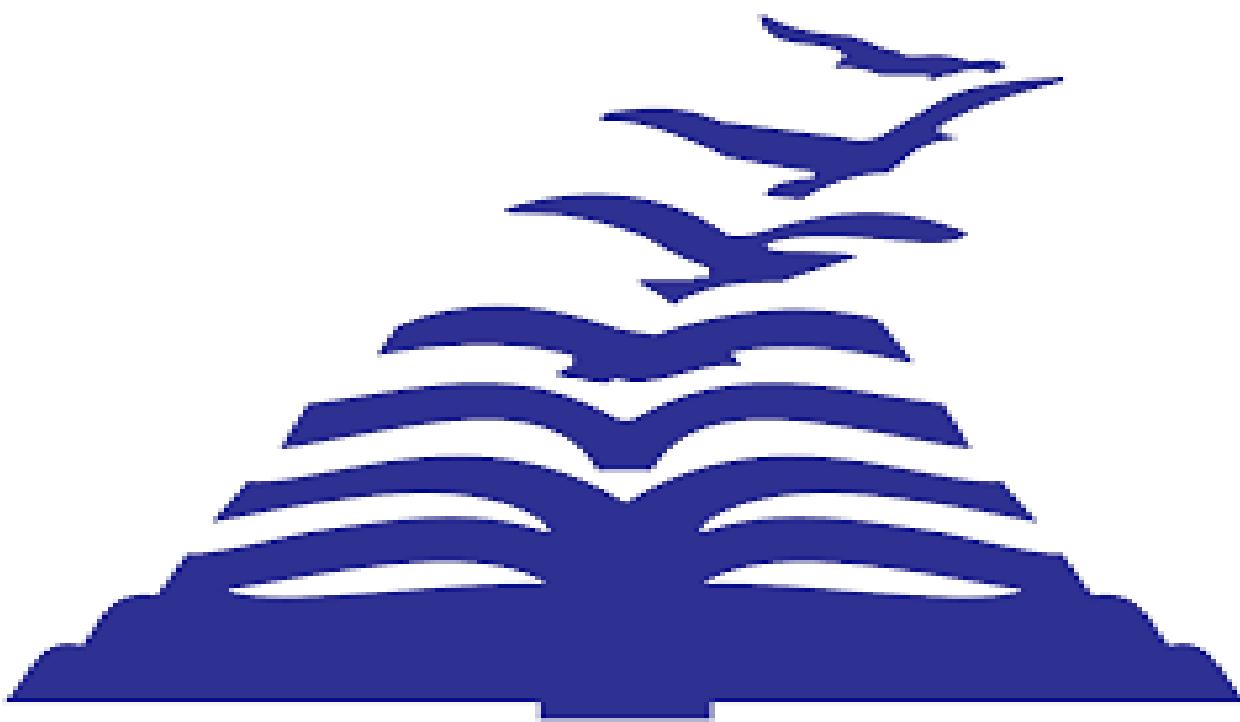
The basic technique is “Template Matching” – The intermediate code statements are replaced by “templates” that represent sequences of machine instruction.



- **Data-flow Analysis Engines**

Much of the information needed to perform **good code optimization** involves “**Data-Flow Analysis**”, the gathering of information about **how values are transmitted from one part of a program to each other part**.

Data-flow analysis is a key part of code optimization.



Presidency University, Bengaluru

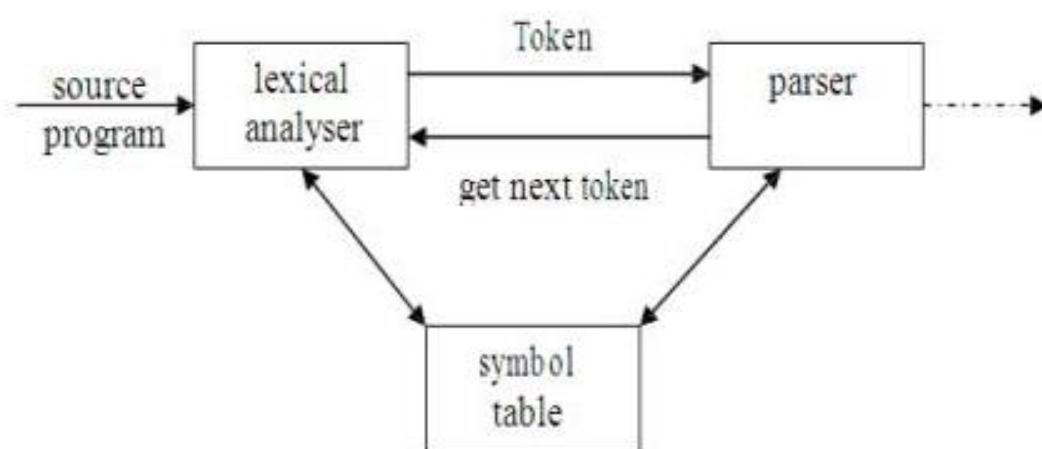
Lexical Analysis

-
**Role and
Need of Lexical
Analysis**

The Role of the Lexical Analyzer



- As the **first phase** of a compiler, the main task of the lexical analyzer is to **read the input characters** of the source program, **group them into lexemes**, and produce as output a sequence of **tokens** for each lexeme in the source program.
- The **stream of tokens** is sent to the parser for syntax analysis.
- It is common for the **lexical analyzer** to interact with the symbol table as well.
- When the lexical analyzer discovers a lexeme constituting an **identifier**, it needs to enter that lexeme into the symbol table.



The Role of the Lexical Analyzer



Issues in Lexical Analysis

There are a number of reasons why the analysis portion of a compiler is normally separated into lexical analysis and parsing (syntax analysis) phases.

1. Simplicity of design is the most important consideration.

The separation of lexical from syntax analysis often allows us to simplify at least one or the other of these phases. (For example, Stripping out comments and white spaces is somewhat easy job by lexical analyzer than syntax analyzer).

2. Compiler efficiency is improved.

A separate lexical analyzer allows us to apply specialized techniques that serve only the lexical task, not the job of parsing. In addition, specialized buffering techniques for reading input characters can speed up the compiler significantly.

3. Compiler portability is enhanced.

Input-device-specific peculiarities can be restricted to the lexical analyzer. (For example, ↑ in pascal)

The Role of the Lexical Analyzer



Lexical Analyzer also performs certain secondary task

- Stripping out comments and whitespace (blank, newline, tab) in the source program.
- Correlating error messages generated by the compiler with the source program.

Sometimes, lexical analyzers are divided into a cascade of two processes:

- **Scanning** consists of the simple processes that do not require tokenization of the input, such as deletion of comments and compaction of consecutive whitespace characters into one.
- **Lexical analysis** proper is the more complex portion, where the scanner produces the sequence of tokens as output.

The Role of the Lexical Analyzer



Tokens, Patterns, and Lexemes

When discussing lexical analysis, we use three related but distinct terms:

Token

- A token is a pair consisting of a token name and an optional attribute value.
- The token name is an abstract symbol representing a kind of lexical unit (Keyword, Identifier, Operator...)

Pattern

- The set strings is described by a rule called a pattern associated with the token.
Or
- A pattern is a rule describing the set of lexemes that can represent a particular token in source programs.

Lexeme

- A lexeme is a sequence of characters in the source program that matched by the pattern for a token.

The Role of the Lexical Analyzer



Pattern	Token	Lexeme
while, for, if, else, int	Keyword	while for if else int
Letter followed by zero or more instance of Letter or Digit	Identifier	a, mark1, length, emp123, s567
Digit followed by zero or more instance of Digit	Number	79, 22, 4, 2021, 123456
Any symbols between “ and “ except “	Literal or String	“Welcome”, “Presidency University”, “Bengaluru”

1. One token for each keyword. The pattern for a keyword is the same as the keyword itself.
2. Tokens for the 1 operators, either individually or in classes.
3. One token representing all identifiers.
4. One or more tokens representing constants, such as numbers and literal strings
5. Tokens for each punctuation symbol, such as left and right parentheses, comma, and semicolon.

The Role of the Lexical Analyzer



Attributes for Tokens

- When more than one lexeme can match a pattern, the lexical analyzer must provide the subsequent compiler phases additional information about the particular lexeme that matched.
- For example, the pattern for token number matches both 0 and 1, but it is extremely important for the code generator to know which lexeme was found in the source program.
- Thus, in many cases the lexical analyzer returns to the parser not only a token name, but an attribute value that describes the lexeme represented by the token;
- The token name influences parsing decisions, while the appropriate attribute value for an identifier is a pointer to the symbol-table entry for that identifier.

<Token Name, Attribute-Value>

<Identifier, 2001>

<Number, 123>

<IF>

<+>

<{>

The Role of the Lexical Analyzer



Lexical Error

- It is hard for a lexical analyzer to tell source-code error.
- For instance, if the string **whiel** is encountered for the first time in a C program in the context:

whiel (Mark >= 90)

a lexical analyzer cannot tell whether **whiel** is a misspelling of the keyword **while** or an undeclared function identifier. Since **whiel** is a **valid lexeme** for the token id, the lexical analyzer must return the token id to the parser.

- However, 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. The **simplest recovery strategy** is "**panic mode**" recovery.
- In Panic Mode, We delete successive characters from the remaining input, until the lexical analyzer can find a well-formed token.

The Role of the Lexical Analyzer

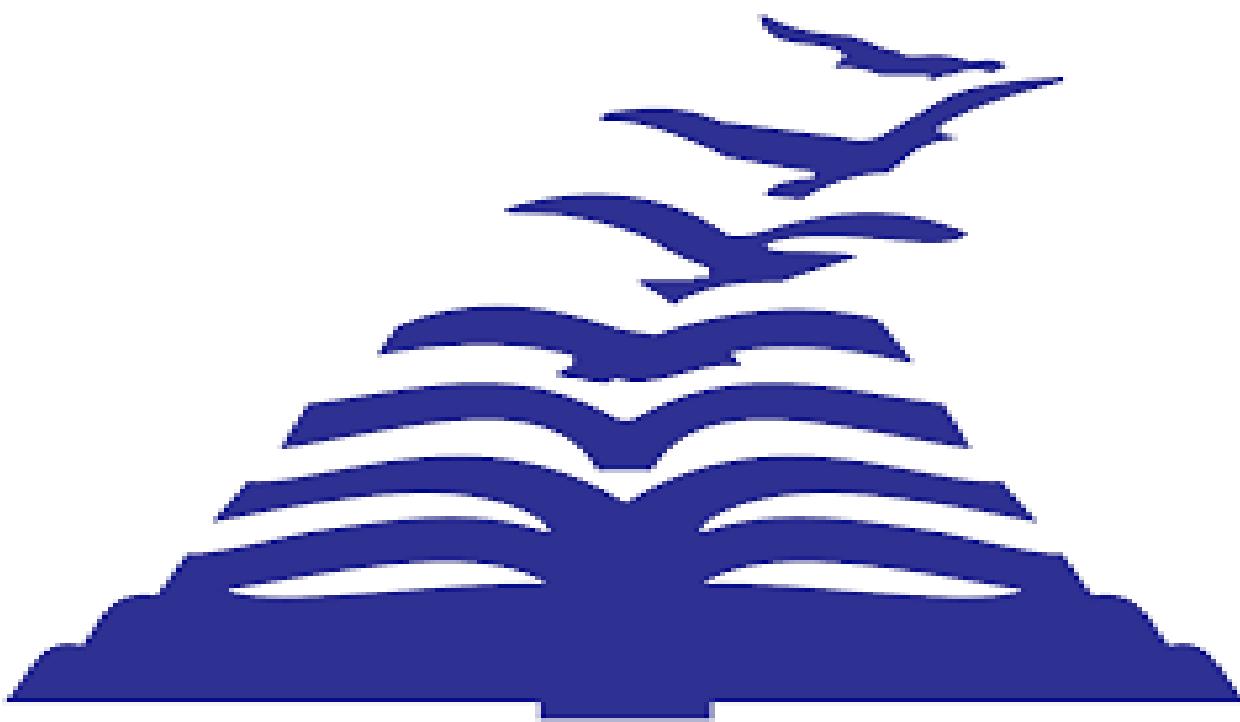


Lexical Error

Other possible error-recovery actions are:

1. Delete one character from the remaining input.
2. Insert a missing character into the remaining input.
3. Replace a character by another character.
4. Transpose two adjacent characters.

A more general correction strategy is to **find the smallest number of transformations** needed to convert the source program into one that **consists only of valid lexemes**, but this approach is considered too expensive in practice to be worth the effort.



Presidency University, Bengaluru

RECOGNITION OF TOKENS



Recognition of Tokens

- In the previous section we learned how to express patterns using regular expressions.
- Now, we must study how to take the patterns for all the needed tokens and build a piece of code that examines the input string and finds a prefix that is a lexeme matching one of the patterns.
- Consider the following example:

stmt $\rightarrow \text{if expr } \textbf{then} \text{ stmt}$
 $\rightarrow \text{if expr } \textbf{then} \text{ stmt } \textbf{else} \text{ stmt}$
 $\rightarrow \epsilon$

expr $\rightarrow \text{term } \textbf{relop} \text{ term}$
 $\rightarrow \text{term}$

term $\rightarrow \text{id}$
 $\rightarrow \text{number}$

Recognition of Tokens



- The grammar describes a simple form of branching statements and conditional expressions.
- The terminals of the grammar, which are **if**, **then**, **else**, **relop**, **id**, and **number**, are the names of tokens as far as the lexical analyzer is concerned. The patterns for these tokens are described using regular definitions.

Letter	→	[A-Za-z]
Digit	→	[0-9]
ID	→	{Letter} ({Letter} {Digit})*
NUMBER	→	{Digit}+
IF	→	if
ELSE	→	else
THEN	→	then
RELOP	→	< <= > >= == !=

- To simplify matters, we make the common assumption that keywords are also reserved words: that is, they are not identifiers, even though their lexemes match the pattern for identifiers.



Recognition of Tokens

In addition, we assign the lexical analyzer the job of stripping out white-space, by recognizing the "token" **ws** defined by:

$$\text{ws} \rightarrow (\text{blank} \mid \text{tab} \mid \text{newline})^+$$

Here, **blank**, **tab**, and **newline** are abstract symbols that we use to express the ASCII characters of the same names.

Token **ws** is different from the other tokens in that, when we recognize it, we do not return it to the parser.

Recognition of Tokens



The following table shows, for each lexeme or family of lexemes, which token name is returned to the parser and what attribute value.

LEXEMES	TOKEN NAME	ATTRIBUTE VALUE
Any ws	-	-
if	if	-
then	then	-
else	else	-
Any Id	id	Pointer to Symbol Table Entry
Any Number	number	Value
<	relop	LT
<=	relop	LE
>	relop	GT
>=	relop	GE
==	relop	EQ
!=	relop	NE

Recognition of Tokens - Transition Diagrams



- As an intermediate step in the construction of a lexical analyzer, we first convert patterns into stylized flowcharts, called "**transition diagrams**".
- Transition diagrams have a collection of nodes or **circles, called states**.
- Each state represents a condition that could occur during the process of scanning the input looking for a lexeme that matches one of several patterns.
- Edges are directed from one state of the transition diagram to another. Each edge is labeled by a symbol or set of symbols.

Recognition of Tokens - Transition Diagrams



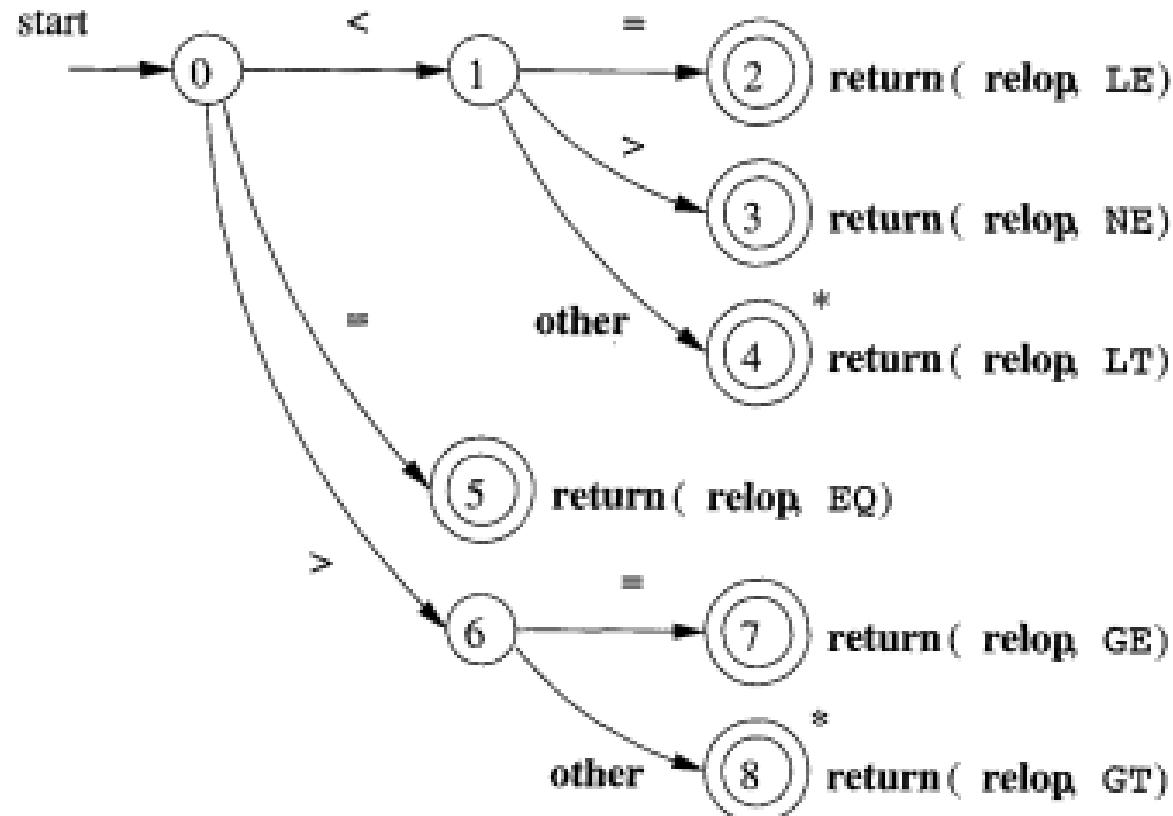
Some important conventions about transition diagrams are:

- One state is designated the **start state, or initial state**; it is indicated by an edge, labeled "start," entering from nowhere. The transition diagram always begins in the start state before any input symbols have been read.
- Certain states are said to be **accepting**, or **final**. We always indicate an accepting state by a **double circle**, and if there is an action to be taken — typically returning a token and an attribute value to the parser.
- In addition, if it is necessary to retract the forward pointer one position then we shall additionally **place a *** near that accepting state.

Recognition of Tokens - Transition Diagrams



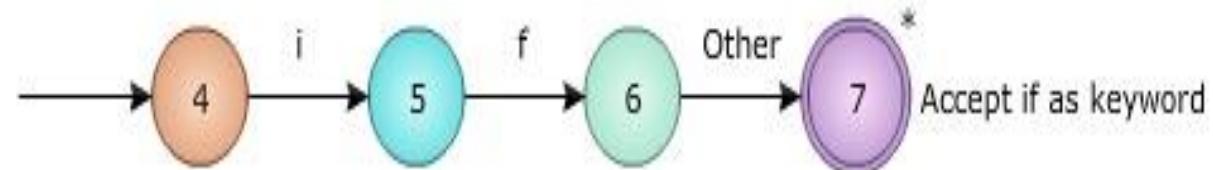
Transition diagram that recognizes the lexemes matching the token **relop**



Recognition of Tokens - Transition Diagrams



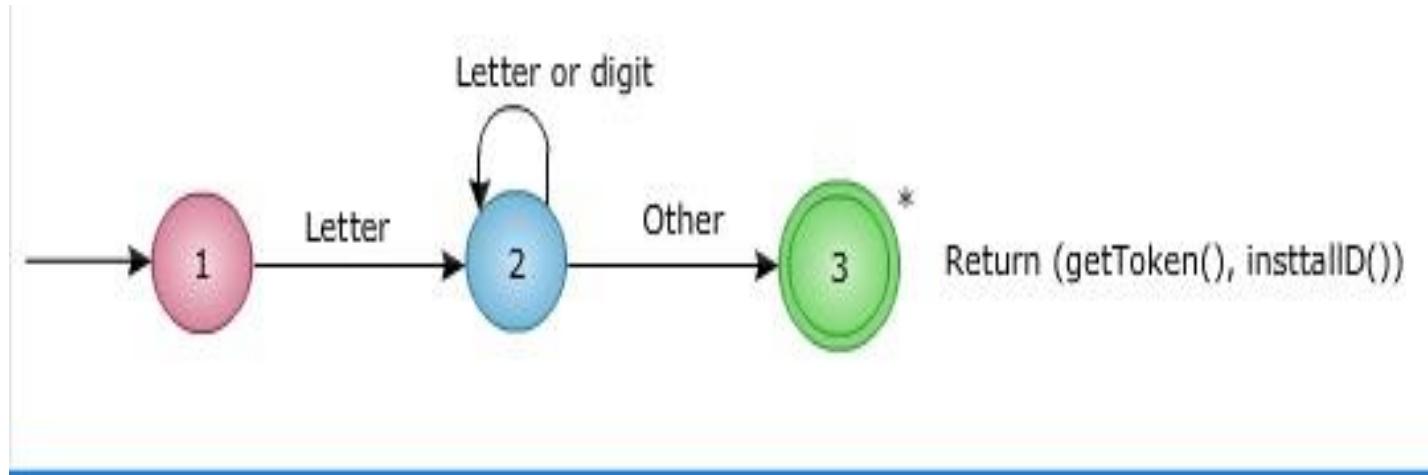
Recognition of Reserved Words (IF)



Recognition of Tokens - Transition Diagrams



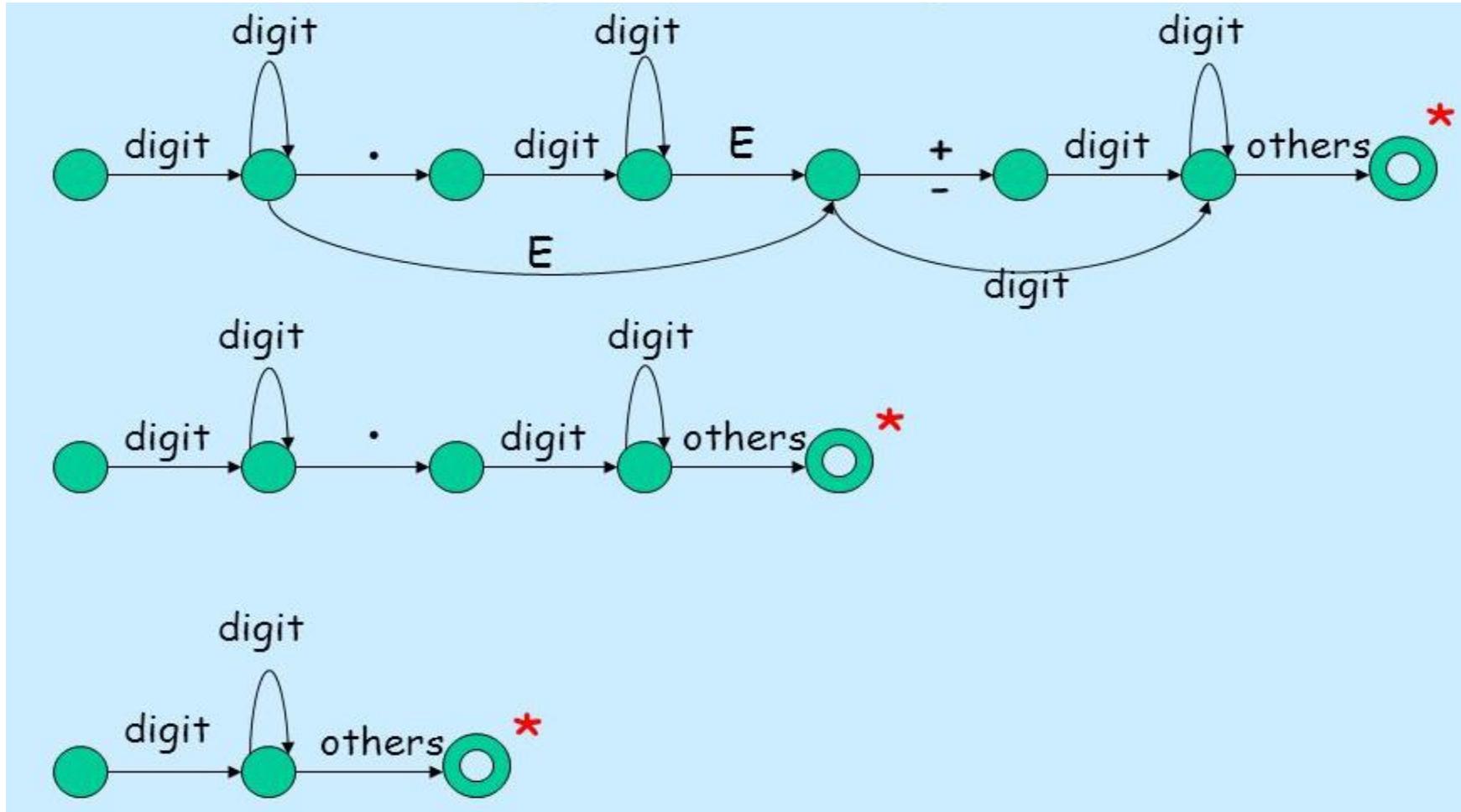
Recognition of Identifiers



Recognition of Tokens - Transition Diagrams



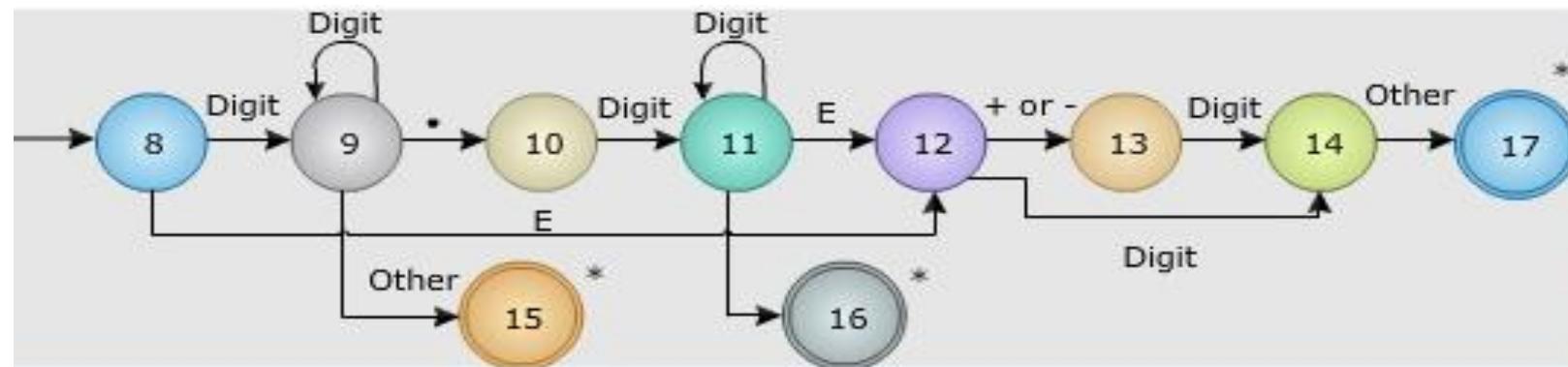
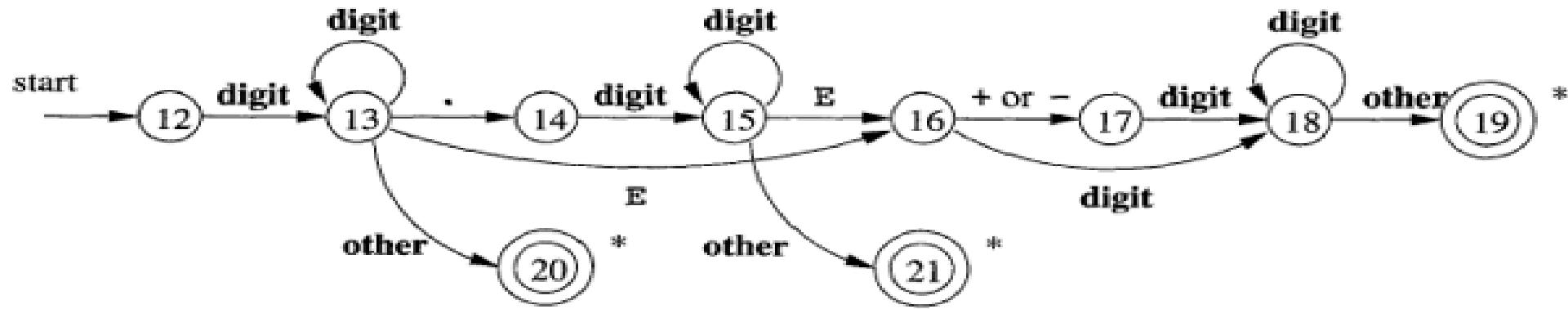
Recognition of Number



Recognition of Tokens - Transition Diagrams



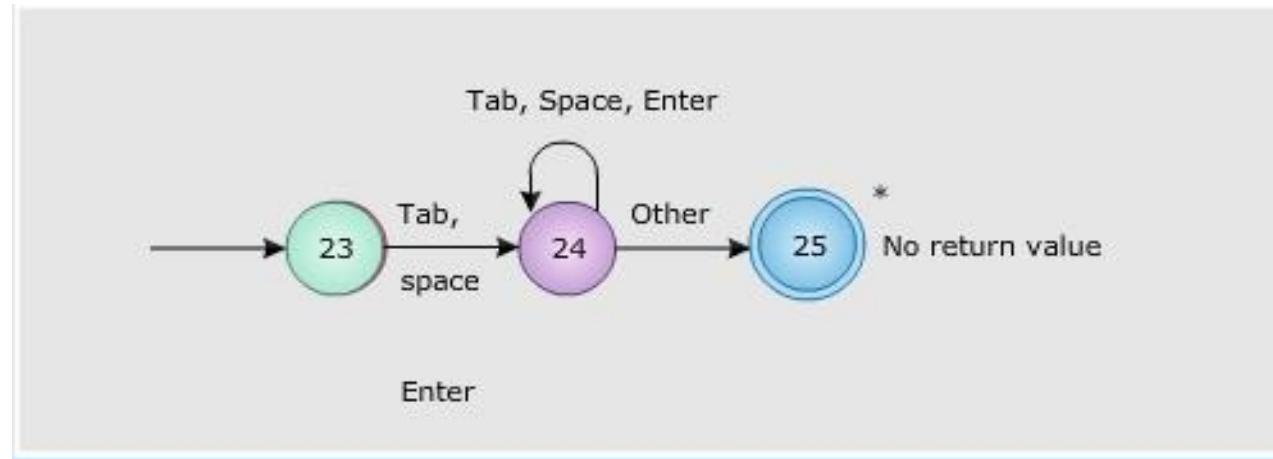
Recognition of Number



Recognition of Tokens - Transition Diagrams



Recognition of White Spaces



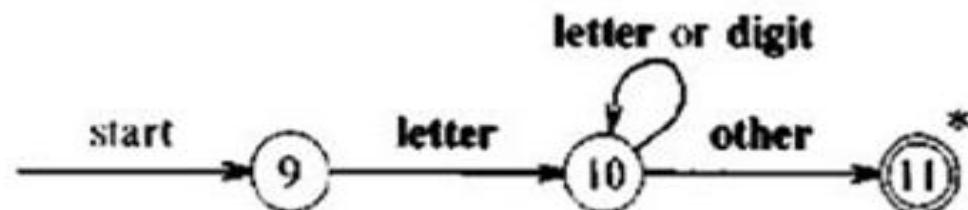
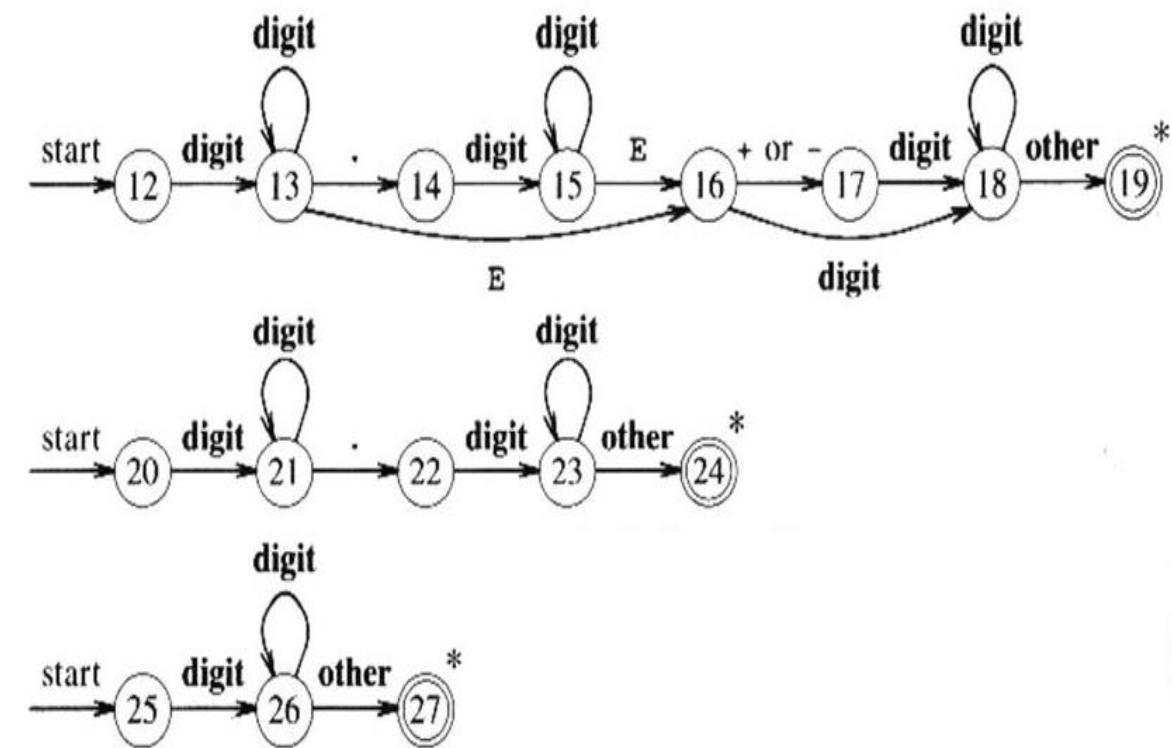
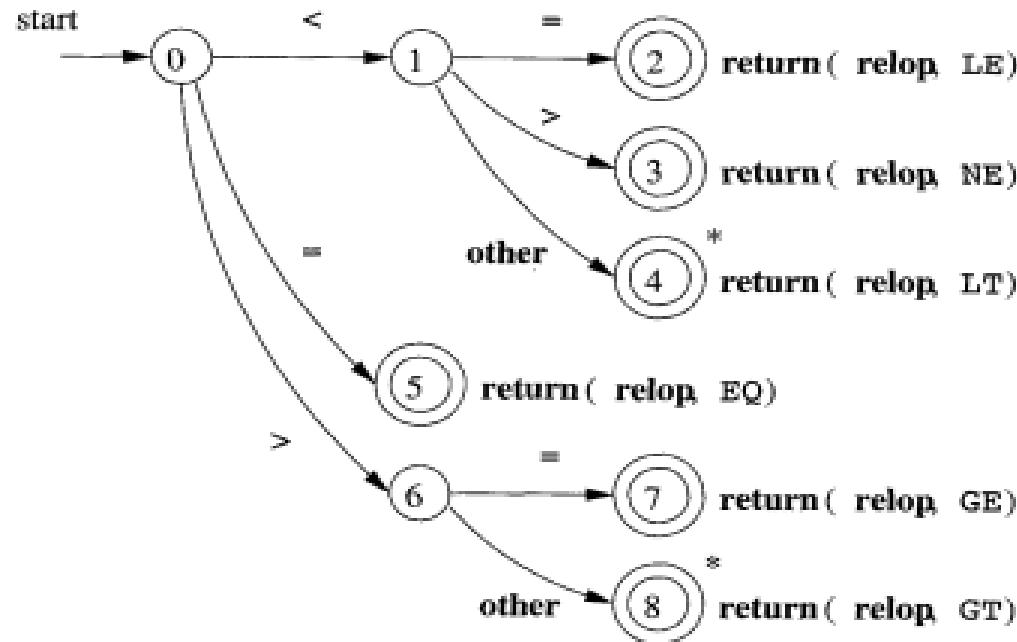
Recognition of Tokens - Transition Diagrams



There are two ways that we can handle reserved words that look like identifiers:

- Install the reserved words in the symbol table initially. A field of the symbol-table entry indicates that these strings are never ordinary identifiers, and tells which token they represent. When we find an identifier, a call to `installD` places it in the symbol table if it is not already there and returns a pointer to the symbol-table entry for the lexeme found.
- Create separate transition diagrams for each keyword.

Recognition of Tokens – Implementing a Transition Diagrams



Recognition of Tokens – Implementing a Transition Diagrams



```
token nexttoken()
{
    while(1) {
        switch (state) {
            case 0: c = nextchar();
                /* c is lookahead character */
                if (c==blank || c==tab || c==newline) {
                    state = 0;
                    lexeme_beginning++;
                    /* advance beginning of lexeme */
                }
                else if (c == '<') state = 1;
                else if (c == '=') state = 5;
                else if (c == '>') state = 6;
                else state = fail();
                break;
            .../* cases 1-8 here */
        }
    }
}
```

```
case 9: c = nextchar();
        if (isletter(c)) state = 10;
        else state = fail();
        break;
case 10: c = nextchar();
        if (isletter(c)) state = 10;
        else if (isdigit(c)) state = 10;
        else state = 11;
        break;
case 11: retract(1); install_id();
        return ( gettoken() );
        .../* cases 12-24 here */
case 25: c = nextchar();
        if (isdigit(c)) state = 26;
        else state = fail();
        break;
case 26: c = nextchar();
        if (isdigit(c)) state = 26;
        else state = 27;
        break;
case 27: retract(1); install_num();
        return ( NUM );
    }
}
}
```

Recognition of Tokens – Implementing a Transition Diagrams



```
int state = 0, start = 0;  
int lexical_value;  
    /* to "return" second component of token */  
  
int fail()  
{  
    forward = token_beginning;  
    switch (start) {  
        case 0:    start = 9; break;  
        case 9:    start = 12; break;  
        case 12:   start = 20; break;  
        case 20:   start = 25; break;  
        case 25:   recover(); break;  
        default:   /* compiler error */  
    }  
    return start;  
}
```

Recognition of Tokens - Finite Automata



We compile a regular expression into a recognizer by constructing a generalized transition diagram called a **finite automaton**.

Finite automata are **recognizers**; they simply say "**yes**" or "**no**" about each possible input string.

Finite automata come in two flavors:

- (a) **Nondeterministic finite automata (NFA)** have no restrictions on the labels of their edges. A symbol can label several edges out of the same state, and ϵ , the empty string, is a possible label.
- (b) **Deterministic finite automata (DFA)** have, for each state, and for each symbol of its input alphabet exactly one edge with that symbol leaving that state.

Recognition of Tokens - Finite Automata



Non-Deterministic Finite Automata (NFA)

Definition: A NFA is a mathematical model that consists of 5-tuple, $M=(S, \Sigma, S_0, F, \delta)$ where,

- **S** is the set of finite states
- Σ is the set of input symbols (The input symbol alphabet)
- **S_0** is the initial state
- **F** is the set of all accepting states or final states.
- **δ** is the transition function **move** that maps state-symbol pairs to set of states

A Deterministic Finite Automaton (DFA) is a special case of an NFA where:

1. There are no moves on input ϵ , and
2. For each state s and input symbol a , there is exactly one edge out of s .

Recognition of Tokens - Finite Automata



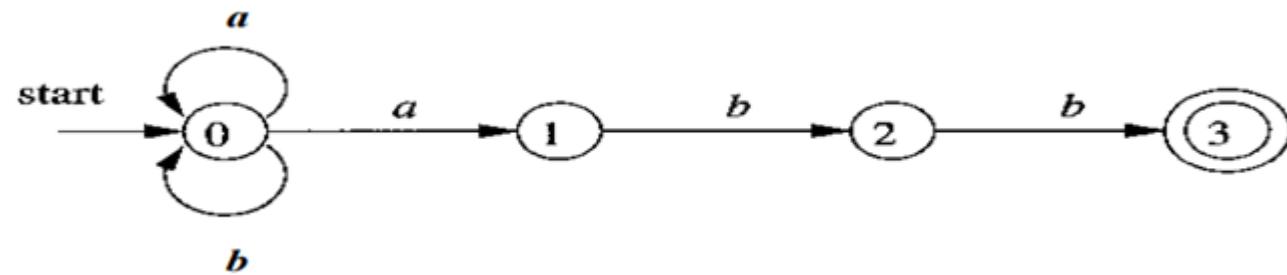
Difference between NFA and DFA

NFA	DFA
In NFA, more than one transition out of a state may be possible on the same input symbol.	For each state, and for each symbol of its input alphabet exactly one edge with that symbol leaving that state.
There is a Time tradeoff, it is a Slow Recognizer	There is a Time tradeoff, it is a Faster Recognizer
NFA may have ϵ transitions.	In DFA, There is no ϵ transitions.
There is a Space tradeoff, it occupies less space	It occupies more space to execute (Much bigger than NFA)

Recognition of Tokens - Finite Automata



- NFA or DFA can be represented by a **transition graph**, where the nodes are states and the labeled edges represent the transition function. There is an edge labeled a from state s to state t if and only if t is one of the next states for state s and input a .



- We can also represent an NFA or DFA by a **transition table**, whose rows correspond to states, and whose columns correspond to the input symbols.

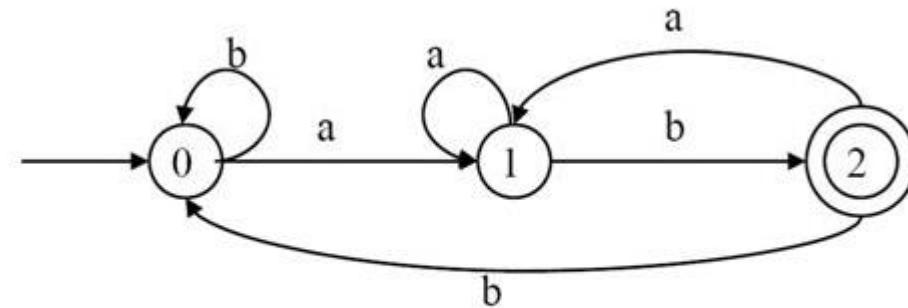
STATE	INPUT SYMBOL	
	a	b
0	{0, 1}	{0}
1	-	{2}
2	-	{3}

Recognition of Tokens – Regular Expression to DFA



Simulating a DFA

```
s = s0;  
c = nextChar();  
while ( c != eof ) {  
    s = move(s, c);  
    c = nextChar();  
}  
if ( s is in F ) return "yes";  
else return "no";
```



DFA for $(a|b)^* a b$

Recognition of Tokens – Regular Expression to DFA



Converting a Regular Expression Directly to a DFA

Algorithm: Construction of a DFA from a regular expression r .

INPUT: A regular expression r .

OUTPUT: A DFA D that recognizes $L(r)$.

METHOD :

1. Construct a **syntax tree T from the augmented regular expression $(r)\#$** .
2. Compute **nullable**, **firstpos**, **lastpos** and **followpos** for T .
3. **Construct Dstates**, the set of states of DFA D , and D_{tran} , the transition function for D .
The states of D are sets of positions in T . Initially, each state is "unmarked," and a state becomes "marked" just before we consider its out-transitions. The start state of D is $\text{firstpos}(n_0)$, where node n_0 is the root of T . The accepting states are those containing the position for the endmarker symbol $\#$.

Recognition of Tokens – Regular Expression to DFA



Rules for computing nullable, firstpos and lastpos

Node n	$\text{nullable}(n)$	$\text{firstpos}(n)$	$\text{lastpos}(n)$
A leaf labeled by ϵ	true	\emptyset	\emptyset
A leaf with position i	false	$\{i\}$	$\{i\}$
$n = c_1 \mid c_2$	$\text{nullable}(c_1)$ or $\text{nullable}(c_2)$	$\text{firstpos}(c_1) \cup \text{firstpos}(c_2)$	$\text{lastpos}(c_1) \cup \text{lastpos}(c_2)$
$n = c_1 \cdot c_2$	$\text{nullable}(c_1)$ and $\text{nullable}(c_2)$	if ($\text{nullable}(c_1)$) $\text{firstpos}(c_1) \cup \text{firstpos}(c_2)$ else $\text{firstpos}(c_1)$	if ($\text{nullable}(c_2)$) $\text{lastpos}(c_1) \cup \text{lastpos}(c_2)$ else $\text{lastpos}(c_2)$
$n = c_1^*$	<i>true</i>	$\text{firstpos}(c_1)$	$\text{lastpos}(c_1)$

Recognition of Tokens – Regular Expression to DFA



Rules for Computing followpos:

There are only two ways that a position of a regular expression can be made to follow another

1. If n is a **cat-node** with left child C_1 and right child C_2 , then for every position i in $\text{lastpos}(C_1)$, all positions in $\text{firstpos}(C_2)$ are in $\text{followpos}(i)$.
2. If n is a **star-node**, and i is a position in $\text{lastpos}(n)$, then all positions in $\text{firstpos}(n)$ are in $\text{followpos}(i)$.

Recognition of Tokens – Regular Expression to DFA



Construct DFA for the regular expression $(a|b)^*abb$

Step1:

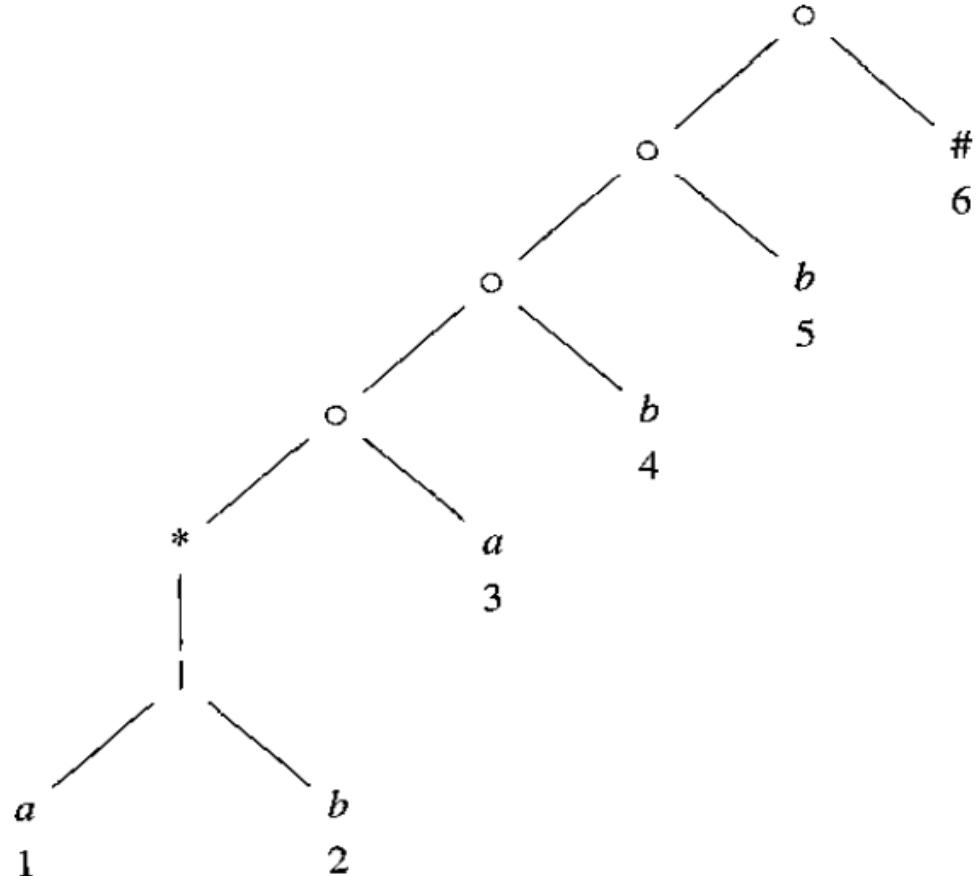
Convert the regular expression r into augmented regular expression $(r)\#$.

$(a|b)abb\#$

Step2:

Construct a syntax tree T from the augmented regular expression $(r)\#$

$(a|b) a b b \#$



Recognition of Tokens – Regular Expression to DFA

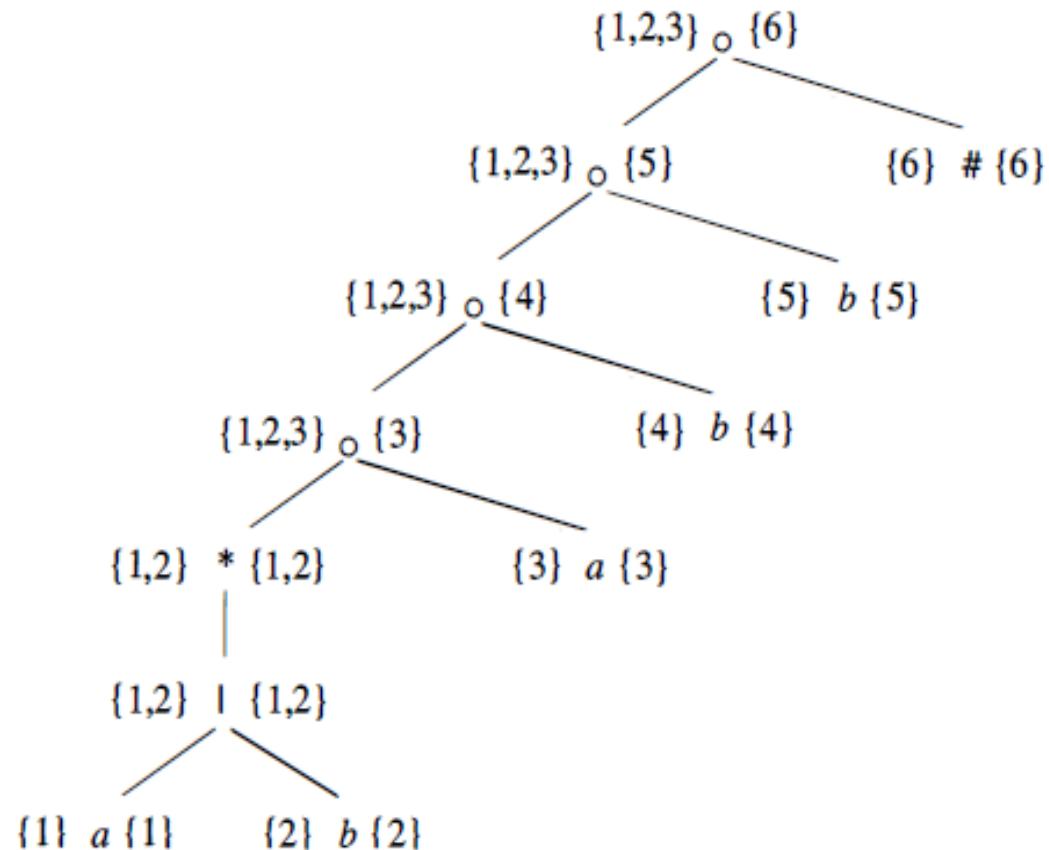


Construct DFA for the regular expression $(a|b)^*abb$

Step3:

Compute nullable, firstpos
and lastpos

NODE	n	$followpos(n)$
1		{1,2,3}
2		{1,2,3}
3		{4}
4		{5}
5		{6}
6		0



Recognition of Tokens – Regular Expression to DFA



Construct DFA for the regular expression $(a \mid b)^*abb$

Step4:

Construct Dstates

The value of firstpos for the root of the tree is $\{1,2,3\}$, so this set is the start state of D.

$$A = \{1,2,3\}$$

$$Dtran[A, a] = \text{followpos}(1) \cup \text{followpos}(3) = \{1,2,3,4\} = B$$

$$Dtran[A, b] = \text{followpos}(2) = \{1,2,3\} = A$$

$$Dtran[B, a] = \text{followpos}(1) \cup \text{followpos}(3) = \{1,2,3,4\} = B$$

$$Dtran[B, b] = \text{followpos}(2) \cup \text{followpos}(4) = \{1,2,3,5\} = C$$

$$Dtran[C, a] = \text{followpos}(1) \cup \text{followpos}(3) = \{1,2,3,4\} = B$$

$$Dtran[C, b] = \text{followpos}(2) \cup \text{followpos}(5) = \{1,2,3,6\} = D$$

$$Dtran[D, a] = \text{followpos}(1) \cup \text{followpos}(3) = \{1,2,3,4\} = B$$

$$Dtran[D, b] = \text{followpos}(1) = \{1,2,3\} = A$$

Recognition of Tokens – Regular Expression to DFA



Construct DFA for the regular expression $(a \mid b)^*abb$

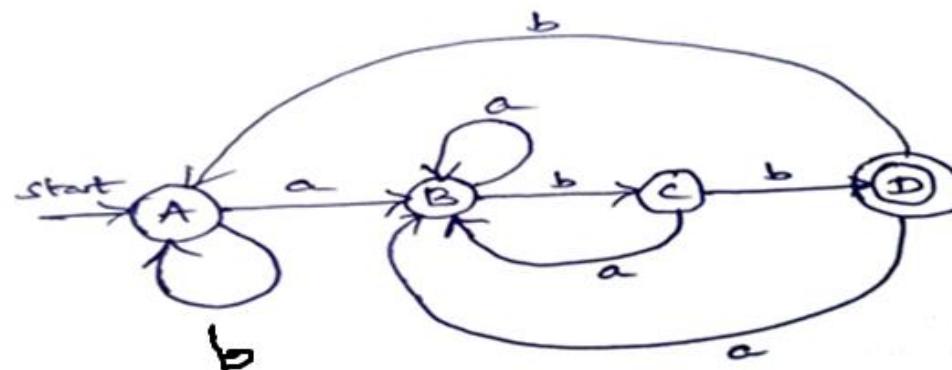
Step5:

Construct DFA Table

states	Input string	
	a	b
A	B	A
B	B	C
C	B	D
D	B	A

Step6:

Draw DFA Diagram



Recognition of Tokens – Regular Expression to DFA



Construct DFA for the following regular expression

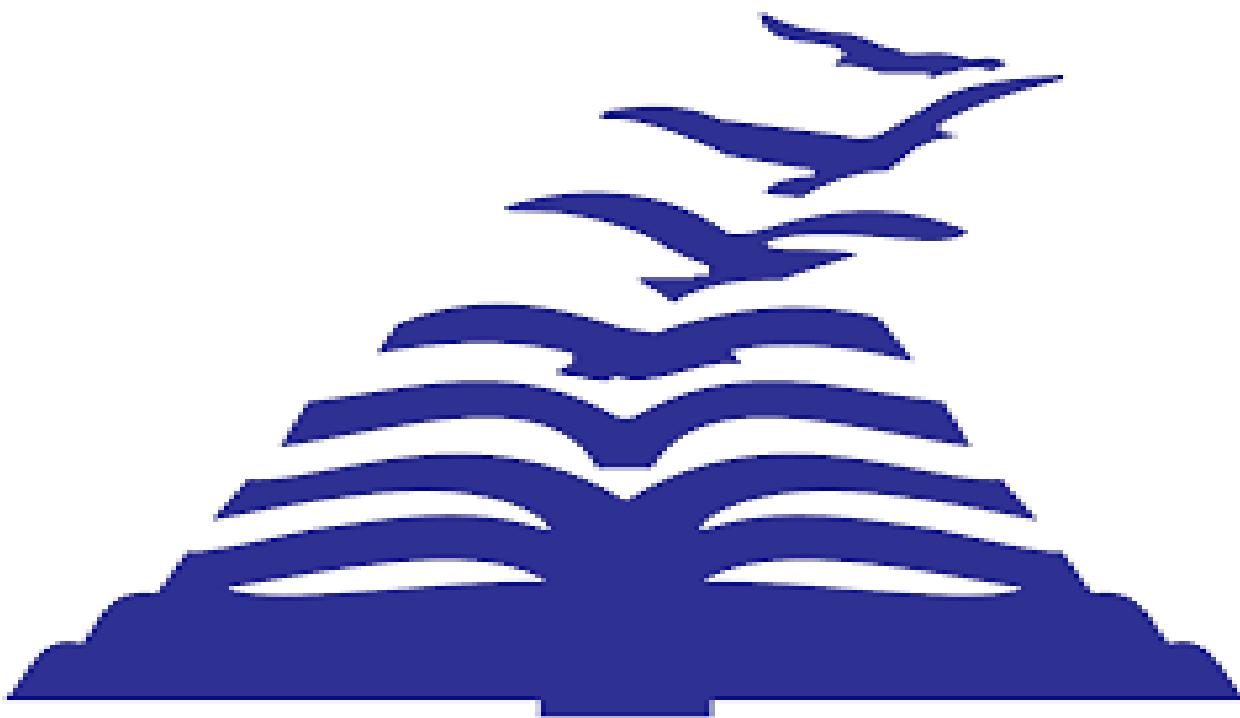
1. $a^*(a \mid b)abb(a \mid b)^*$

2. $ab(a \mid b)(a \mid b)^*b^*a$

3. $(a \mid b \mid c)c^*abb^*$

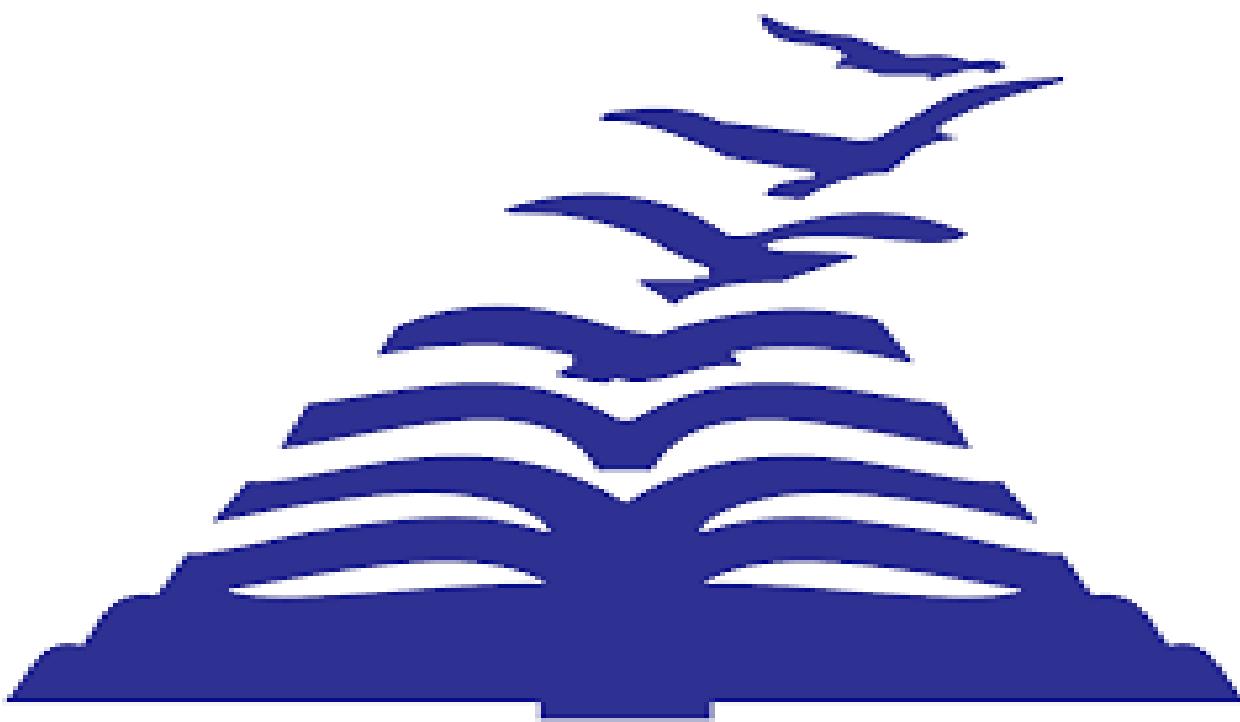
4. $(0 \mid 1)101(1 \mid 0)^*$

5. $1^*0^*10100^*$



Presidency University, Bengaluru

Input Buffering and Specification of Tokens



Presidency University, Bengaluru

Input Buffering



Input Buffering

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

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

1. By using a lexical-analyzer generator, such as **lex compiler** to produce the lexical analyzer from a regular expression based specification. In this, the generator provides routines for reading and buffering the input.
2. By writing the lexical analyzer in a **conventional systems-programming language**, using I/O facilities of that language to read the input.
3. By writing the lexical analyzer in **assembly language** and explicitly managing the reading of input.



Input Buffering - Buffer Pairs

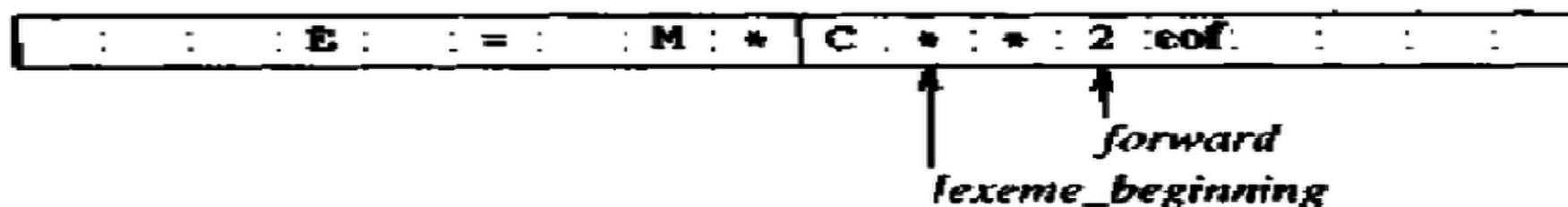
Two Techniques are used for Input Buffering:

- **Buffer Pairs**
- **Sentinels**

Buffer Pairs

Because of large amount of time consumption in moving characters, **specialized buffering techniques** have been developed to reduce the amount of overhead required to process an input character.

Fig shows the buffer pairs which are used to hold the input data.





Input Buffering - Buffer Pairs

- Buffer Pairs scheme Consists of **two buffers**, each consists of N-character size which are reloaded alternatively.
- N-Number of characters on one disk block, e.g., **1024 or 4096**.
- N characters are read from the input file to the buffer using one **system read command** rather than invoking a read command for each character.
- **eof** is inserted at the end if the number of characters is less than N.

Two pointers to the input are maintained:

- Pointer **lexeme_beginning**, marks the beginning of the current lexeme, whose extent we are attempting to determine.
- Pointer **forward** scans ahead until a pattern match is found.

Once the next lexeme is determined, forward is set to the character at its right end.

The string of characters between the two pointers is the current lexeme.

Input Buffering - Buffer Pairs



Disadvantages of this scheme

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

Code to advance forward pointer

```
if forward at end of the 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
    forward = forward + 1;
```

Input Buffering - Sentinels



Sentinels:

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

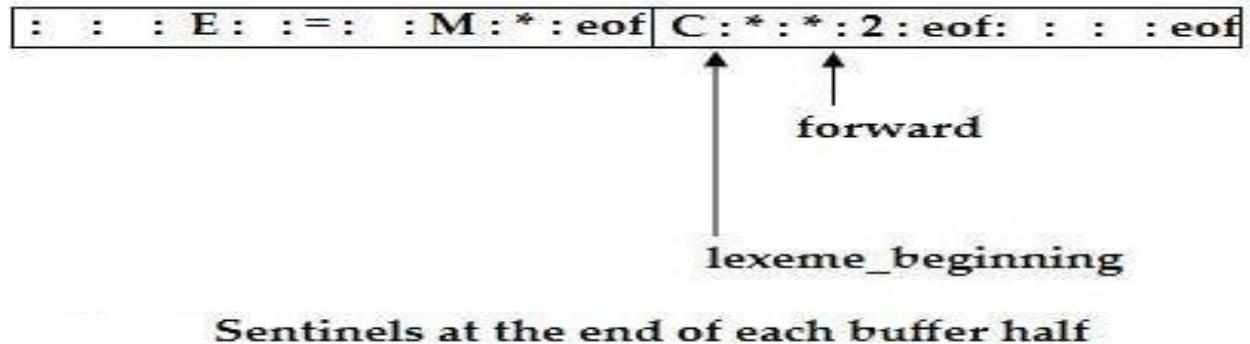
Test 1: For end of buffer.

Test 2: To determine what character is read.

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



Input Buffering - Sentinels



forward = forward + 1;

If **forward = eof** then begin

 if **forward** at end of the 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 /* eof within a buffer signifying end of input */

terminate lexical analysis;

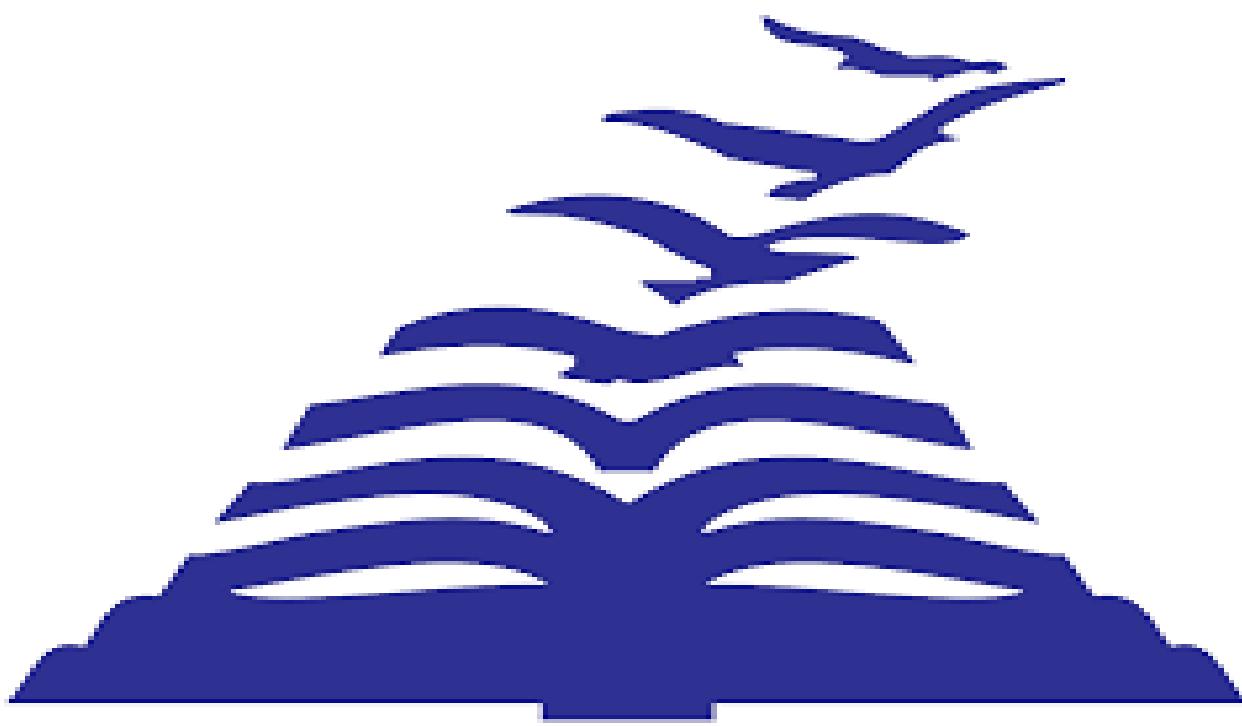
end

Input Buffering - Sentinels



Advantages

- **Most of the time, It performs only one test** to see whether forward pointer points to an *eof*.
- Only when it reaches the end of the buffer half or *eof*, it performs more tests.



Presidency University, Bengaluru

Specification of Tokens

Specification of Tokens



Regular expressions are an important notation for specifying lexeme patterns.

Specification of Token consists of

- **Strings and Languages**
- **Operations on Languages**
- **Regular Expressions**
- **Regular Definitions**
- **Notational Shorthands**
- **Nonregular Sets**



Strings and Languages

- **An alphabet is any finite set of symbols.** Typical examples of symbols are letters, digits, and punctuation.
- The set $\{0,1\}$ is the **binary alphabet**.
- **ASCII** is an important example of an alphabet; it is used in many software systems.
- **Unicode**, which includes approximately 100,000 characters from alphabets around the world, is another important example of an alphabet.

Specification of Tokens - Strings and Languages



Strings

- A **string** over an alphabet is a finite sequence of symbols drawn from that alphabet.
 - In language theory, the terms "**sentence**" and "**word**" are often used as synonyms for "string."
 - The **length of a string s**, usually written $|s|$, is the number of occurrences of symbols in s. For example, banana is a string of length six. The **empty string, denoted ϵ , is the string of length zero.**
 - If x and y are strings, then the **concatenation** of x and y, denoted xy , is the string formed by appending y to x.

Languages

- A **language** is any set of strings over some fixed alphabet.
- Abstract languages like \emptyset , the empty set, or $\{\epsilon\}$, the set containing only the empty string.

Specification of Tokens - Strings and Languages



Terms for parts of a string

TERM	DEFINITION
prefix of s	A string obtained by removing zero or more trailing symbols of string s; ban is a prefix of banana.
suffix of s	A string formed by deleting zero or more of the leading symbols of s; nana is a suffix of banana.
substring of s	A string obtained by deleting a prefix and a suffix from s; nan is a substring of banana. Every prefix and every suffix of s is a substring of s, but not every substring of s is a prefix or a suffix of s. For every string s, both s and e are prefixes, suffixes, and substrings of s.
proper prefix, suffix, or substring of s	Any nonempty string x that is, respectively, a prefix, suffix, or substring of s such that $s \neq x$.
subsequence of s	Any string formed by deleting zero or more not necessarily contiguous symbols from s; baaa is a subsequence of banana.

Specification of Tokens - Operations on Languages



Operations on Languages

- There are several important operations that can be applied to Languages, For lexical analysis, the most important operations on languages are **union**, **concatenation**, and **closure**.

Definitions of operations on languages

OPERATION	DEFINITION
<i>union of L and M written $L \cup M$.</i>	$L \cup M = \{ s \mid s \text{ is in } L \text{ or } s \text{ is in } M \}$
<i>concatenation of L and M written LM</i>	$LM = \{ st \mid s \text{ is in } L \text{ and } t \text{ is in } M \}$
<i>Kleene closure of L written L^*</i>	$L^* = \bigcup_{i=0}^{\infty} L^i$ <i>L^* denotes “zero or more concatenations of” L.</i>
<i>positive closure of L written L^+</i>	$L^+ = \bigcup_{i=1}^{\infty} L^i$ <i>L^+ denotes “one or more concatenations of” L.</i>

Specification of Tokens - Regular Expressions



Regular Expressions

Regular expressions are an important notation for specifying lexeme patterns.

Identifier - **Letter (Letter | Digit)***

Integer - **Digit (Digit)***

Float - **Digit (Digit)* . Digit (Digit)***

Specification of Tokens - Regular Expressions



The regular expressions are built up out of smaller regular expressions using set of defining rules

or

The regular expressions are built recursively out of smaller regular expressions.

Each regular expression r denotes a language $L(r)$, which is also defined recursively from the languages denoted by r 's subexpressions.

- a) The unary operator $*$ has highest precedence and is left associative.
- b) Concatenation has second highest precedence and is left associative.
- c) $|$ has lowest precedence and is left associative.

A language that can be defined by a regular expression is called a **regular set**.

Specification of Tokens - Regular Expressions



If two regular expressions r and s denote the same regular set, we say they are equivalent and write $r = s$. For instance, $(a|b) = (b|a)$.

There are a number of algebraic laws for regular expressions.

AXIOM	DESCRIPTION
$r s = s r$	is commutative
$r (s t) = (r s) t$	is associative
$r(st) = (rs)t$	Concatenation is associative
$r(s t) = rs rt$ $(s t)r = sr tr$	Concatenation distributes over
$\epsilon r = r\epsilon = r$	ϵ is the identity for concatenation
$r^* = (r \epsilon)^*$	ϵ is guaranteed in a closure
$r^{**} = r^*$	* is idempotent

Specification of Tokens - Regular Definitions



For notational convenience, we may wish to give names to certain regular expressions and to define regular expressions using these names as if they were symbols. If Σ is an alphabet of basic symbols, then a regular definition is a sequence of definitions of the form:

$$d_1 \rightarrow r_1$$

$$d_2 \rightarrow r_2$$

.

.

.

$$d_n \rightarrow r_n$$

Where each d_i is a district name, and each r_i is a regular expression over the symbols in $\Sigma \cup \{d_1, d_2, \dots, d_{i-1}\}$, i.e., the basic symbols and the previously defined names.

Specification of Tokens - Regular Definitions



Letter → A | B | C | ... | Z | a | b | c | ... | z |

Digit → 0 | 1 | 2 | ... | 9 |

Identifier → {Letter} ({Letter} | {Digit})*

Specification of Tokens - Notational Shorthands



Certain constructs occur **so frequently in regular expressions** that it is convenient to introduce **notational shorthands** for them.

1. **One or more instances.** The unary, postfix operator **+** represents the positive closure of a regular expression and its language.

Two useful algebraic laws, $r^* = r^+ \mid \epsilon$ and $r^+ = rr^* = r^*r$ relate the Kleene closure and positive closure.

2. **Zero or one instance.** The unary postfix operator **?** means "zero or one occurrence." That is, $r?$ is equivalent to $r \mid \epsilon$.

3. **Character classes.**

The notation **[a-z]** represents

a | b | c | ... | z

The notation **[1-4]** represents

1 | 2 | 3 | 4

Specification of Tokens - Nonregular Sets



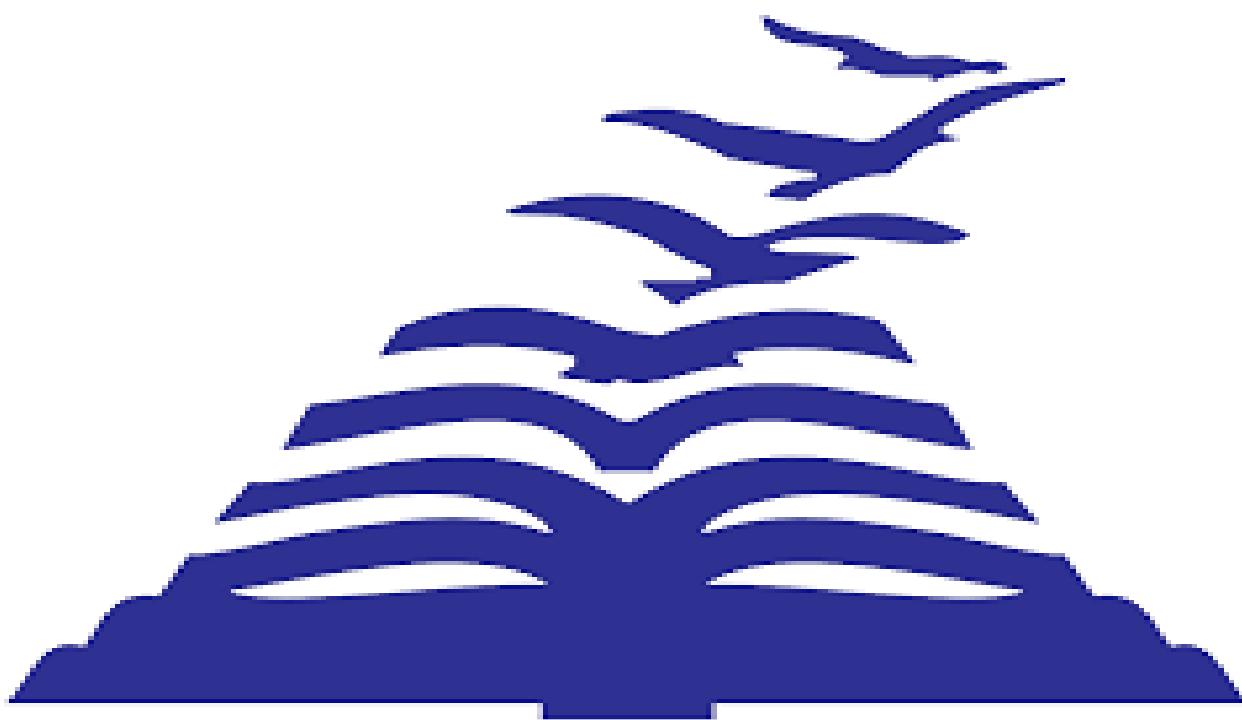
- Some languages cannot be described by any regular expressions.
- A language which cannot be described by any regular expression is a **non-regular set**.
- Example: The set of all strings of balanced parentheses and repeating strings cannot be described by a regular expression.
- This set can be specified by a context-free grammar

The set

{ **wcw | w is a strings of a's and b's}** }

cannot be denoted by regular expressions for repeating strings, nor can it be described by a CFG.

S → aSb | ab

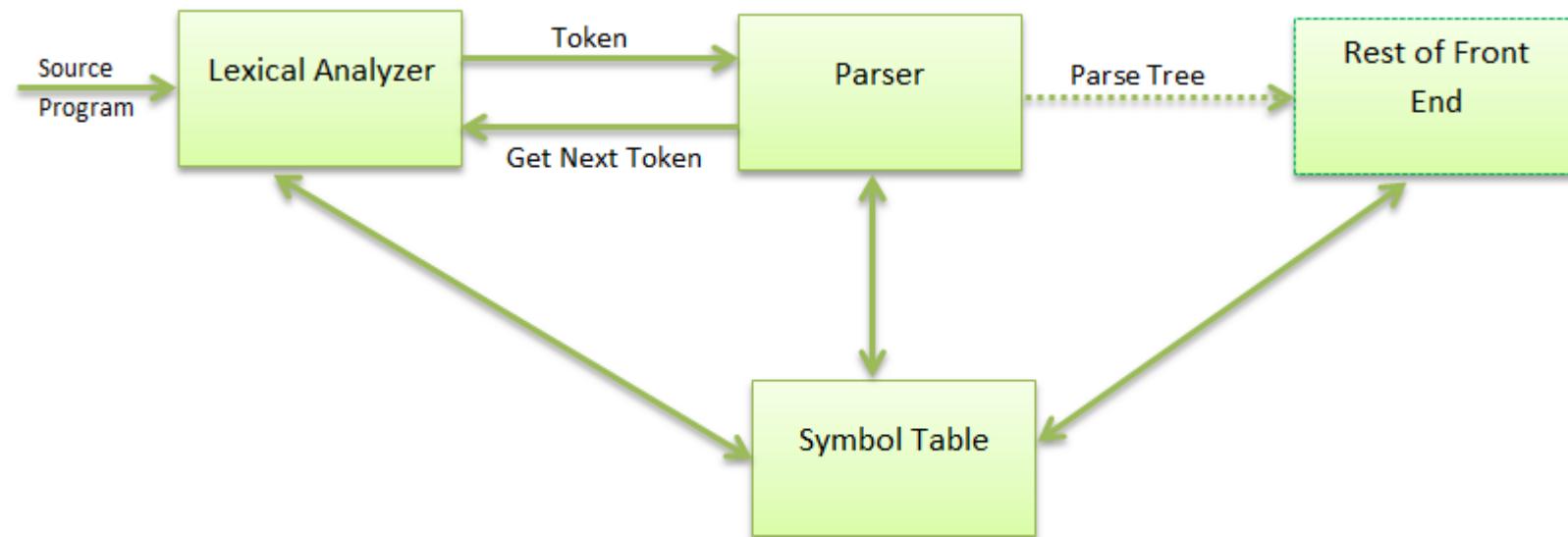


Presidency University, Bengaluru

Role of the Parser

Role of the Parser

- In our compiler model, the parser obtains a string of tokens from the lexical analyzer.
- Syntax Analyzer verifies that the string of token names can be generated by the grammar for the source language.
- We expect the parser to report any syntax errors in an intelligible fashion and to recover from commonly occurring errors to continue processing the remainder of the program.
- Conceptually, for well-formed programs, the parser constructs a parse tree and passes it to the rest of the compiler for further processing.

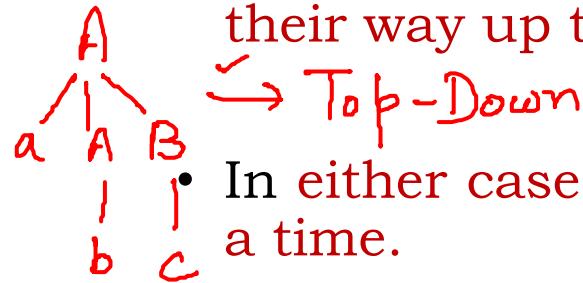


There are three general types of parsers for grammars:

- **Universal**
 - **Top-down**
 - **Bottom-up**
-
- Universal parsing methods such as the **Cocke-Younger-Kasami algorithm** and **Earley's algorithm** can parse any grammar. These general methods are, however, too inefficient to use in production compilers.
 - The methods commonly used in compilers can be classified as being either **top-down** or **bottom-up**.

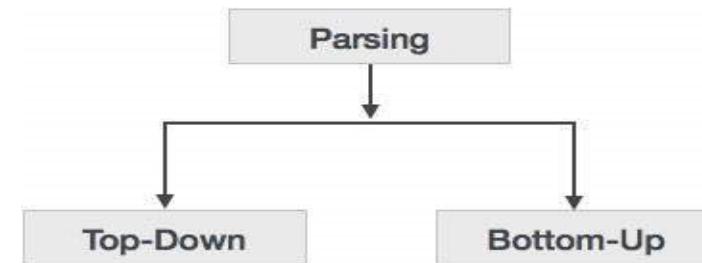
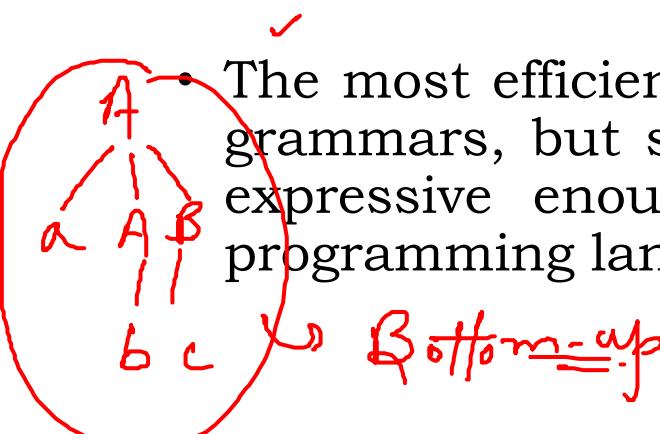
Types of Parsers

- Parsing can be defined as **top-down** or **bottom-up** based on how the parse-tree is constructed.
- As implied by their names, top-down methods build parse trees from the top (root) to the bottom (leaves), while bottom-up methods start from the leaves and work their way up to the root.



$w = \underline{a} \underline{b} \underline{c}$

- In either case, the input to the parser is scanned from left to right, one symbol at a time.



The **error handler** in a parser has goals that are simple to state but challenging to realize:

- **Report the presence of errors clearly and accurately.**
- **Recover from each error quickly enough to detect subsequent errors.**
- **Add minimal overhead to the processing of correct programs.**

- **Planning the error handling** right from the start can both **simplify the structure of a compiler and improve its handling of errors.**
- Another reason for emphasizing error recovery during parsing is that many errors appear syntactic, whatever their cause, and are exposed when **parsing cannot continue.**

Once an error is detected, how should the parser recover? Although no strategy has proven itself universally acceptable, a few methods have broad applicability.

- **Panic-Mode Recovery**
- **Phrase-Level Recovery**
- **Error Productions**
- **Global Correction**

- **Panic-Mode Recovery**
 - With this method, on discovering an error, the parser discards input symbols one at a time until one of a designated set of synchronizing tokens is found.
- **Phrase-Level Recovery**
 - On discovering an error, a parser may perform local correction on the remaining input; that is, it may replace a prefix of the remaining input by some string that allows the parser to continue. A typical local correction is to replace a comma by a semicolon, delete an extraneous semicolon, or insert a missing semicolon.
- **Error Productions**
 - A parser constructed from a grammar augmented by these error productions detects the anticipated errors when an error production is used during parsing.
- **Global Correction**
 - Ideally, we would like a compiler to make as few changes as possible in processing an incorrect input string. There are algorithms for choosing a minimal sequence of changes to obtain a globally least-cost correction.

Context Free Grammar

Reg → FA < NFA
DFA
CFG → PDA → Push Down Automata
CSG → LBA



By design, every programming language has **precise rules** that prescribe the **syntactic structure of well-formed programs**.

The syntax of programming language constructs can be specified by **context-free grammars** or **BNF (Backus-Naur Form) notation**.

Grammars offer significant benefits for both language designers and compiler writers.

- A grammar gives a precise, yet easy-to-understand, syntactic specification of a programming language.
int a b c; \Rightarrow
- From certain classes of grammars, we can construct automatically an efficient parser that determines the syntactic structure of a source program.
- The structure imparted to a language by a properly designed grammar is useful for translating source programs into correct object code and for detecting errors.
- A grammar allows a language to be evolved or developed iteratively, by adding new constructs to perform new tasks. These new constructs can be integrated more easily into an implementation that follows the grammatical structure of the language.

Grammars systematically describe the syntax of programming language constructs like expressions and statements.

A CFG can be used to generate strings in its language

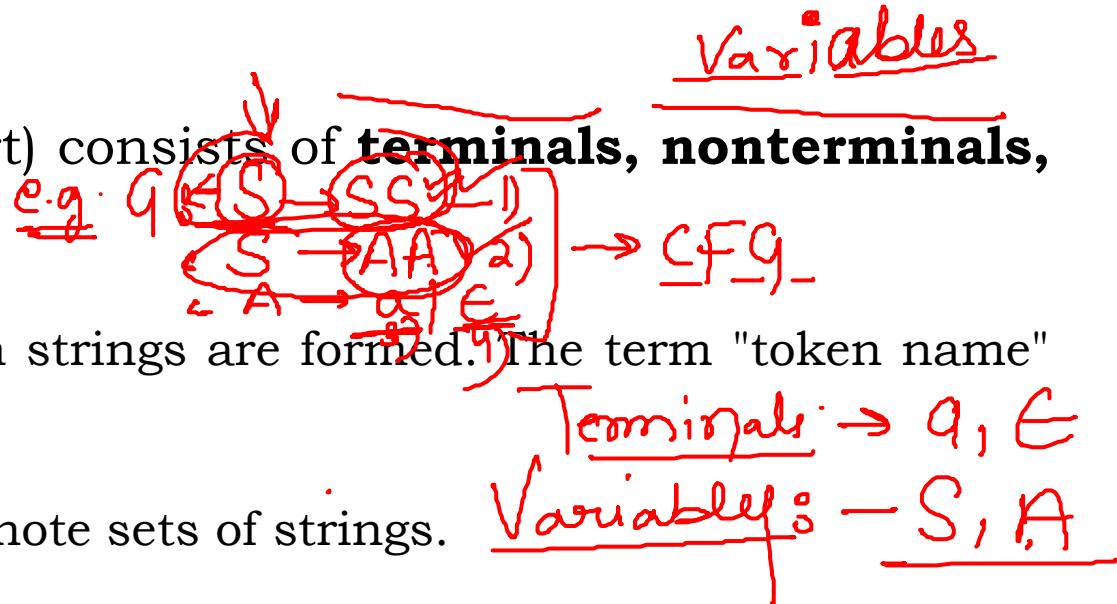
- “Given the CFG, construct a string that is in the language”

A CFG can also be used to recognize strings in its language

- “Given a string, decide whether it is in the language”

Formal Definition of a CFG

A context-free grammar (grammar for short) consists of **terminals**, **nonterminals**, **a start symbol**, and **productions**.



- **Terminals** are the basic symbols from which strings are formed. The term "token name" is a synonym for "terminal".
- **Nonterminals** are syntactic variables that denote sets of strings.
- In a grammar, one nonterminal is distinguished as the **start symbol**, and the set of strings it denotes is the language generated by the grammar.
- The **productions** of a grammar specify the manner in which the terminals and nonterminals can be combined to form strings.
 - Each production consists of: (a) A nonterminal called the head or left side of the production.
 - A body or right side consisting of zero or more terminals and nonterminals.

PARSING

It is the process of analyzing a continuous stream of input in order to determine its grammatical structure with respect to a given formal grammar.

Parse tree:

$$\rightarrow \quad \text{I/P: } a^* a^* a^* a^* \\ \text{G: } X \rightarrow X + X / X * X / a / \epsilon$$

Graphical representation of a derivation or deduction is called a parse tree. Each interior node of the parse tree is a non-terminal; the children of the node can be terminals or non-terminals.

Types of parsing:

1. Top down parsing
2. Bottom up parsing

- Top-down parsing : A parser can start with the start symbol and try to transform it to the input string. Example : **LL Parsers**.
- Bottom-up parsing : A parser can start with input and attempt to rewrite it into the start symbol. Example : **LR Parsers**. ✓

- ✓ 1) Terminals :- ? a, *, +, ε
- 2) Variables :- ? X
- 3) Parse Tree (Left most Derivation)
- 4) Ambiguous / Unambiguous.

Top-down parsing can be viewed as the problem of constructing a parse tree for the input string, starting from the root and creating the nodes of the parse tree in preorder.

Equivalently, It can be viewed as an attempt to find a left-most derivation for an input string or an attempt to construct a parse tree for the input starting from the root to the leaves.

Types of Top-Down Parsing

1. **Recursive descent parsing**
2. **Predictive parsing**

A general form of top-down parsing, called recursive descent parsing, which may require backtracking to find the correct production to be applied.

Typically, top-down parsers are implemented as a set of recursive functions that descend through a parse tree for a string. This approach is known as recursive descent parsing, also known as LL(k) parsing where the first L stands for left-to-right, the second L stands for leftmost-derivation, and k indicates k-symbol lookahead.

This parsing method may involve backtracking.

Example for : Backtracking

Consider the grammar G :

$$S \rightarrow cAd$$

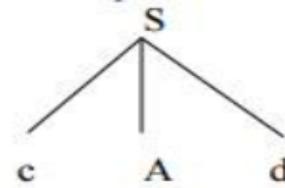
$$A \rightarrow ab \mid a$$

and the input string w=cad.

The parse tree can be constructed using the following top-down approach :

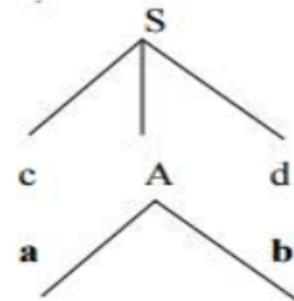
Step1:

Initially create a tree with single node labeled S. An input pointer points to 'c', the first symbol of w. Expand the tree with the production of S.



Step2:

The leftmost leaf 'c' matches the first symbol of w, so advance the input pointer to the second symbol of w 'a' and consider the next leaf 'A'. Expand A using the first alternative.

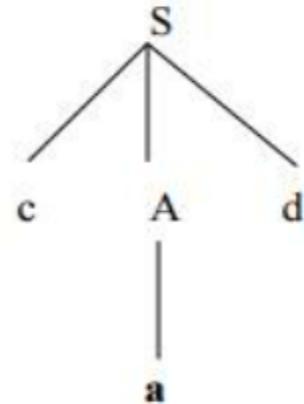


Step3:

The second symbol ‘a’ of w also matches with second leaf of tree. So advance the input pointer to third symbol of w ‘d’.But the third leaf of tree is b which does not match with the input symbol **d**.Hence discard the chosen production and reset the pointer to second **backtracking**.

Step4:

Now try the second alternative for A.

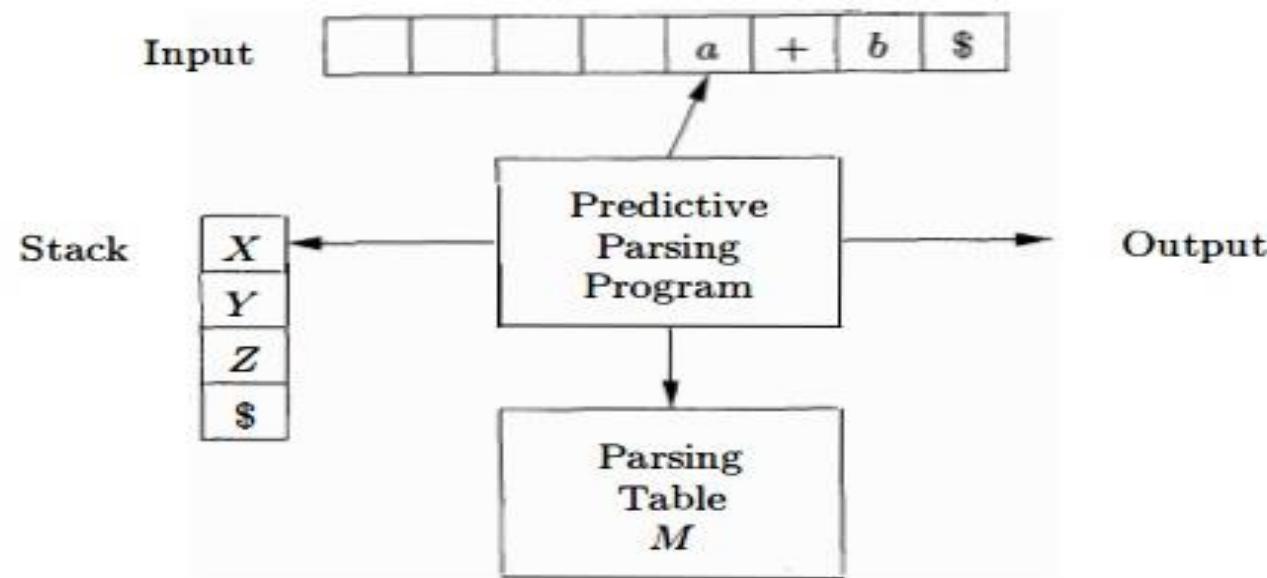


Now we can halt and announce the successful completion of parsing.

- It is possible to build a **nonrecursive predictive parser** by maintaining a stack explicitly, rather than implicitly via recursive calls. (A parser using the single-symbol look-ahead method and top-down parsing **without backtracking** is called **LL(1) Parser** or **Non-Recursive Parser**)
- The key problem during predictive parsing is that of determining the production to be applied for a nonterminal.
- The nonrecursive parser looks up the production to be applied in parsing table.

Non Recursive Predictive Parser

- A table-driven predictive parser has an **input buffer**, a **stack**, a **parsing table**, and an **output stream**.
- The **input buffer** contains the string to be parsed, followed by **\$**, a symbol used as a right endmarker to indicate the end of the input string.
- The **stack** contains a sequence of grammar symbols with **\$** on the bottom, indicating the **bottom of the stack**. Initially, the stack contains the start symbol of the grammar on top of **\$**.
- The **parsing table** is a **two dimensional array $M[A,a]$** where **A** is a nonterminal, and **a** is a terminal or the symbol **\$**.



The parser is controlled by a program that behaves as follows.

- The program considers X, the symbol on the top of the stack, and a, the current input symbol. These two symbols determine the action of the parser.
- There are three possibilities.
 1. If **X = a = \$**, the parser halts and announces successful completion of parsing.
 2. If **X = a ≠ \$**, the parser pops X off the stack and advances the input pointer to the next input symbol.
 3. If **X is a nonterminal**, the program consults entry **M[X,a]** of the parsing table M. This entry will be either an X-production of the grammar or an error entry. If, for example, **M[X,a] = {X→UVW}**, the parser replaces X on top of the stack by WVU (with U on top). If **M[X,a]=error**, the parser calls an error recovery routine.

RECURSIVE DESCENT PARSING

A type of top-down parsing built from a set of mutually recursive procedures where each procedure implements one of the non-terminals of the grammar

May or may not require backtracking

Uses procedures for every terminal and non-terminal entity

PREDICTIVE PARSING

A type of top-down parsing approach, which is also a type of recursive descent parsing, that does not involve any backtracking

Does not require any backtracking

Finds out the production to use by replacing the input string

- **Before Table Construction**

- Eliminate Left Recursion
- Preform Left Factoring

- **For Table Construction**

- FIRST
- FOLLOW

- **Table Construction**

Elimination of Left Recursion

Left recursion is eliminated by converting the grammar into a right recursive grammar.

If we have the left-recursive pair of productions-

$$A \rightarrow A\alpha / \beta$$

(Left Recursive Grammar)

where β does not begin with an A .

Then, we can eliminate left recursion by replacing the pair of productions with-

$$A \rightarrow \beta A'$$

$$A' \rightarrow \alpha A' / \epsilon$$

(Right Recursive Grammar)

For Example,
 $E \rightarrow E + T \mid T$

is replaced by

$$\begin{aligned}E &\rightarrow TE' \\E' &\rightarrow +TE' \mid \epsilon\end{aligned}$$

Left Recursion

Consider the following grammar and eliminate left recursion-

$$A \rightarrow ABd / Aa / a$$

$$B \rightarrow Be / b$$

Q 2. Consider the following grammar and eliminate left recursion-

$$E \rightarrow E + E / E \times E / a$$

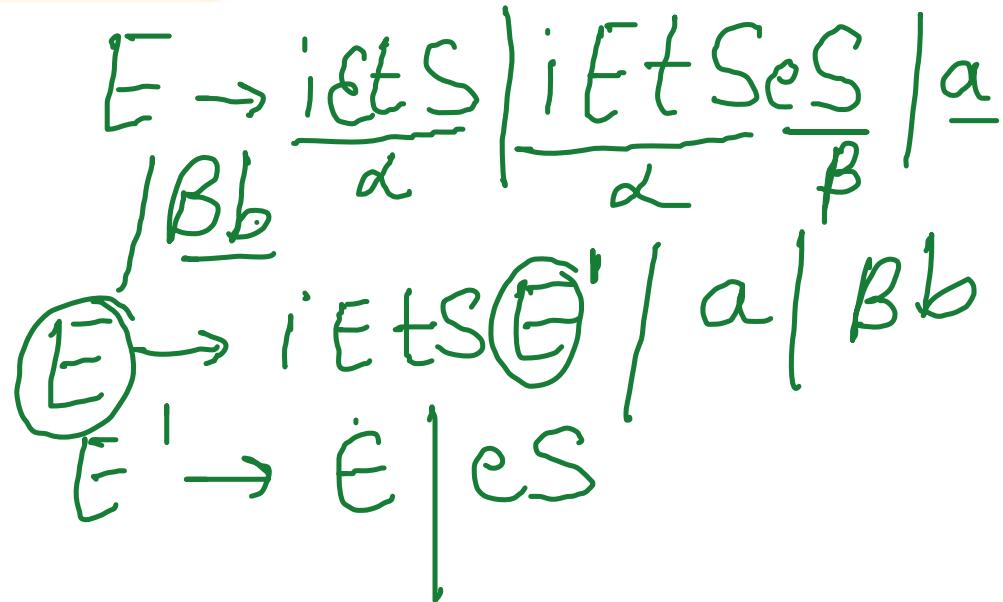
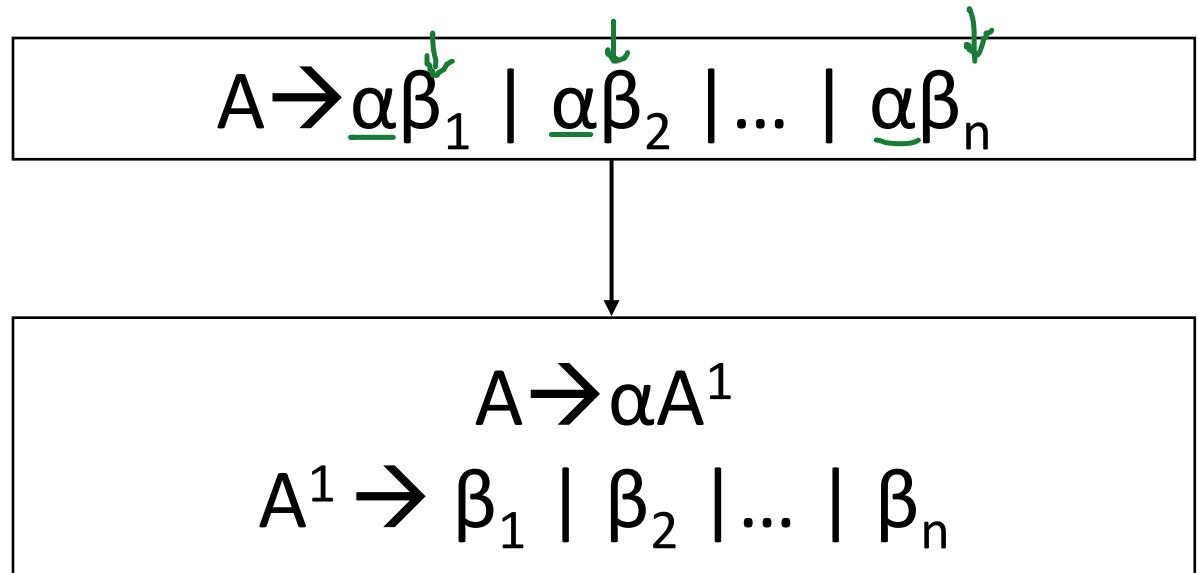
Q 3. Consider the following grammar and eliminate left recursion-

$$E \rightarrow E + T / T$$

$$T \rightarrow T \times F / F$$

$$F \rightarrow id$$

Left Factoring



For Example,

$E \rightarrow iEtS \mid iEtSeS \mid a \mid Bb$

is replaced by

$E \rightarrow iEtSE' \mid a \mid Bb$

$E' \rightarrow \epsilon \mid eS$

Left Factoring

Q1 .Do left factoring in the following grammar-

- $S \rightarrow iEtS / iEtSeS / a$
- $E \rightarrow b$

Q 2. Do left factoring in the following grammar-

- $A \rightarrow aAB / aBc / aAc$

Q 3. Do left factoring in the following grammar-

- $S \rightarrow bSSaaS / bSSaSb / bSb / a$

Rules for FIRST Sets

1. If X is a terminal **then** $\text{First}(X)$ is just $\{ X \}$
2. If there is a Production $X \rightarrow \varepsilon$ **then** add ε to $\text{first}(X)$
3. If there is a Production $X \rightarrow Y_1Y_2..Y_k$ **then** add $\text{first}(Y_1Y_2..Y_k)$ to $\text{first}(X)$

Example - First

Consider the production rules-

$$A \rightarrow abc / def / ghi$$

Then, we have-

$$\text{First}(A) = \{ a, d, g \}$$

$$2) S \rightarrow aBDh$$

$$B \rightarrow cC$$

$$C \rightarrow bC / \epsilon$$

$$D \rightarrow EF$$

$$E \rightarrow g / \epsilon$$

$$F \rightarrow f / \epsilon$$

First Functions-

$$\text{First}(S) = \{ a \}$$

$$\text{First}(B) = \{ c \}$$

$$\text{First}(C) = \{ b, \in \}$$

$$\text{First}(D) = \{ \text{First}(E) - \in \} \cup \text{First}(F) = \{ g, f, \in \}$$

$$\text{First}(E) = \{ g, \in \}$$

$$\text{First}(F) = \{ f, \in \}$$

Rules for Follow Sets

1. First put \$ (the end of input marker) in $\text{Follow}(S)$ (S is the start symbol)
2. If there is a production $A \rightarrow aB\beta$ **then** everything in $\text{FIRST}(\beta)$ except for ϵ is placed in $\text{FOLLOW}(B)$
3. If there is a production $A \rightarrow aB$ **then** everything in $\text{FOLLOW}(A)$ is in $\text{FOLLOW}(B)$
 - If there is a production $A \rightarrow aB\beta$ where $\text{FIRST}(\beta)$ contains ϵ , **then** everything in $\text{FOLLOW}(A)$ is in $\text{FOLLOW}(B)$

Example

$S \rightarrow aBDh$

$B \rightarrow cC$

$C \rightarrow bC / \epsilon$

$D \rightarrow EF$

$E \rightarrow g / \epsilon$

$F \rightarrow f / \epsilon$

- **Follow Functions-**

- $\text{Follow}(S) = \{ \$ \}$
- $\text{Follow}(B) = \{ \text{First}(D) - \epsilon \} \cup \text{First}(h) = \{ g, f, h \}$
- $\text{Follow}(C) = \text{Follow}(B) = \{ g, f, h \}$
- $\text{Follow}(D) = \text{First}(h) = \{ h \}$
- $\text{Follow}(E) = \{ \text{First}(F) - \epsilon \} \cup \text{Follow}(D) = \{ f, h \}$
- $\text{Follow}(F) = \text{Follow}(D) = \{ h \}$

Practice Questions on First and Follow

1) Calculate the first and follow functions for the given grammar-

$$S \rightarrow A$$

$$A \rightarrow aB / Ad$$

$$B \rightarrow b$$

$$C \rightarrow g$$

2) Calculate the first and follow functions for the given grammar-

$$S \rightarrow (L) / a$$

$$L \rightarrow SL'$$

$$L' \rightarrow ,SL' / \epsilon$$

INPUT : Grammar G

OUTPUT: Parsing table M

For each production $A \rightarrow a$, do the following :

1. For each terminal 'a' in $\text{FIRST}(A)$, add $A \rightarrow a$ to $M[A,a]$.
2. If ϵ is in $\text{FIRST}(a)$ then for each terminal b in $\text{FOLLOW}(A)$.
 $A \rightarrow a$ to $M[A,b]$. If b is $\$$ then also add $A \rightarrow a$ to $M[A,\$]$.
3. If there is no production in $M[A,a]$, then set $M[A,a]$ to error.

Construct Predictive Parsing Table for the following CFG

$$S \rightarrow cAd$$

$$A \rightarrow ab \mid a$$

Step 1: Eliminate the Left Recursion and Perform Left Factoring if present

- There is no Left Recursion in the given CFG.
- Left Factoring present in the given CFG (In the Non-Terminal A).
Left Factoring can be performed by the following

$A \rightarrow ab \mid a$ can be replaced by

$$\begin{array}{l} A \rightarrow aA' \\ A' \rightarrow b \mid \epsilon \end{array}$$

CFG after Left Factoring:

$$S \rightarrow cAd$$

$$A \rightarrow aA'$$

$$A' \rightarrow b \mid \epsilon$$

Step 2: Find FIRST set for all the non-terminals

$$\text{FIRST}(S) = \{ c \} \quad - \text{Rule (1)}$$

$$\text{FIRST}(A) = \{ a \} \quad - \text{Rule (1)}$$

$$\text{FIRST}(A') = \{ b, \epsilon \} \quad - \text{Rule (1) \& (2)}$$

Step 3: Find FOLLOW set for all the non-terminals

FOLLOW(S) = { \$ }

- Rule (1)

FOLLOW(A) = { d }

- Rule (2)

FOLLOW(A') = { d }

- Rule (3)

Step 4: Construct Predictive Parsing Table

NT/T	a	b	c	d	\$
S			S → cAd		
A	A → aA'				
A'		A' → b		A' → ε	

S → cAd
A → aA'
A' → b | ε

Construct Predictive Parsing Table for the following CFG

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * F \mid F$$

$$F \rightarrow (E) \mid id$$

Step 1: Eliminate the Left Recursion and Perform Left Factoring if present

- **Left Recursion present in the given CFG. (In the Non-Terminals E and T)**

$E \rightarrow E + T \mid T$ can be replaced by

$$\begin{aligned} E &\rightarrow TE' \\ E' &\rightarrow +TE' \mid \epsilon \end{aligned}$$

$T \rightarrow T * F \mid F$ can be replaced by

$$\begin{aligned} T &\rightarrow FT' \\ T' &\rightarrow +FT' \mid \epsilon \end{aligned}$$

CFG after Left Recursion:

```
E → TE'  
E' → +TE' | ε  
T → FT'  
T' → +FT' | ε  
F → (E) | id
```

No Left Factoring present in the given CFG.

Step 2: Find FIRST set for all the non-terminals

- | | |
|-----------------------------|---------------|
| FIRST(E) = { (, id } | - Rule (3) |
| FIRST(E') = { +, ε } | - Rule (1, 2) |
| FIRST(T) = { (, id } | - Rule (3) |
| FIRST(T') = { *, ε } | - Rule (1, 2) |
| FIRST(F) = { (, id } | - Rule (3) |

Table Construction

Step 3: Find FOLLOW set for all the non-terminals

$$\text{FOLLOW}(E) = \{ \$,) \}$$

- Rule (1,2)

$$\text{FOLLOW}(E') = \{ \$,) \}$$

- Rule (3)

$$\text{FOLLOW}(T) = \{ +, \$,) \}$$

- Rule (2,3)

$$\text{FOLLOW}(T') = \{ +, \$,) \}$$

- Rule (3)

$$\text{FOLLOW}(F) = \{ *, +, \$,) \}$$

- Rule (2,3)

$E \rightarrow TE'$
$E' \rightarrow +TE' \mid \epsilon$
$T \rightarrow FT'$
$T' \rightarrow +FT' \mid \epsilon$
$F \rightarrow (E) \mid id$

Step 4: Construct Predictive Parsing Table

NT/T	id	+	*	()	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$		

Input Validation by Predictive Parser

Consider the input String **id+id*id\$**

NT/T	id	+	*	()	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$		

Stack	Input	Output
\$E	id+id*id\$	
\$E'T	id+id*id\$	$E \rightarrow TE'$
\$E'T'F	id+id*id\$	$T \rightarrow FT'$
\$E'T'id	id+id*id\$	$F \rightarrow id$
\$E'T'	+id*id\$	
\$E'	+id*id\$	$T' \rightarrow \epsilon$
\$E'T+	+id*id\$	$E' \rightarrow +TE'$
\$E'T	id*id\$	
\$E'T'F	id*id\$	$T \rightarrow FT'$
\$E'T'id	id*id\$	$F \rightarrow id$
\$E'T'	*id\$	
\$E'T'F*	*id\$	$T' \rightarrow *FT'$
\$E'T'F	id\$	
\$E'T'id	id\$	$F \rightarrow id$
\$E'T'	\$	
\$E'	\$	$T' \rightarrow \epsilon$
\$	\$	$E' \rightarrow \epsilon$ 15

Predictive parsers, that is, recursive-descent parsers needing no backtracking, can be constructed for a class of grammars called LL(1).

A Grammar whose parsing table has no multiply-defined entries is said to be LL(1).

LL(1) grammar have several distinctive properties.

- No ambiguous or left recursive grammar can be LL(1).
- It can be also be shown that a grammar G is LL(1) if and only if whenever $A \rightarrow \alpha \mid \beta$ are two distinct productions of G the following conditions hold:
 1. For no terminal a do both α and β derive strings beginning with a .
 2. At most one of α and β can derive the empty string.
 3. If $\beta \rightarrow \epsilon$, then α does not derive any string beginning with a terminal in $\text{FOLLOW}(A)$.

Construct Predictive parser table and perform parsing

$S \rightarrow (L) / a$

$L \rightarrow SL'$

$L' \rightarrow ,SL' / \in$

- Input : (a, a)

Consider this following grammar:

$$\begin{aligned} S &\rightarrow iEtS \mid iEtSeS \mid a \\ E &\rightarrow b \end{aligned}$$

After eliminating left factoring, we have

$$\begin{aligned} S &\rightarrow iEtSS' \mid a \\ S' &\rightarrow eS \mid \epsilon \\ E &\rightarrow b \end{aligned}$$

To construct a parsing table, we need FIRST() and FOLLOW() for all the non-terminals.

$$\text{FIRST}(S) = \{ i, a \}$$

$$\text{FIRST}(S') = \{ e, \epsilon \}$$

$$\text{FIRST}(E) = \{ b \}$$

$$\text{FOLLOW}(S) = \{ \$, e \}$$

$$\text{FOLLOW}(S') = \{ \$, e \}$$

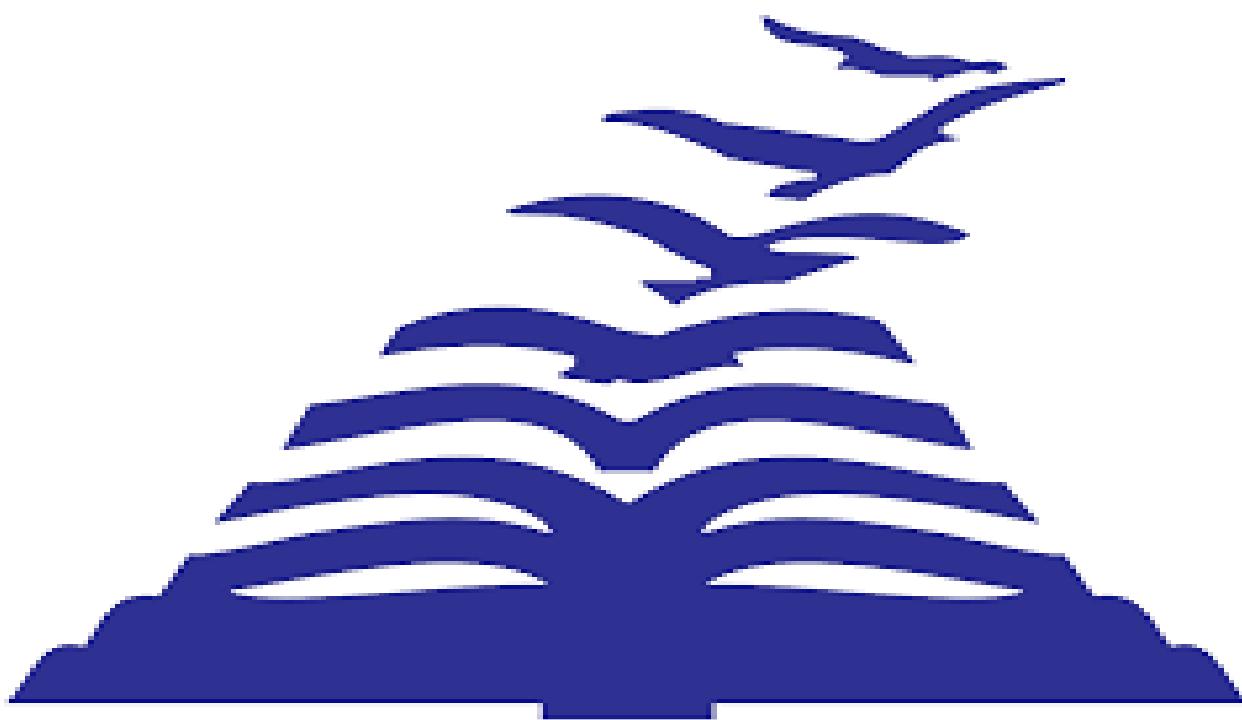
$$\text{FOLLOW}(E) = \{ t \}$$

Step 4: Construct Predictive Parsing Table

$$\begin{aligned} S &\rightarrow iEtSS' \mid a \\ S' &\rightarrow eS \mid \epsilon \\ E &\rightarrow b \end{aligned}$$

NT/T	i	t	a	e	b	\$
S	$S \rightarrow iEtSS'$		$S \rightarrow a$			
S'				$S \rightarrow eS$ $S' \rightarrow \epsilon$		$S' \rightarrow \epsilon$
E					$E \rightarrow b$	

M[S', e] contains two productions in a single entry ($S' \rightarrow eS$ and $S' \rightarrow \epsilon$). Since there are more than one production, the grammar is **not LL(1) grammar**.



Presidency University, Bengaluru

Bottom-Up Parsing

- Constructing a parse tree for an input string beginning at the leaves and going towards the root is called bottom-up parsing.
- A general type of bottom-up parser is a shift-reduce parser.
- A much more general method of shift-reduce parsing, called as LR parsing.

SHIFT-REDUCE PARSING

- Constructing a parse tree for an input string beginning at the leaves (bottom) and going towards the root (top) is called shift-reduce parsing.

Reduction:

- We can think of bottom-up parsing as the process of "reducing" a string w to the start symbol of the grammar. At each reduction step, a specific substring matching the body of a production is replaced by the nonterminal at the head of that production.

Shift-reduce Parsing

Example:

Consider the grammar:

S → **aABe**
A → **Abc** | **b**
B → **d**

The sentence to be recognized is **abbcde**.

abbcde	(A → b)
aAbcde	(A → Abc)
aAde	(B → d)
aABe	(S → aABe)
S	

The reductions trace out the right-most derivation in reverse.

Handles:

A handle of a string is a **substring that matches the right side of a production**, and whose reduction to the non-terminal on the left side of the production represents **one step along the reverse of a rightmost derivation**.

Handle pruning:

A rightmost derivation in reverse can be obtained by “handle pruning”.

Shift-reduce Parsing

Given CFG

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * F \mid F$$

$$F \rightarrow (E) \mid id$$

Input String : **id + id * id**

Rightmost Derivation of *id₁ + id₂ * id₃*

$$\begin{aligned} E &\rightarrow E + \underline{T} \\ &\rightarrow E + T * \underline{F} \\ &\rightarrow E + \underline{T} * id_3 \\ &\rightarrow E + \underline{F} * id_3 \\ &\rightarrow E + id_2 * id_3 \\ &\rightarrow \underline{E} + id_2 * id_3 \\ &\rightarrow \underline{T} + id_2 * id_3 \\ &\rightarrow \underline{F} + id_2 * id_3 \\ &\rightarrow id_1 + id_2 * id_3 \end{aligned}$$

Handle pruning:

A rightmost derivation in reverse can be obtained by “handle pruning”.

Reduction of *id₁ + id₂ * id₃*

$$\begin{aligned} &\underline{id_1} + id_2 * id_3 \\ &\rightarrow \underline{F} + id_2 * id_3 \\ &\rightarrow \underline{T} + id_2 * id_3 \\ &\rightarrow E + \underline{id_2} * id_3 \\ &\rightarrow E + \underline{F} * id_3 \\ &\rightarrow E + T * \underline{id_3} \\ &\rightarrow E + \underline{T * F} \\ &\rightarrow \underline{E + T} \\ &\rightarrow E \end{aligned}$$

Actions in shift-reduce parser:

- | | |
|---------------|--|
| shift | - The next input symbol is shifted onto the top of the stack. |
| reduce | - The parser replaces the handle within a stack with a non-terminal. |
| accept | - The parser announces successful completion of parsing. |
| error | - The parser discovers that a syntax error has occurred and calls an error recovery routine. |

Conflicts in shift-reduce parsing:

There are two conflicts that occur in shift-reduce parsing:

1. **Shift-reduce conflict:** The parser cannot decide whether to shift or to reduce.

2. **Reduce-reduce conflict:** The parser cannot decide which of several reductions to make.

Stack Implementation of Shift Reduce Parsing

Initial Configuration

Stack	Input Buffer
\$	w\$

After N number of Handle Pruning (Rightmost Derivation in Reverse)

Final Configuration

Stack	Input Buffer
\$S	\$

Where \$ used to mark the bottom of the Stack and also right end of the Input String.

Viable prefixes:

The set of prefixes of right sentinel forms that can appear on the stack of a shift-reduce parser are called viable prefixes

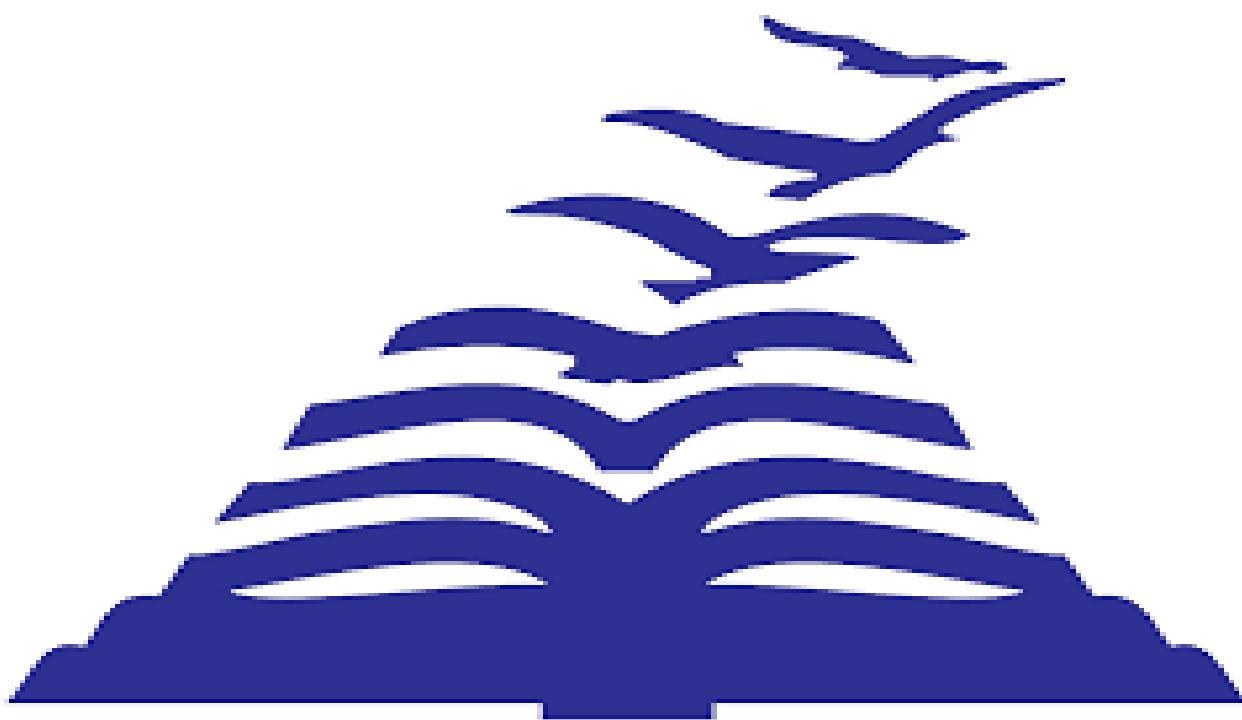
Shift-reduce Parsing

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow (E) \mid id \end{aligned}$$

STACK	INPUT	ACTION
\$	id ₁ * id ₂ \$	shift
\$ id ₁	* id ₂ \$	reduce by $F \rightarrow id$
\$ F	* id ₂ \$	reduce by $T \rightarrow F$
\$ T	* id ₂ \$	shift
\$ T *	id ₂ \$	shift
\$ T * id ₂	\$	reduce by $F \rightarrow id$
\$ T * F	\$	reduce by $T \rightarrow T * F$
\$ T	\$	reduce by $E \rightarrow T$
\$ E	\$	accept

$$\begin{aligned} E &\rightarrow E+E \\ E &\rightarrow E*E \\ E &\rightarrow (E) \\ E &\rightarrow id \end{aligned}$$

Stack	Input	Action
\$	id ₁ + id ₂ * id ₃ \$	shift
Si d ₁	+ id ₂ * id ₃ \$	reduce by $E \rightarrow id$
SE	+ id ₂ * id ₃ \$	shift
SE +	id ₂ * id ₃ \$	shift
SE + id ₂	* id ₃ \$	reduce by $E \rightarrow id$
SE + E	* id ₃ \$	shift
SE + E *	id ₃ \$	shift
SE + E * id ₃	\$	reduce by $E \rightarrow id$
SE + E * E	\$	reduce by $E \rightarrow E * E$
SE + E	\$	reduce by $E \rightarrow E + E$
SE	\$	accept



Presidency University, Bengaluru

LR Parser

- Bottom-up parsing : A parser can start with input and attempt to rewrite it into the start symbol. Example : **LR Parsers**.

An **efficient bottom-up syntax analysis technique** that can be used CFG is called **LR(k) parsing**. The '**L**' is for **left-to-right scanning** of the input, the '**R**' for **constructing a rightmost derivation in reverse**, and the ' k ' for the number of input symbols.

We shall consider the cases where **$k = 0$ and $k = 1$** . An **LR(0) parser does not use look-ahead to decide its shift or reduce actions**.

Advantages of LR parsing:

1. It recognizes virtually **all programming language constructs** for which CFG can be written.
2. It is an **efficient non-backtracking shift-reduce parsing method**.
3. A grammar that can be parsed using LR method is a **proper superset of a grammar** that can be parsed with predictive parser
4. It detects a **syntactic error as soon as possible**.

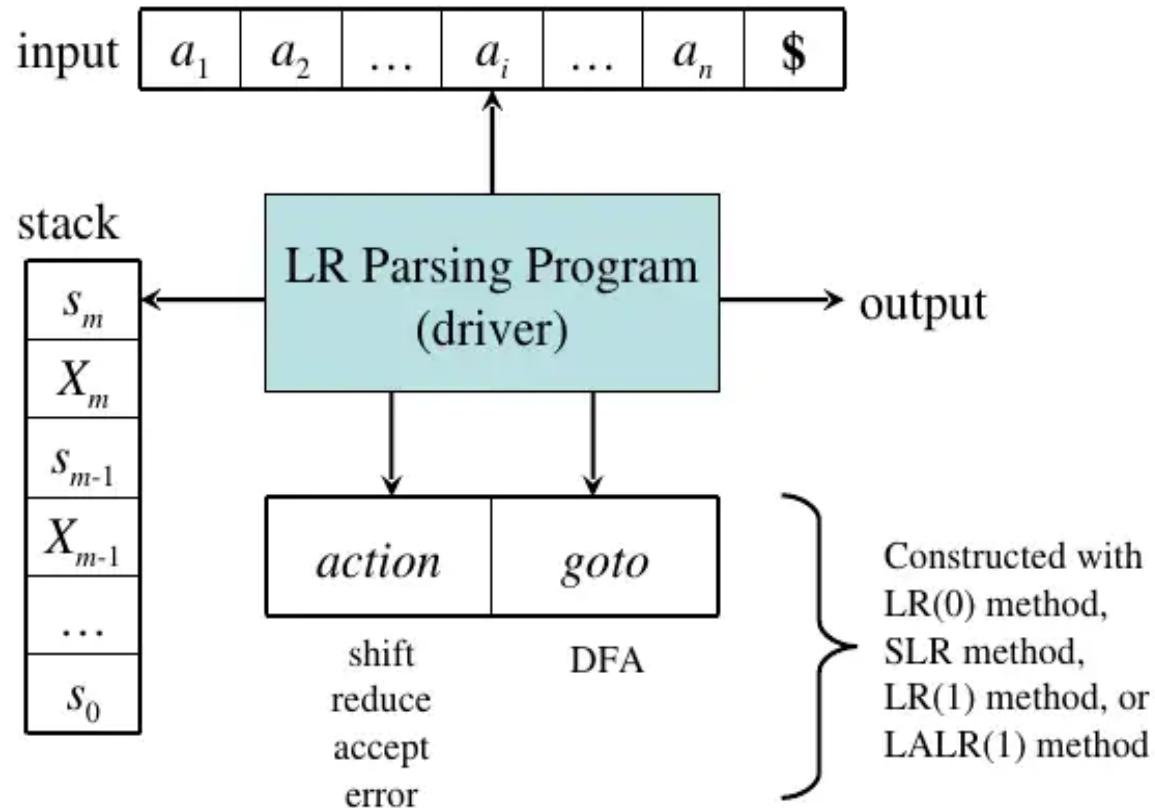
Drawbacks of LR method:

- It is **too much of work to construct** a LR parser.
- A **specialized tool**, an LR parser generator, is needed (**Eg. YACC**)

Types of LR parsing method:

1. **SLR- Simple LR**
Easiest to implement, least powerful.
2. **CLR- Canonical LR**
Most powerful, most expensive.
3. **LALR- Look-Ahead LR**
Intermediate in size and cost between the other two methods.

Model of an LR Parser



The LR parsing algorithm:

It consists of an input, an output, a stack, a driver program, and a pa parts (action and goto).

- The **driver program is the same for all LR parser**.
- The parsing program reads characters from an input buffer one at a time.
- The program uses a stack to store a string of the form $s_0X_1s_1X_2s_2\dots X_ms_m$, where s_m is on top. Each X_i is a **grammar symbol** and each s_i is a **state**.
- The parsing table consists of two parts: **action** and **goto** functions.

LR Parsing algorithm:

Input: An input string w and an LR parsing table with functions action and goto for grammar G. **Output:** If w is in L(G), a bottom-up-parse for w; otherwise, an error indication.

Method: Initially, the parser has s_0 on its stack, where s_0 is the initial state, and $w\$$ in the input buffer. The parser then executes the following program:

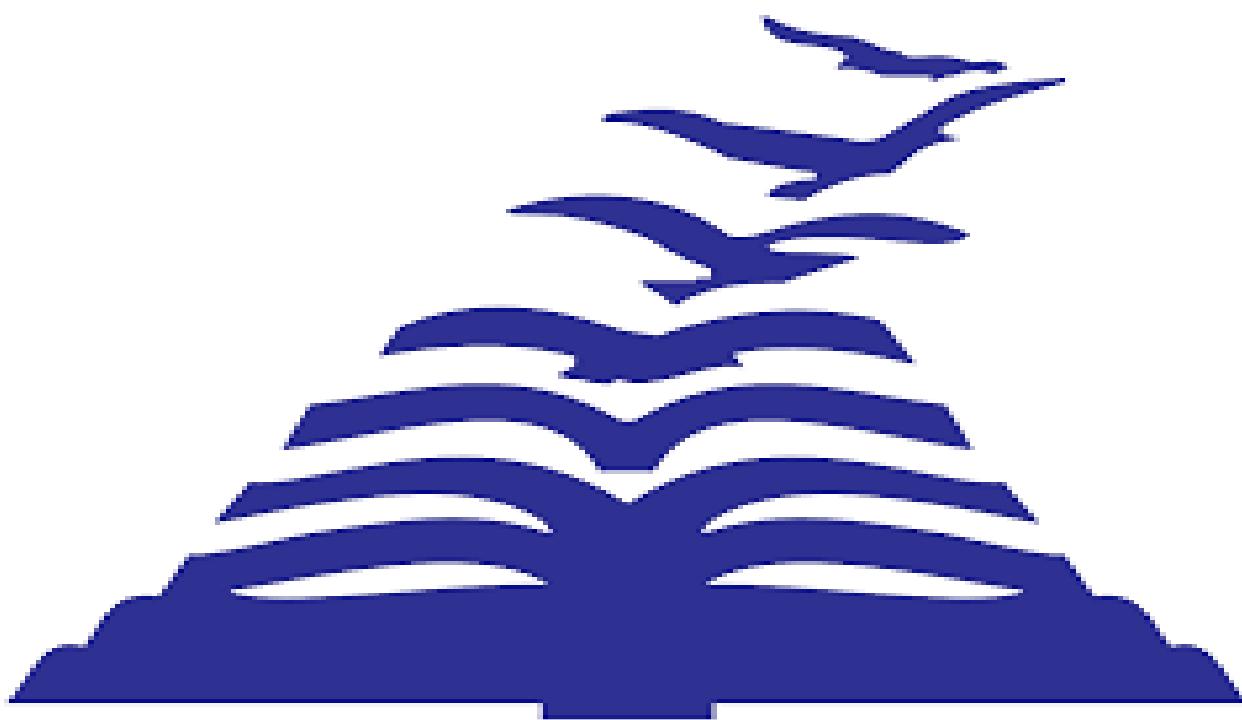
```
set ip to point to the first input symbol of w$; repeat forever begin
    let s be the state on top of the stack and a the symbol pointed to by ip;

    if action[s, a] = shift s' then begin
        push a then s' on top of the stack; advance ip to the next input symbol end

    else if action[s, a] = reduce A→β then begin
        pop 2* | β | symbols off the stack;
        let s' be the state now on top of the stack;
        push A then goto[s', A] on top of the stack;
        output the production A→ β
    end

    else if action[s, a] = accept then
        return

    else error()
end
```



Presidency University, Bengaluru

SLR Parser

To perform SLR parsing, take grammar as input and do the following:

1. **Find LR(0) items.**
2. **Completing the closure.**
3. **Compute goto(I,X), where, I is set of items and X is grammar symbol.**

LR(0) items:

An LR(0) item of a grammar G is a production of G with a dot at some position of the right side.

For example, production $A \rightarrow XYZ$ yields the four items :

A → .XYZ

A → X .YZ

A → XY .Z

A → XYZ .

Closure operation:

If I is a set of items for a grammar G , then $\text{closure}(I)$ is the set of items constructed from I by the two rules:

- Initially, every item in I is added to $\text{closure}(I)$.
- If $A \rightarrow a . B\beta$ is in $\text{closure}(I)$ and $B \rightarrow \gamma$ is a production, then add the item $B \rightarrow . \gamma$ to I , if it is not already there. We apply this rule until no more new items can be added to $\text{closure}(I)$.

Goto operation:

$\text{Goto}(I, X)$ is defined to be the closure of the set of all items $[A \rightarrow aX . \beta]$ such that $[A \rightarrow a . X\beta]$ is in I .

Steps to construct SLR parsing table for grammar G are:

- Augment G and produce G'
- Construct the canonical collection of set of items C for G'
- Construct the parsing action function action and goto using the following algorithm that requires $\text{FOLLOW}(A)$ for each non-terminal of grammar.

Algorithm for construction of SLR parsing table:

Input : An augmented grammar G'

Output : The SLR parsing table functions action and goto for G'

Method :

1. Construct $C = \{I_0, I_1, \dots, I_n\}$, the collection of sets of LR(0) items for G' .
2. State i is constructed from I_i . The parsing functions for state i are determined as follows:
 - (a) If $[A \rightarrow a \cdot a\beta]$ is in I_i and $\text{goto}(I_i, a) = I_j$, then set $\text{action}[i, a]$ to “shift j ”. Here a must be terminal.
 - (b) If $[A \rightarrow a \cdot]$ is in I_i , then set $\text{action}[i, a]$ to “reduce $A \rightarrow a$ ” for all a in $\text{FOLLOW}(A)$.
 - (c) If $[S' \rightarrow S.]$ is in I_i , then set $\text{action}[i, \$]$ to “accept”.

If any conflicting actions are generated by the above rules, we say grammar is not SLR(1).

3. The goto transitions for state i are constructed for all non-term
If $\text{goto}(I_i, A) = I_j$, then $\text{goto}[i, A] = j$.
4. All entries not defined by rules (2) and (3) are made “error”.
5. The initial state of the parser is the one constructed from the $[S' \rightarrow .S]$.

Constructing SLR Parsing Table



Grammar:

1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow id$

State	Action							Goto		
	id	+	*	()	\$	E	T	F	
0	S5			S4			1	2	3	
1			S6			accept				
2		R2	S7		R2	R2				
3		R4	R4		R4	R4				
4	S5			S4			8	2	3	
5		R6	R6		R6	R6				
6	S5			S4				9	3	
7	S5			S4					10	
8		S6			S11					
9		R1	S7		R1	R1				
10		R3	R3		R3	R3				
11		R5	R5		R5	R5				

Constructing SLR Parsing Table



	STACK	SYMBOLS	INPUT	ACTION
(1)	0		id * id + id \$	shift
(2)	0 5	id	* id + id \$	reduce by $F \rightarrow id$
(3)	0 3	F	* id + id \$	reduce by $T \rightarrow F$
(4)	0 2	T	* id + id \$	shift
(5)	0 2 7	T *	id + id \$	shift
(6)	0 2 7 5	T * id	+ id \$	reduce by $F \rightarrow id$
(7)	0 2 7 10	T * F	+ id \$	reduce by $T \rightarrow T * F$
(8)	0 2	T	+ id \$	reduce by $E \rightarrow T$
(9)	0 1	E	+ id \$	shift
(10)	0 1 6	E +	id \$	shift
(11)	0 1 6 5	E + id	\$	reduce by $F \rightarrow id$
(12)	0 1 6 3	E + F	\$	reduce by $T \rightarrow F$
(13)	0 1 6 9	E + T	\$	reduce by $E \rightarrow E + T$
(14)	0 1	E	\$	accept

Construct SLR Parsing Table for the following CFG

Grammar:

1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow id$

Construction of SLR Parsing Table

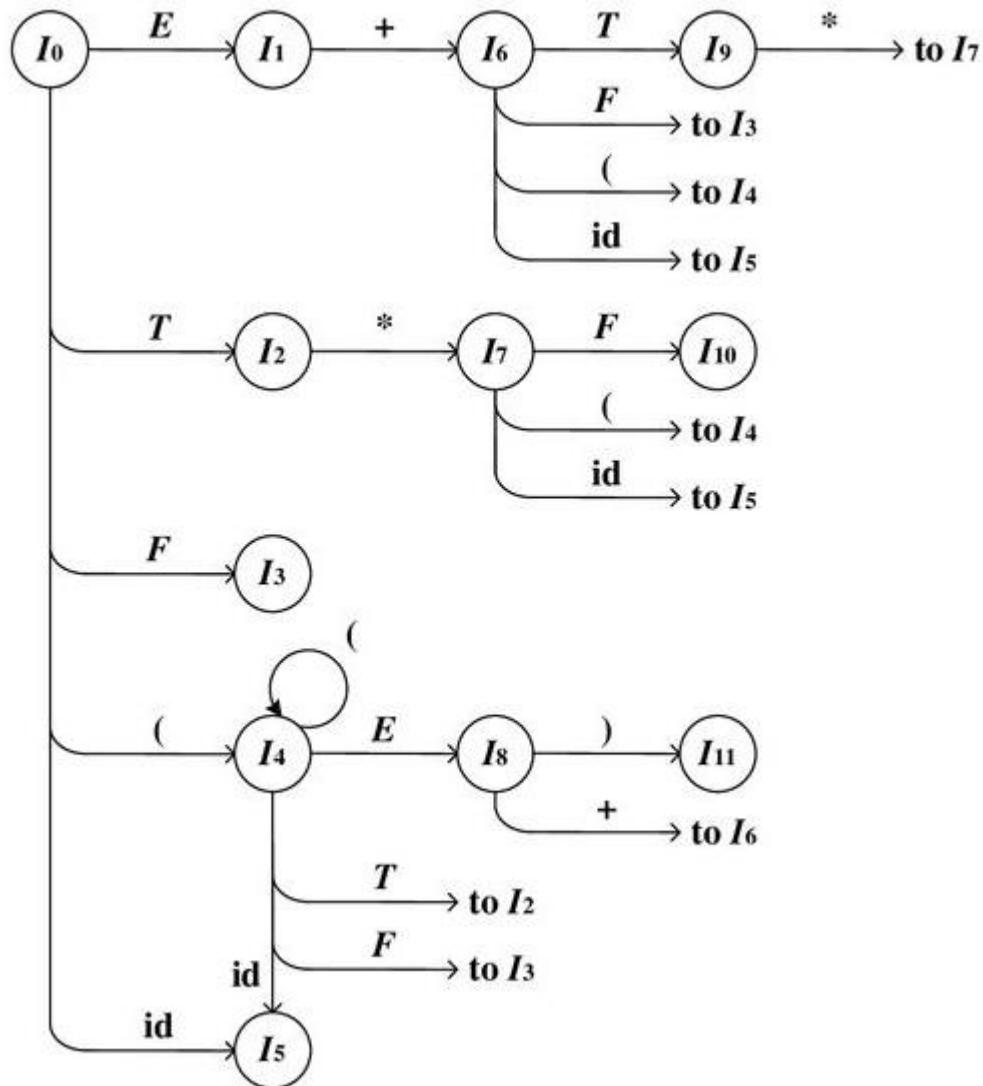
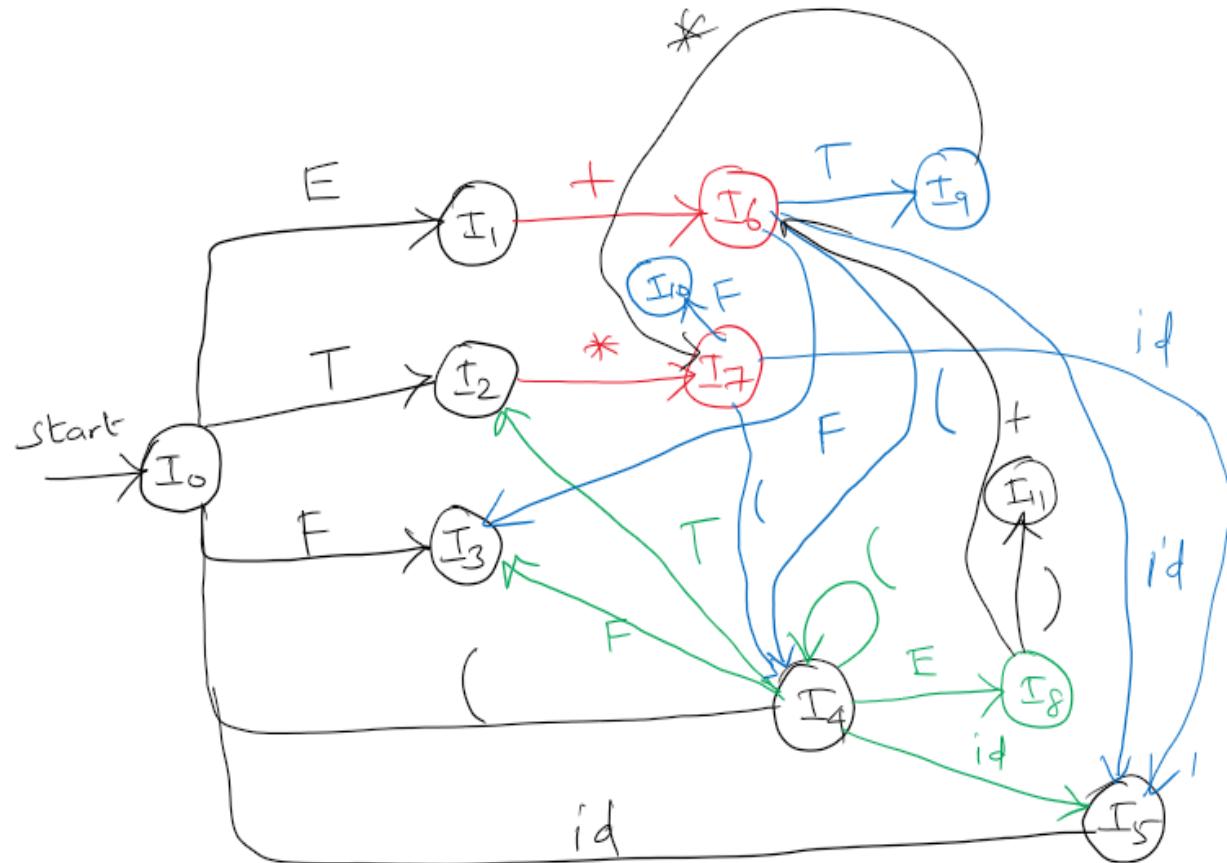
Step1: Construct Augmented Grammar

$E' \rightarrow E$
 $E \rightarrow E + T$
 $E \rightarrow T$
 $T \rightarrow T * F$
 $T \rightarrow F$
 $F \rightarrow (E)$
 $F \rightarrow id$

Step2: Find LR(0) Items, Perform closure operation & Compute Goto		I_0	I_3	I_7	I_9	I_{10}	I_7	I_8	I_4	I_5	I_1	I_6	I_9	I_{11}
		$E' \rightarrow \cdot E$ $E \rightarrow \cdot E + T$ $E \rightarrow \cdot T$ $T \rightarrow \cdot T * F$ $T \rightarrow \cdot F$ $F \rightarrow \cdot (E)$ $F \rightarrow \cdot id$	$goto(I_0, F)$ $T \rightarrow F.$	$goto(I_0, *)$ $T \rightarrow T * . F$	$goto(I_4, id)$ $F \rightarrow id. \Rightarrow I_5$	$goto(I_7, F)$ $T \rightarrow T * F.$	$goto(I_7, +)$ $E \rightarrow E + . T$	$goto(I_8, *)$ $T \rightarrow T * . F$	$goto(I_4, E)$ $F \rightarrow (\cdot E)$	$goto(I_7, ()$ $F \rightarrow (\cdot E)$	$goto(I_9, *)$ $T \rightarrow T * . F$	$E \rightarrow \cdot T \Rightarrow I_4$	$goto(I_9, +)$ $E \rightarrow E + . T$	$goto(I_7, id)$ $F \rightarrow id.$
		$E' \rightarrow E \cdot$ $E \rightarrow E \cdot + T$	$goto(I_0, E)$	$goto(I_0, id)$ $F \rightarrow id.$	$goto(I_4, T)$ $E \rightarrow T. \Rightarrow I_2$	$goto(I_7, ())$ $F \rightarrow (\cdot E)$	$goto(I_8, *)$ $F \rightarrow T * . F$	$goto(I_4, F)$ $T \rightarrow F. \Rightarrow I_3$	$goto(I_4, T)$ $E \rightarrow T. \Rightarrow I_2$	$goto(I_7, ())$ $F \rightarrow (\cdot E)$	$goto(I_9, *)$ $T \rightarrow T * . F$	$E \rightarrow \cdot T \Rightarrow I_4$	$goto(I_7, id)$ $F \rightarrow id.$	$goto(I_8, ())$ $F \rightarrow (\cdot E)$
		$E \rightarrow T \cdot$ $T \rightarrow T \cdot * F$	$goto(I_0, T)$	$goto(I_1, +)$ $E \rightarrow E + T$	$goto(I_4, F)$ $T \rightarrow F. \Rightarrow I_3$	$goto(I_7, F)$ $T \rightarrow T * . F$	$goto(I_4, ())$ $F \rightarrow (\cdot E)$	$goto(I_4, F)$ $T \rightarrow F. \Rightarrow I_3$	$goto(I_4, T)$ $E \rightarrow T. \Rightarrow I_2$	$goto(I_7, F)$ $T \rightarrow T * . F$	$goto(I_9, *)$ $T \rightarrow T * . F$	$E \rightarrow \cdot T \Rightarrow I_4$	$goto(I_7, id)$ $F \rightarrow id.$	$goto(I_8, ())$ $F \rightarrow (\cdot E)$
		$T \rightarrow F \cdot$ $F \rightarrow (E)$ $F \rightarrow id$	$goto(I_0, F)$	$goto(I_1, +)$ $E \rightarrow E + T$	$goto(I_4, F)$ $T \rightarrow F. \Rightarrow I_3$	$goto(I_7, F)$ $T \rightarrow T * . F$	$goto(I_4, ())$ $F \rightarrow (\cdot E)$	$goto(I_4, F)$ $T \rightarrow F. \Rightarrow I_3$	$goto(I_4, T)$ $E \rightarrow T. \Rightarrow I_2$	$goto(I_7, F)$ $T \rightarrow T * . F$	$goto(I_9, *)$ $T \rightarrow T * . F$	$E \rightarrow \cdot T \Rightarrow I_4$	$goto(I_7, id)$ $F \rightarrow id.$	$goto(I_8, ())$ $F \rightarrow (\cdot E)$

Constructing SLR(1) Parsing Table

Step 3: Draw Transition Diagram



Constructing SLR(1) Parsing Table



Step 4: Find the FOLLOW set for all the Non-Terminals to perform Reduction

$$\text{FOLLOW}(E) = \{ +,) \}$$

$$\text{FOLLOW}(T) = \{ +,), * \}$$

$$\text{FOLLOW}(F) = \{ +,), * \}$$

Grammar:

1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow \text{id}$

SLR Parsing Table:

State	Action							Goto		
	id	+	*	()	\$	E	T	F	
0	S5		S4				1	2	3	
1		S6				accept				
2		R2	S7			R2	R2			
3		R4	R4			R4	R4			
4	S5			S4			8	2	3	
5		R6	R6			R6	R6			
6	S5			S4				9	3	
7	S5			S4					10	
8		S6			S11					
9		R1	S7		R1	R1				
10		R3	R3		R3	R3				
11		R5	R5		R5	R5				

Construction of SLR Parsing Table

2. Construct the SLR Parsing Table for the following Grammar

$S \rightarrow (L)$

$S \rightarrow a$

$L \rightarrow L, S$

$L \rightarrow S$

3. Construct the SLR Parsing Table for the following Grammar

$S \rightarrow L=R$

$S \rightarrow R$

$L \rightarrow *R$

$L \rightarrow id$

$R \rightarrow L$

4. Construct the SLR Parsing Table for the following Grammar

$S \rightarrow aAd$

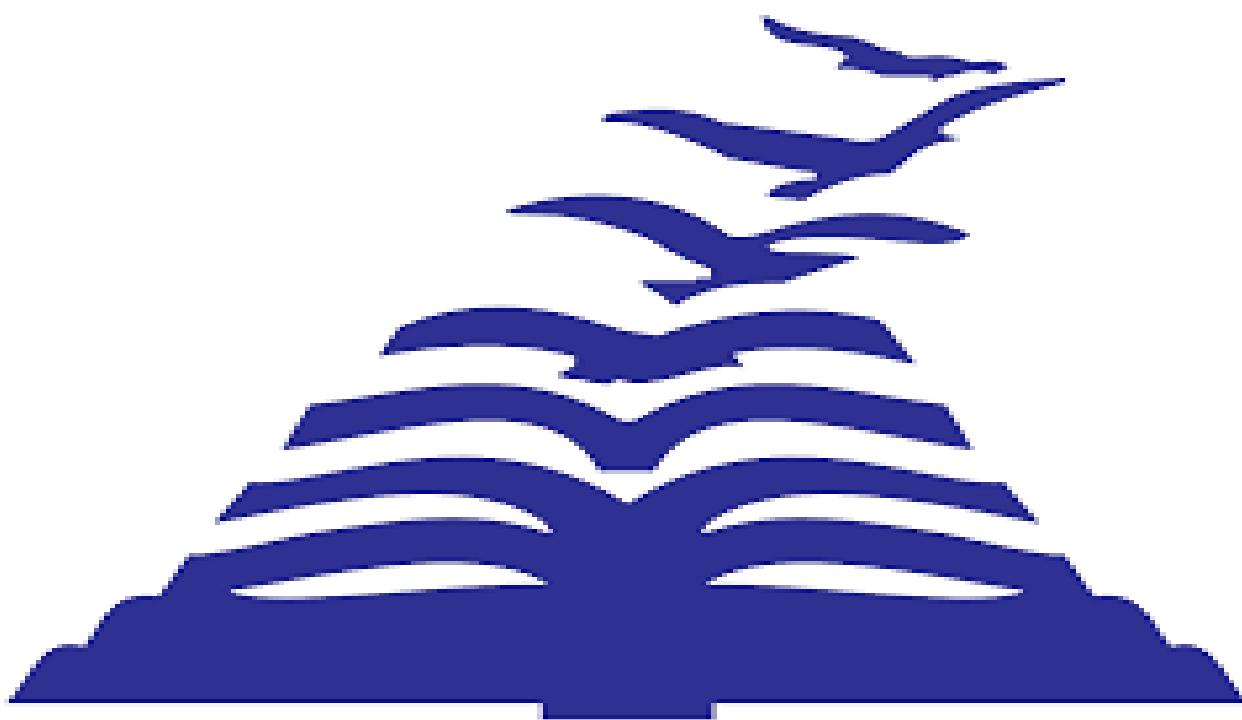
$S \rightarrow bBd$

$S \rightarrow aBe$

$S \rightarrow bAe$

$A \rightarrow c$

$B \rightarrow c$



Presidency University, Bengaluru

Canonical LR Parser (CLR)

More Powerful LR Parsers

LR parsing techniques that use one symbol of lookahead on the input.

There are two different methods:

- The "**canonical-LR**" or just "**LR**" method, which makes full use of the lookahead symbol(s). This method uses a large set of items, called the LR(1) items.
- The "**lookahead-LR**" or "**LALR**" method, which is based on the LR(0) sets of items, and has many fewer states than typical parsers based on the LR(1) items.
 - Introducing lookaheads into the LR(0) items, we can handle many more grammars with the LALR method than with the SLR method, and build parsing tables that are no bigger than the SLR tables.
 - LALR is the method of choice in most situations.

CLR Parsing algorithm:

Input: An input string w and an LR parsing table with functions action and goto for grammar G. **Output:** If w is in L(G), a bottom-up-parse for w; otherwise, an error indication.

Method: Initially, the parser has s_0 on its stack, where s_0 is the initial state, and $w\$$ in the input buffer. The parser then executes the following program:

```
set ip to point to the first input symbol of w$; repeat forever begin
    let s be the state on top of the stack and a the symbol pointed to by ip;

    if action[s, a] = shift s' then begin
        push a then s' on top of the stack; advance ip to the next input symbol end

    else if action[s, a] = reduce A→β then begin
        pop 2* | β | symbols off the stack;
        let s' be the state now on top of the stack;
        push A then goto[s', A] on top of the stack;
        output the production A→ β
    end

    else if action[s, a] = accept then
        return

    else error()
end
```

To perform CLR parsing, take grammar as input and do the following:

1. **Find LR(1) items.**
2. **Completing the closure.**
3. **Compute goto(I,X), where, I is set of items and X is grammar symbol.**

```
SetOfItems CLOSURE( $I$ ) {  
    repeat  
        for ( each item  $[A \rightarrow \alpha \cdot B\beta, a]$  in  $I$  )  
            for ( each production  $B \rightarrow \gamma$  in  $G'$  )  
                for ( each terminal  $b$  in FIRST( $\beta a$ ) )  
                    add  $[B \rightarrow \cdot \gamma, b]$  to set  $I$ ;  
    until no more items are added to  $I$ ;  
    return  $I$ ;  
}  
  
SetOfItems GOTO( $I, X$ ) {  
    initialize  $J$  to be the empty set;  
    for ( each item  $[A \rightarrow \alpha \cdot X\beta, a]$  in  $I$  )  
        add item  $[A \rightarrow \alpha X \cdot \beta, a]$  to set  $J$ ;  
    return CLOSURE( $J$ );  
}  
  
void items( $G'$ ) {  
    initialize  $C$  to CLOSURE( $\{[S' \rightarrow \cdot S, \$]\}$ );  
    repeat  
        for ( each set of items  $I$  in  $C$  )  
            for ( each grammar symbol  $X$  )  
                if ( GOTO( $I, X$ ) is not empty and not in  $C$  )  
                    add GOTO( $I, X$ ) to  $C$ ;  
    until no new sets of items are added to  $C$ ;  
}
```

Construction of LR(1) Parsing Tables

1. Construct the canonical collection of sets of LR(1) items for G' .

$$C \leftarrow \{I_0, \dots, I_n\}$$

2. Create the parsing action table as follows

- If a is a terminal, $A \rightarrow \alpha.$ $a\beta, b$ in I_i and $\text{goto}(I_i, a) = I_j$ then $\text{action}[i, a]$ is ***shift j***.
- If $A \rightarrow \alpha.$, a is in I_i , then $\text{action}[i, a]$ is ***reduce A \rightarrow α*** where $A \neq S'$.
- If $S' \rightarrow S.$, $\$$ is in I_i , then $\text{action}[i, \$]$ is ***accept***.
- If any conflicting actions generated by these rules, the grammar is not LR(1).

3. Create the parsing goto table

- for all non-terminals A , if $\text{goto}(I_i, A) = I_j$ then $\text{goto}[i, A] = j$

4. All entries not defined by (2) and (3) are errors.

5. Initial state of the parser contains $S' \rightarrow .S, \$$

CLR PARSING TABLE CONSTRUCTION

Canonical LR Parsing (CLR)

Construct the Canonical LR Parsing Table for the following Grammar

$$S \rightarrow cAd$$

$$A \rightarrow ab$$

$$A \rightarrow a$$

Step 1: Construct Augmented Grammar

$$S' \rightarrow S$$

$$S \rightarrow cAd$$

$$A \rightarrow ab$$

$$A \rightarrow a$$

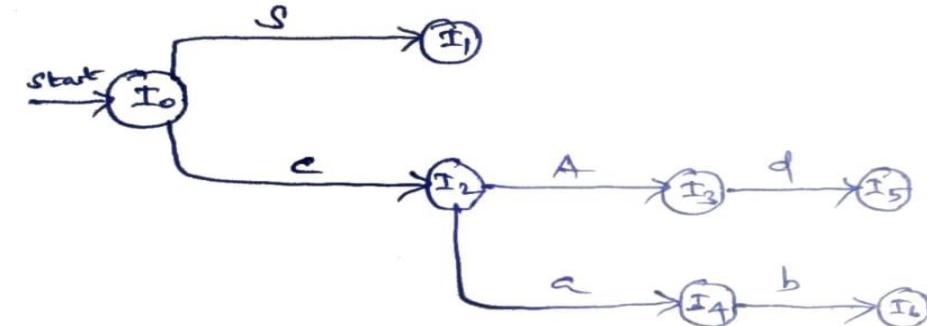
Step 2: Find LR(1) items, closure operation and compute goto function

$$(I_0) \quad S' \rightarrow \cdot S, \$$$

$$S \rightarrow \cdot cAd, \$$$

I_1	goto(I_0, S) $S \rightarrow \cdot S, \$$	I_5	goto(I_3, d) $S \rightarrow cAd, \$$
I_2	goto(I_0, c) $S \rightarrow c \cdot Ad, \$$	I_6	goto(I_4, b) $A \rightarrow ab, \$$
	$A \rightarrow \cdot ab, \$$		goto(I_0, S) $\Rightarrow I_1$
	$A \rightarrow \cdot a, \$$		goto(I_0, c) $\Rightarrow I_2$
I_3	goto(I_2, A) $S \rightarrow cA \cdot d, \$$	I_7	goto(I_2, A) $= I_3$
I_4	goto(I_2, a) $A \rightarrow a \cdot b, \$$	I_8	goto(I_2, a) $= I_4$
	$A \rightarrow a, \$$		goto(I_3, d) $= I_5$
			goto(I_4, b) $= I_6$

Step 3: Transition Diagram for LR(1) Items



Step 4: CLR Parsing Table

States	Action					Goto	
	a	b	c	d	\$	S	A
I_0					S_2		
I_1							accept
I_2		S_4					
I_3					S_5		
I_4				S_6	R_3		
I_5						R_1	
I_6					R_2		

CLR PARSING TABLE CONSTRUCTION

Construct CLR Parsing Table for the following Grammar

$$S \rightarrow L = R$$

$$S \rightarrow R$$

$$L \rightarrow *R$$

$$L \rightarrow id$$

$$R \rightarrow L$$

Step1: Construct Augmented Grammar

$$S^1 \rightarrow S$$

$$S \rightarrow L = R$$

$$S \rightarrow R$$

$$L \rightarrow *R$$

$$L \rightarrow id$$

$$R \rightarrow L$$

Step2: Find LR(0) items, calculate closure and goto operations

$$S^1 \rightarrow \cdot S, \$$$

$$S \rightarrow \cdot L = R, \$$$

$$S \rightarrow \cdot R, \$$$

$$L \rightarrow \cdot *R, =$$

$$L \rightarrow \cdot id, =$$

$$R \rightarrow \cdot L, \$$$

$$\begin{aligned} L &\rightarrow \cdot *R, \$ \\ L &\rightarrow \cdot id, \$ \end{aligned}$$

$$S^1 \rightarrow \cdot S, \$$$

$$S \rightarrow \cdot L = R, \$$$

$$S \rightarrow \cdot R, \$$$

$$L \rightarrow \cdot *R, =\$$$

$$L \rightarrow \cdot id, =\$$$

$$R \rightarrow \cdot L, \$$$

I_1	$\text{goto}(\text{I}_0, S)$ $S^1 \rightarrow S \cdot, \$$	I_2	$\text{goto}(\text{I}_2, =)$ $S \rightarrow L \cdot = R, \$$ $R \rightarrow \cdot L, \$$	I_3	$\text{goto}(\text{I}_0, R)$ $S \rightarrow R \cdot, \$$	I_4	$\text{goto}(\text{I}_4, R)$ $L \rightarrow *R \cdot, = \$$	I_5	$\text{goto}(\text{I}_0, id)$ $L \rightarrow id \cdot, \$$
I_2	$\text{goto}(\text{I}_0, L)$ $S \rightarrow L \cdot = R, \$$ $R \rightarrow L \cdot, \$$	I_7	$\text{goto}(\text{I}_7, R)$ $L \rightarrow *R \cdot, = \$$	I_6	$\text{goto}(\text{I}_4, L)$ $R \rightarrow L \cdot, = \$$	I_7	$\text{goto}(\text{I}_7, *)$ $L \rightarrow * \cdot R, = \$$	I_8	$\text{goto}(\text{I}_8, L)$ $R \rightarrow L \cdot, = \$$
I_3	$\text{goto}(\text{I}_0, R)$ $S \rightarrow R \cdot, \$$	I_8	$\text{goto}(\text{I}_8, L)$ $R \rightarrow L \cdot, = \$$	I_4	$\text{goto}(\text{I}_4, *)$ $L \rightarrow * \cdot R, = \$$	I_9	$\text{goto}(\text{I}_9, *)$ $L \rightarrow * \cdot R, = \$$	I_{10}	$\text{goto}(\text{I}_{10}, id)$ $L \rightarrow id \cdot, \$$
I_4	$\text{goto}(\text{I}_4, *)$ $L \rightarrow * \cdot R, = \$$	I_9	$\text{goto}(\text{I}_9, *)$ $L \rightarrow * \cdot R, = \$$	I_5	$\text{goto}(\text{I}_5, id)$ $L \rightarrow id \cdot, = \$$	I_{11}	$\text{goto}(\text{I}_{11}, R)$ $L \rightarrow *R \cdot, \$$	I_{12}	$\text{goto}(\text{I}_{12}, L)$ $\Rightarrow \text{I}_{10}$
I_5	$\text{goto}(\text{I}_5, id)$ $L \rightarrow id \cdot, = \$$	I_{11}	$\text{goto}(\text{I}_{11}, *)$ $L \rightarrow *R \cdot, \$$	I_{12}	$\text{goto}(\text{I}_{12}, id)$ $L \rightarrow id \cdot, = \$$	I_{13}	$\text{goto}(\text{I}_{13}, id)$ $L \rightarrow id \cdot, = \$$	I_{14}	$\text{goto}(\text{I}_{14}, id)$ $L \rightarrow id \cdot, = \$$

CLR PARSING TABLE CONSTRUCTION



Step 3: Construction of CLR Parsing Table

State	Action				Goto		
	=	*	id	\$	S	L	R
0	-	s_4	s_5		1	2	3
1					Accept		
2	s_b				R_5		
3					R_2		
4		s_4	s_5			8	7
5	R_4				R_4		
6		s_{11}	s_{12}			10	9
7	R_3				R_3		
8	R_5				R_5		
9					R_1		
10					R_5		
11		s_{11}	s_{12}			10	13
12					R_4		
13					R_3		

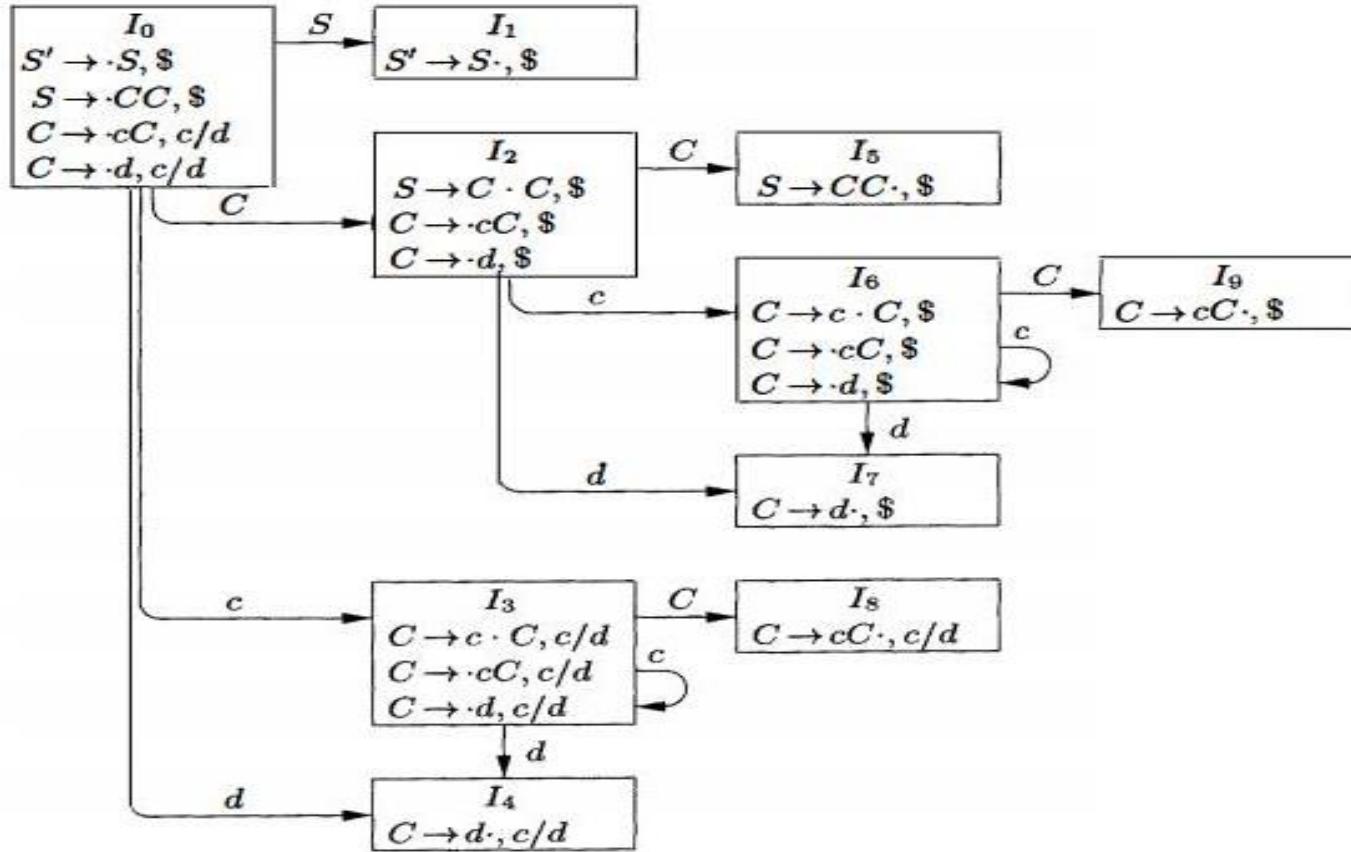
CLR PARSING TABLE CONSTRUCTION

Construct CLR for the following CFG

$S \rightarrow CC$

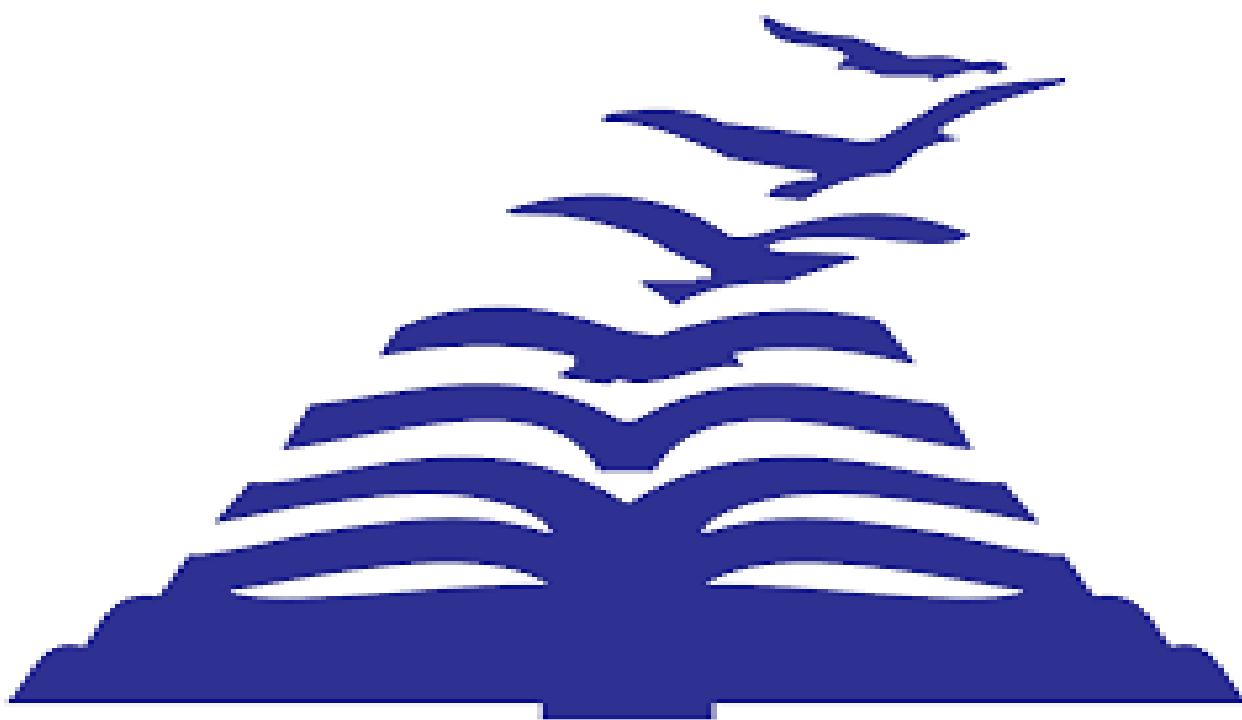
$C \rightarrow cC$

$C \rightarrow d$



Canonical LR (CLR) Parsing Table

STATE	ACTION			GOTO	
	c	d	$\$$	S	C
0	s3	s4		1	2
1					acc
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5				r1	
6	s6	s7			9
7				r3	
8	r2	r2			
9				r2	



Presidency University, Bengaluru

Look Ahead LR Parser (LALR)

- The "**lookahead-LR**" or "**LALR**" method, which is based on the LR(0) sets of items, and has many fewer states than typical parsers based on the LR(1) items.
 - Introducing lookaheads into the LR(0) items, we can handle many more grammars with the LALR method than with the SLR method, and build parsing tables that are no bigger than the SLR tables.
 - LALR is the method of choice in most situations. (Eg., YACC).
- For a comparison of parser size, the **SLR and LALR tables** for a grammar always have the **same number of states**, and this number is typically several hundred states for a language like C. The canonical LR table would typically have several thousand states for the same-size language. Thus, **it is much easier and more economical to construct SLR and LALR tables than the canonical LR tables.**

Algorithm: LALR Table Construction

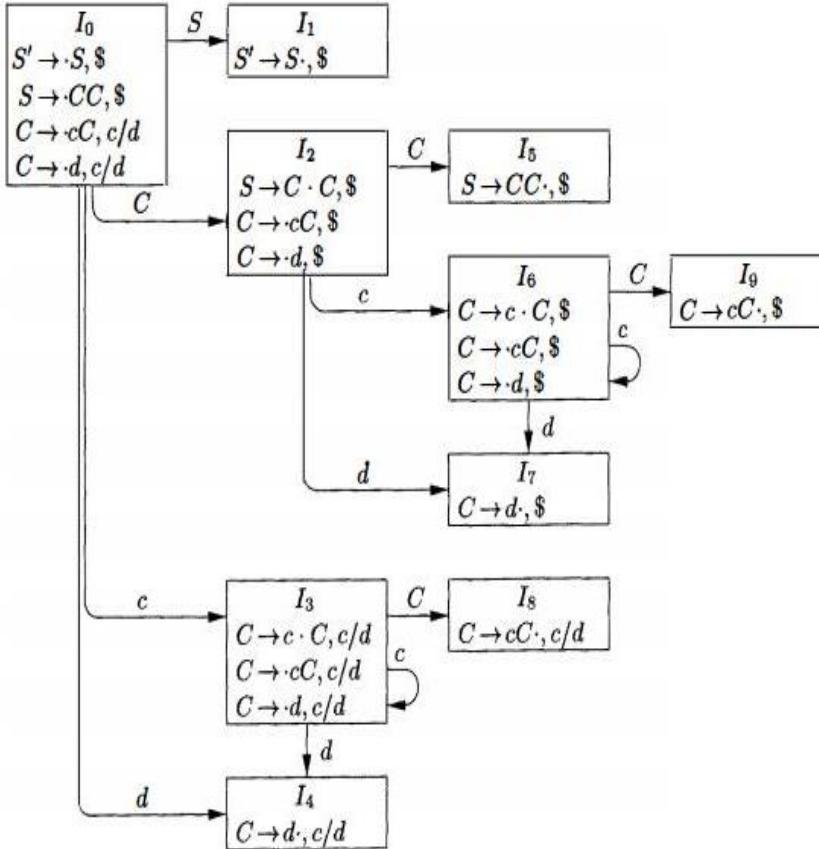
INPUT : An augmented grammar G' .

OUTPUT : The LALR parsing-table functions ACTION and GOTO for G' .

METHOD :

1. Construct $C = \{I_0, I_1, \dots, I_n\}$, the collection of sets of LR(1) items.
2. For each core present among the set of LR(1) items, find all sets having that core, and replace these sets by their union.
3. Let $C' = \{J_0, J_1, \dots, J_m\}$ be the resulting sets of LR(1) items. The parsing actions for state i are constructed from J_i in the same manner as in Algorithm 4.56. If there is a parsing action conflict, the algorithm fails to produce a parser, and the grammar is said not to be LALR(1).
4. The GOTO table is constructed as follows. If J is the union of one or more sets of LR(1) items, that is, $J = I_1 \cap I_2 \cap \dots \cap I_k$, then the cores of $\text{GOTO}(I_1, X)$, $\text{GOTO}(I_2, X), \dots, \text{GOTO}(I_k, X)$ are the same, since I_1, I_2, \dots, I_k all have the same core. Let K be the union of all sets of items having the same core as $\text{GOTO}(I_1, X)$. Then $\text{GOTO}(J, X) = K$.

LALR PARSING TABLE CONSTRUCTION



$LR(1)$	I_3 $C \rightarrow c \cdot C, c/d$ $c/d \$$	I_7 $C \rightarrow d \cdot, \$$	I_7 $C \rightarrow d \cdot, \$$
$S \rightarrow CC$	I_0 $S \rightarrow \cdot S, \$$	I_8 $S \rightarrow CC, \cdot \$$	I_8 $S \rightarrow CC, \cdot \$$
$C \rightarrow cC$	I_0 $S \rightarrow \cdot S, \$$	I_3 $C \rightarrow \cdot cC, c/d$	I_3 $C \rightarrow \cdot cC, c/d$
$C \rightarrow d$	I_0 $S \rightarrow \cdot S, \$$	I_7 $C \rightarrow \cdot d, \$$	I_7 $C \rightarrow \cdot d, \$$
	I_0 $S \rightarrow \cdot S, \$$	I_4 $C \rightarrow d \cdot, c/d$	I_4 $C \rightarrow d \cdot, c/d$
	I_1 $S' \rightarrow S, \cdot \$$	I_5 $S \rightarrow CC, \cdot \$$	I_5 $S \rightarrow CC, \cdot \$$
	I_2 $S \rightarrow C \cdot C, \$$	I_6 $C \rightarrow c \cdot C, \$$	I_6 $C \rightarrow c \cdot C, \$$
	I_3 $C \rightarrow c \cdot C, c/d$	I_7 $C \rightarrow d \cdot, \$$	I_7 $C \rightarrow d \cdot, \$$
	I_4 $C \rightarrow d, c/d$	I_8 $C \rightarrow cC, \cdot c/d$	I_8 $C \rightarrow cC, \cdot c/d$
	I_5 $S \rightarrow CC, \cdot \$$	I_9 $C \rightarrow cC, \cdot \$$	I_9 $C \rightarrow cC, \cdot \$$
	I_6 $C \rightarrow c \cdot C, \$$	I_7 $C \rightarrow d \cdot, \$$	I_7 $C \rightarrow d \cdot, \$$
	I_7 $C \rightarrow d, \cdot \$$	I_8 $C \rightarrow cC, \cdot c/d$	I_8 $C \rightarrow cC, \cdot c/d$
	I_8 $C \rightarrow cC, \cdot c/d$	I_9 $C \rightarrow cC, \cdot \$$	I_9 $C \rightarrow cC, \cdot \$$
	I_9 $C \rightarrow cC, \cdot \$$	I_0 $S \rightarrow \cdot S, \$$	I_0 $S \rightarrow \cdot S, \$$

Step 3: Construct LALR(1) items

$I_0, I_1, I_2, I_{36}, I_{47}, I_5, I_{89}$

$I_{36}:$

- $C \rightarrow c \cdot C, c/d | \$$
- $C \rightarrow \cdot cC, c/d | \$$
- $C \rightarrow \cdot d, c/d | \$$

$I_{47}:$

- $C \rightarrow d \cdot, c/d | \$$

$I_{89}:$

- $C \rightarrow cC \cdot, c/d | \$$

LALR PARSING TABLE CONSTRUCTION



$R_1: S \rightarrow CC$

$R_2: C \rightarrow CC$

$R_3: C \rightarrow d$

goto(I_0, S) = I_1

goto(I_0, C) = I_2

goto(I_0, c) = I_{36}

goto(I_0, d) = I_{47}

goto(I_2, C) = I_5

goto(I_2, c) = I_{36}

goto(I_2, d) = I_{47}

goto(I_{36}, C) = I_{89}

goto(I_{36}, c) = I_{36}

goto(I_{36}, d) = I_{47}

goto(I_{36}, C)

goto(I_{36}, c)

goto(I_{36}, d)

states	Action			Goto	
	c	d	s	S	C
0	S_{36}	S_{47}		1	2
1				Accept	
2	S_{36}	S_{47}			5
36	S_{36}	S_{47}			89
47	R_3	R_3	R_3		
5				R_1	
89	R_2	R_2	R_2		

LALR PARSING TABLE CONSTRUCTION

$S \rightarrow AaAb$ $S \rightarrow BbBa$ $A \rightarrow \epsilon$ $B \rightarrow \epsilon$ <hr/> <u>Step 1:</u> AG $S' \rightarrow S$ $S \rightarrow AaAb$ $S \rightarrow BbBa$ $A \rightarrow \epsilon$ $B \rightarrow \epsilon$ <hr/> <u>LR(1)</u> , closures <u>goto operations</u>	I_1 goto(I_0, S) $S' \rightarrow S \cdot, \$$	I_7 goto(I_5, B) $S \rightarrow BbB \cdot a, \$$
	I_2 goto(I_0, A) $S \rightarrow A \cdot aAb, \$$	I_8 goto(I_6, b) $S \rightarrow AaA \cdot b, \$$
	I_3 goto(I_0, B) $S \rightarrow B \cdot bBa, \$$	I_9 goto(I_7, a) $S \rightarrow BbBa \cdot, \$$
	I_4 goto(I_2, a) $S \rightarrow Aa \cdot Ab, \$$ $B \rightarrow \cdot, b$	<u>Step 3: LALR(1) items</u> $I_0, I_1, I_2, I_3, I_4,$ I_5, I_6, I_7, I_8, I_9
	I_5 goto(I_3, b) $S \rightarrow Bb \cdot Ba, \$$ $B \rightarrow \cdot, a$	
	I_6 goto(I_4, A) $S \rightarrow AaA \cdot b, \$$	

$R_1: S \rightarrow AaAb$

$R_2: S \rightarrow BbBa$

$R_3: A \rightarrow \epsilon$

$R_4: B \rightarrow \epsilon$

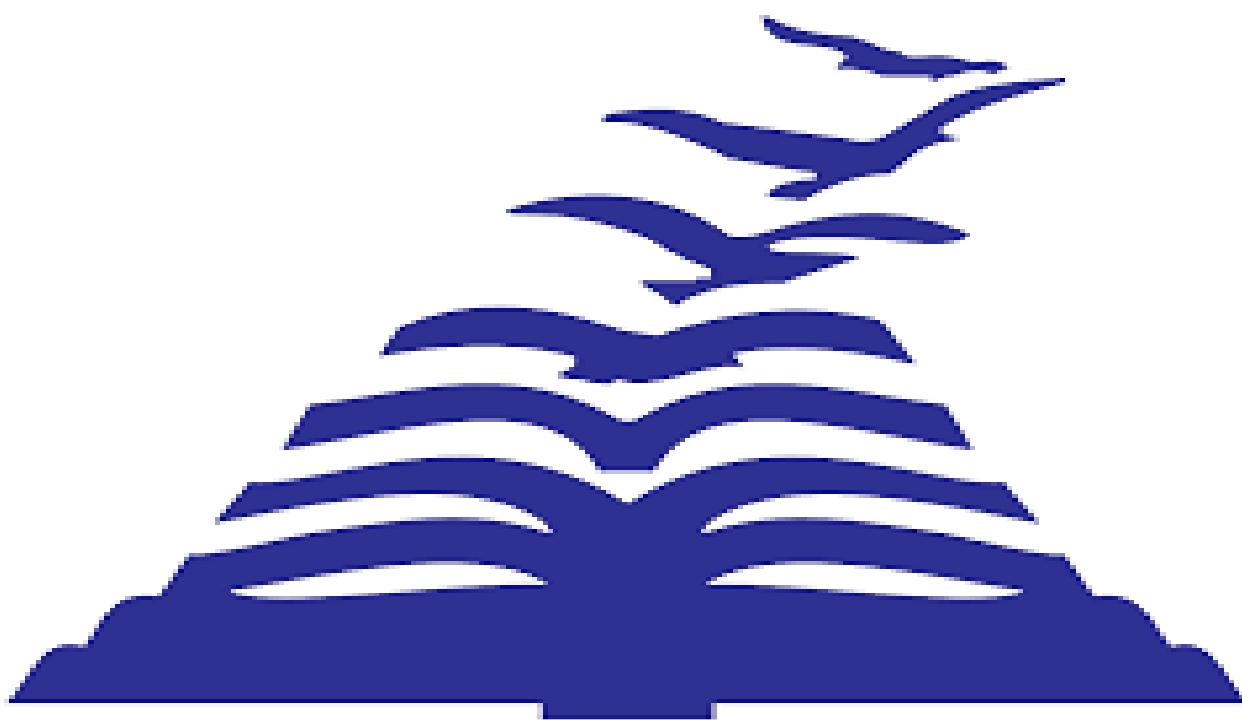
states	Action			Goto		
	a	b	\$	S	A	B
0	R_3	R_4			1	2
1						Accept
2		S_4				
3			S_5			
4				R_3		6
5				R_4		7
6			S_8			
7			S_9			
8					R_1	
9					R_2	

Construct SLR ,CLR and LALR parser table for the following grammar.Observe the differences and write it.

$S \rightarrow XX$

$X \rightarrow aX$

$X \rightarrow b$



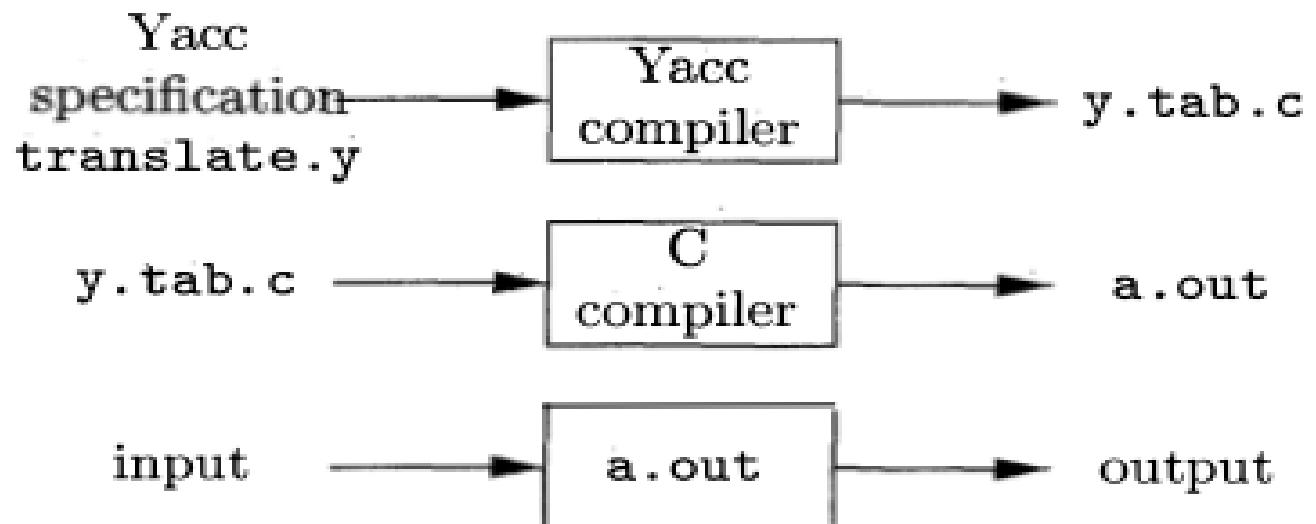
Presidency University, Bengaluru

PARSER GENERATORS (YACC)

Parser Generators - (YACC)

- **Yacc** stands for "yet another compiler-compiler," reflecting the popularity of parser generators in the early 1970s when the first version of **Yacc** was created by **S. C. Johnson**.
- **Yacc** is available as a command on the **UNIX system**, and has been used to help implement many production compilers.

A translator can be constructed using **Yacc** in the following manner



First, a file, say **translate.y**, containing a **Yacc** specification of the translator is prepared. The UNIX system command

```
yacc translate.y
```

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

The program **y.tab.c** is a representation of an LALR parser written in C, along with other C routines that the user may have prepared.

By **compiling y.tab.c** along with the **ly library** that contains the **LR parsing program** using the command

```
cc y.tab.c -ly
```

we obtain the desired object program **a.out** that performs the translation specified by the original Yacc program. If other procedures are needed, they can be compiled or loaded with y.tab.c, just as with any C program.

A Yacc source program has three parts:

declarations

%%

translation rules

%%

supporting C routines

The Declarations Part

There are **two sections** in the declarations part of a Yacc program; **both are optional**.

- In the first section, we put ordinary C declarations, delimited by `%{` and `%}`.
- Here we place **declarations of any temporaries used by the translation rules or procedures of the second and third sections.**

For Example, We may include

```
# include      <ctype.h>
```

Also in the declarations part are **declarations of grammar tokens**.

For Example, The statement

```
%token DIGIT
```

declares DIGIT to be a token.

Tokens declared in this section can then be used in the second and third parts of the Yacc specification. If Lex is used to create the lexical analyzer that passes token to the Yacc parser, then these token declarations are also made available to the analyzer generated by Lex.

The Translation Rules Part

- In the part of the Yacc specification after the first %% pair, we put the translation rules.
- Each rule consists of a grammar production and the associated semantic action.

$$\langle \text{head} \rangle \rightarrow \langle \text{body} \rangle_1 \mid \langle \text{body} \rangle_2 \mid \dots \mid \langle \text{body} \rangle_n$$

would be written in Yacc as

```
 $\langle \text{head} \rangle : \langle \text{body} \rangle_1 \{ \langle \text{semantic action} \rangle_1 \}$ 
|  $\langle \text{body} \rangle_2 \{ \langle \text{semantic action} \rangle_2 \}$ 
...
|  $\langle \text{body} \rangle_n \{ \langle \text{semantic action} \rangle_n \}$ 
;
```

- **Alternative bodies** can be **separated by a vertical bar**, and a semicolon follows each head with its alternatives and their semantic actions.
- The **first head** is taken to be the **start symbol**.

- A **Yacc semantic action** is a **sequence of C statements**.
- In a semantic action, the **symbol \$\$ refers** to the **attribute value** associated with the **nonterminal of the head**, while **\$i refers** to the value associated with the **ith grammar symbol (terminal or nonterminal) of the body**.
- The **semantic action is performed** whenever we **reduce by the associated production**, so normally the **semantic action computes a value for \$\$ in terms of the \$i's**.

$E \rightarrow E + T \mid T$

and their associated semantic actions as:

```
expr : expr '+' term      { $$ = $1 + $3; }
      | term
      ;
```

Notice that we have added a new starting production

```
line : expr '\n'   { printf("%d\n", $1); }
```

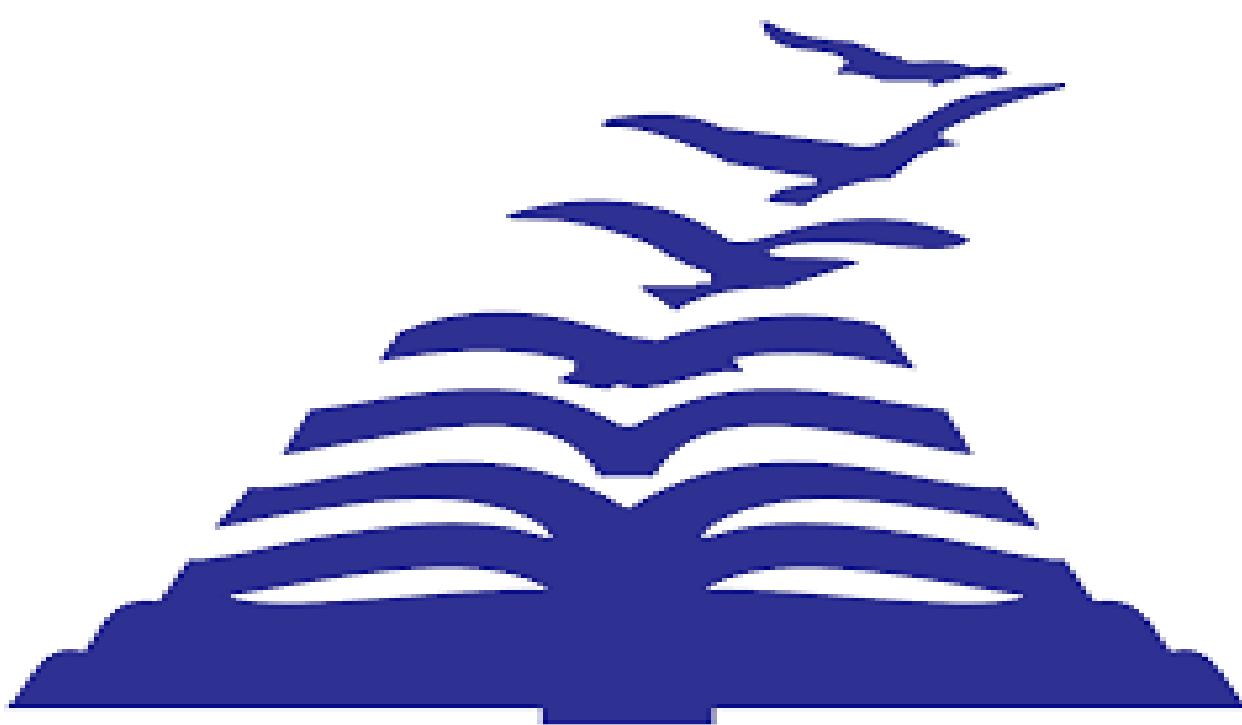
The Supporting C - Routines Part

- The third part of a Yacc specification consists of **supporting C-routines**.
- A lexical analyzer by the name **yylex()** must be provided.
- Using Lex to produce yylex() is a common choice;
- The lexical analyzer yylex() produces tokens consisting of a token name and its associated attribute value.
- If a token name such as DIGIT is returned, the token name must be declared in the first section of the Yacc specification.
- The attribute value associated with a token is communicated to the parser through a Yacc-defined variable **yyval**.

Parser Generators - (YACC)

```
%{  
#include <ctype.h>  
#include <stdio.h>  
#define YYSTYPE double /* double type for Yacc stack */  
%}  
%token NUMBER  
  
%left '+' '-'  
%left '*' '/'  
%right UMINUS  
%%  
  
lines : lines expr '\n' { printf("%g\n", $2); }  
| lines '\n'  
| /* empty */  
;  
expr : expr '+' expr { $$ = $1 + $3; }  
| expr '-' expr { $$ = $1 - $3; }  
| expr '*' expr { $$ = $1 * $3; }  
| expr '/' expr { $$ = $1 / $3; }  
| '(' expr ')' { $$ = $2; }  
| '-' expr %prec UMINUS { $$ = - $2; }  
| NUMBER  
;  
%%  
yylex() {  
    int c;  
    while ( ( c = getchar() ) == ' ' );  
    if ( (c == '.') || (isdigit(c)) ) {  
        ungetc(c, stdin);  
        scanf("%lf", &yyval);  
        return NUMBER;  
    }  
    return c;  
}
```

Yacc specification for a more advanced desk calculator.



Presidency University, Bengaluru

Syntax Directed Definition and Translation

Syntax Directed Translation

- **Syntax-directed translation (SDT)** refers to a method of compiler implementation where the source language translation is completely driven by the parser.
- The parsing process and parse trees are used to direct semantic analysis and the translation of the source program.
- We can augment grammar with information to control the semantic analysis and translation. Such grammars are called **attribute grammars**.

Syntax Directed Translation

- ❖ Associate attributes with each grammar symbol that describes its properties.
- ❖ An **attribute has a name** and an **associated value**.
- ❖ With **each production in a grammar, give semantic rules or actions**.
- ❖ The **general approach to syntax-directed translation is to construct a parse tree or syntax tree and compute the values of attributes at the nodes of the tree by visiting them in some order**.

There are two ways to represent the semantic rules associated with grammar symbols.

- **Syntax-Directed Definitions (SDD)**
- **Syntax-Directed Translation Schemes (SDT)**

- ❖ A **syntax-directed definition (SDD)** is a context free grammar together with attributes and rules.
- ❖ **Attributes are associated with grammar symbols and rules are associated with productions.**

PRODUCTION

$$E \rightarrow E_1 + T$$

SEMANTIC RULE

$$E.\text{code} = E_1.\text{code} \parallel T.\text{code} \parallel '+'$$

- ❖ **SDDs are highly readable** and **give high-level specifications** for translations but they hide many implementation details.
For example, they do not specify order of evaluation of semantic actions.
- ❖ **Syntax-Directed Translation Schemes (SDT)** embed program fragments called semantic actions within production bodies.
- ❖ **SDTs are more efficient than SDDs** as they indicate the order of evaluation of semantic actions associated with a production rule.

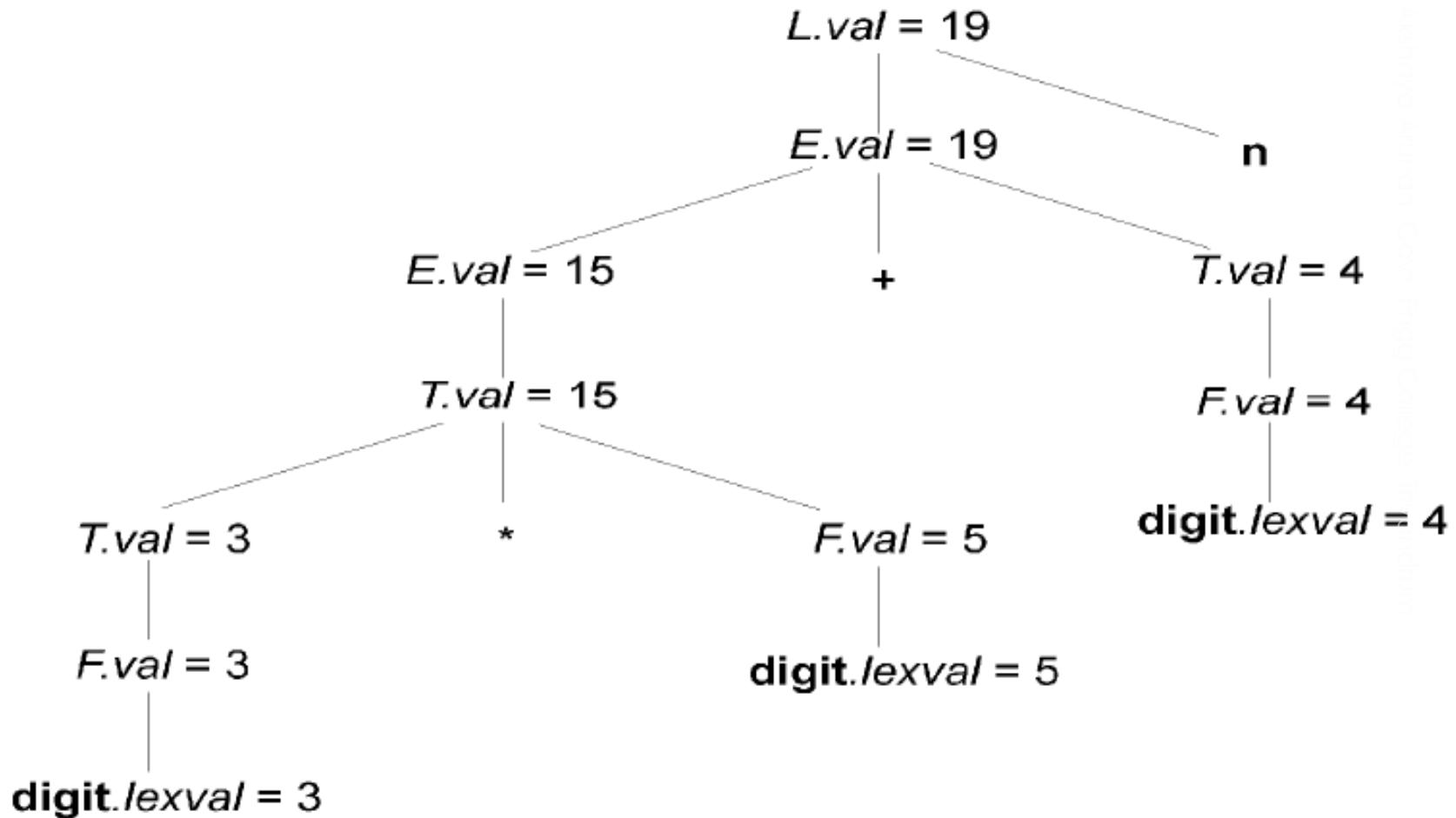
Synthesized Attributes

Production	Semantic Rules
1) $L \rightarrow E \text{ n}$	$L.\text{val} = E.\text{val}$
2) $E \rightarrow E_1 + T$	$E.\text{val} = E_1.\text{val} + T.\text{val}$
3) $E \rightarrow T$	$E.\text{val} = T.\text{val}$
4) $T \rightarrow T_1 * F$	$T.\text{val} = T_1.\text{val} \times F.\text{val}$
5) $T \rightarrow F$	$T.\text{val} = F.\text{val}$
6) $F \rightarrow (E)$	$F.\text{val} = E.\text{val}$
7) $F \rightarrow \text{digit}$	$F.\text{val} = \text{digit}.lexval$

- Each of the non-terminals has a **single synthesized attribute, called val.**
- An SDD that involves only synthesized attributes is called **S-attributed.**
- **Each rule computes an attribute for the non-terminal at the head of a production from attributes taken from the body of the production.**

- A parse tree, showing the value(s) of its attribute(s) is called an **annotated parse tree**.
- With synthesized attributes, **evaluate attributes in bottom-up order**.

Annotated Parse Tree for $3 * 5 + 4n$

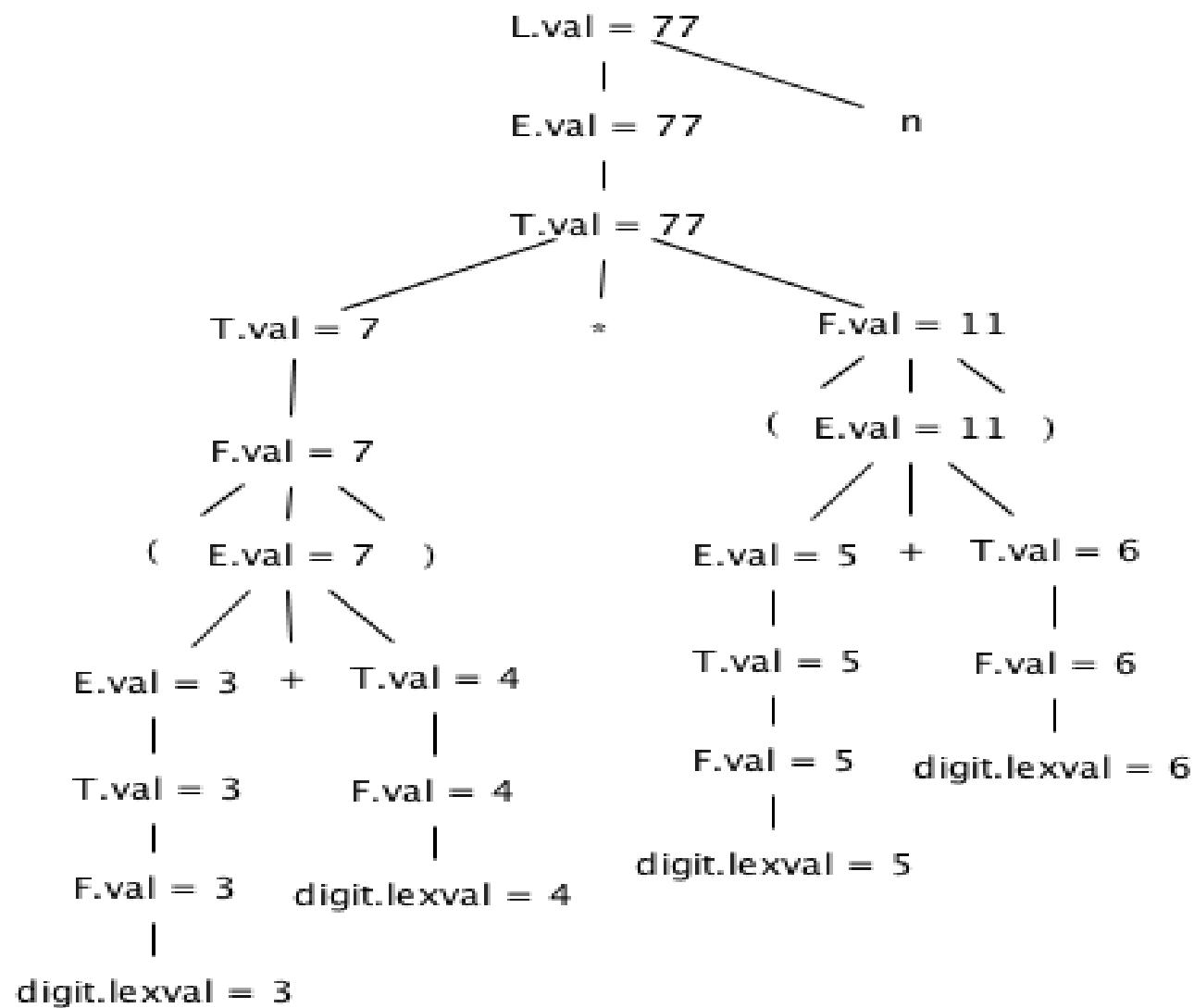


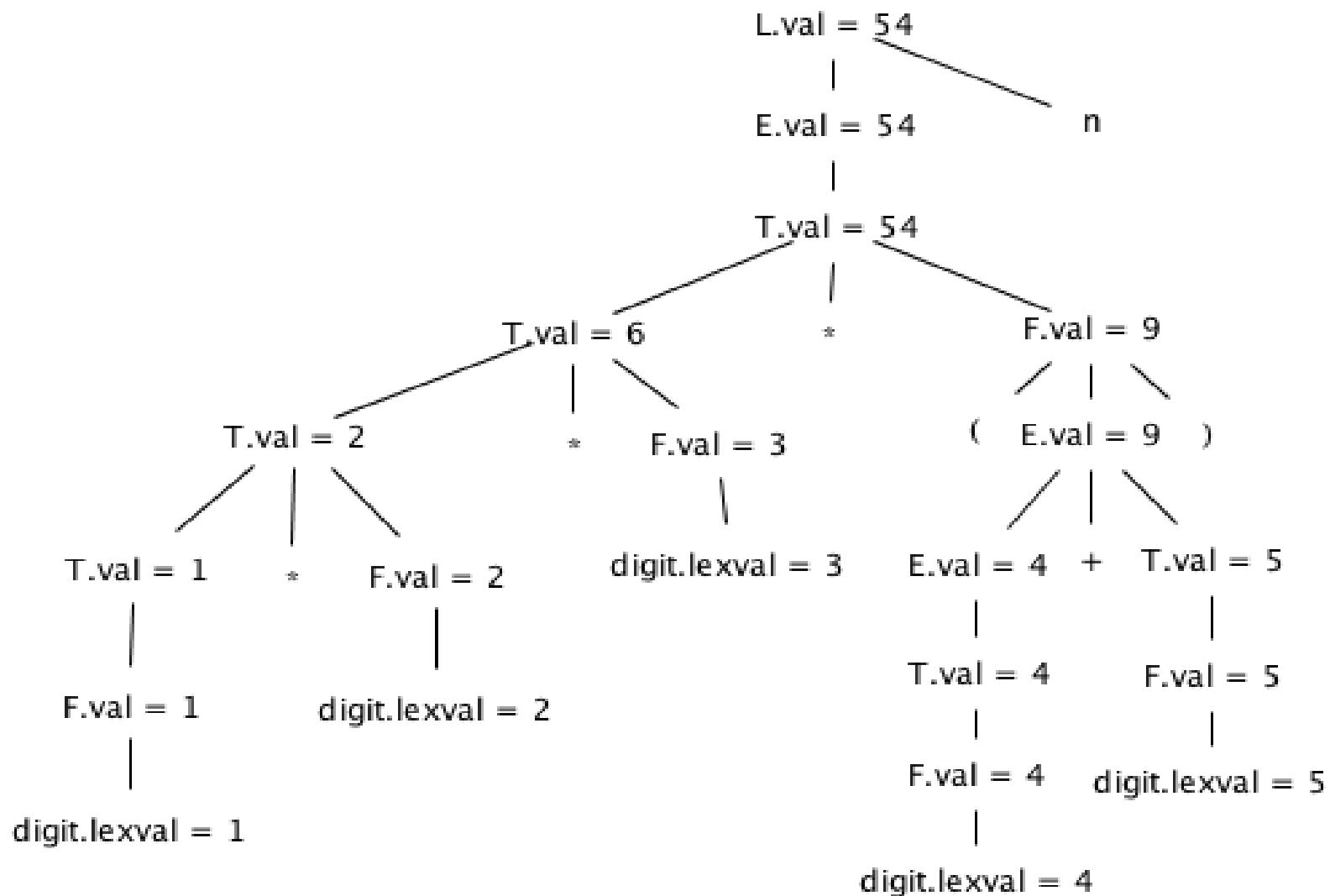
Give annotated parse trees for the following expressions:

$$(3 + 4) * \underline{(5 + 6)} \text{ n.}$$

$$1 * 2 * 3 * (4 + 5) \text{ n.}$$

$$(9 + 8 * (7 + 6) + 5) * 4 \text{ n.}$$





Inherited Attributes

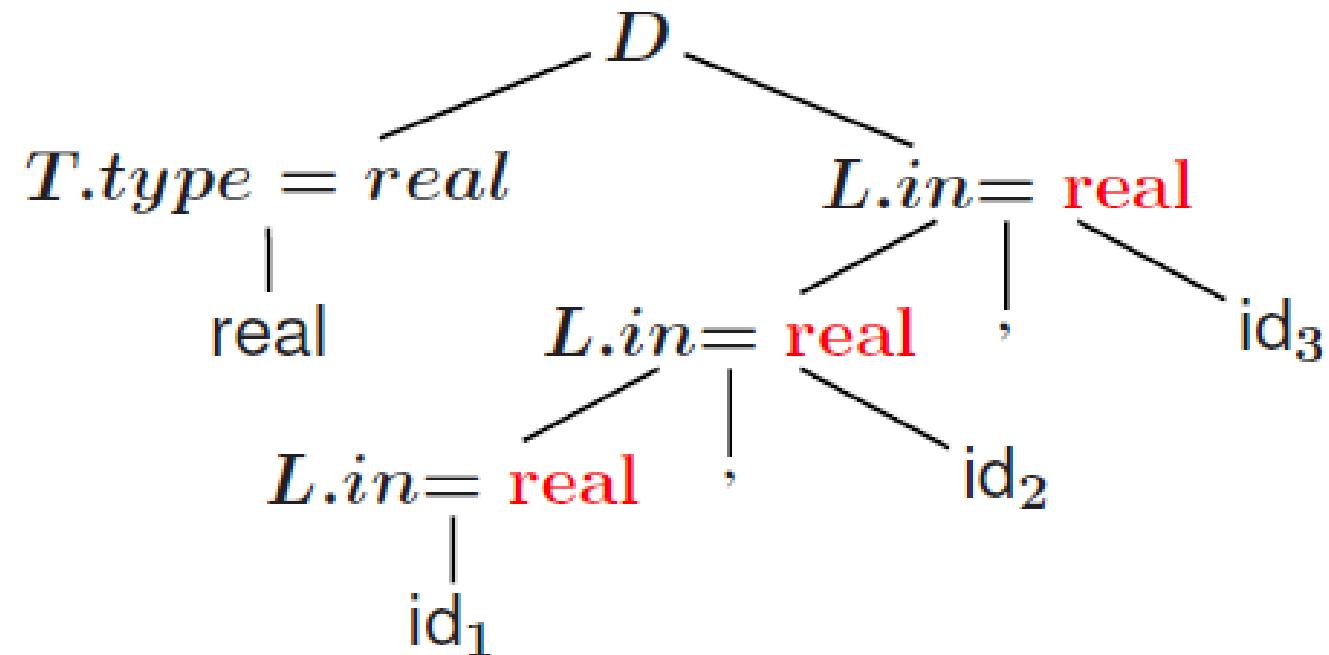
- **An Inherited Attribute** for a non-terminal B at a parse-tree node N is defined by a semantic rule associated with the production at the parent of N. The production must have B as a symbol in its body.

- **An inherited attribute at node N is defined only in terms of attribute values at N's parent, and N's siblings.**

SDD for expression grammar with inherited attributes

PRODUCTION	SEMANTIC RULE
$D \rightarrow TL$	$L.in := T.type$
$T \rightarrow \text{int}$	$T.type := \text{integer}$
$T \rightarrow \text{real}$	$T.type := \text{real}$
$L \rightarrow L_1, \text{id}$	$L_1.in := L.in; \text{ addtype(id.entry, } L.in)$
$L \rightarrow \text{id}$	$\text{addtype(id.entry, } L.in)$

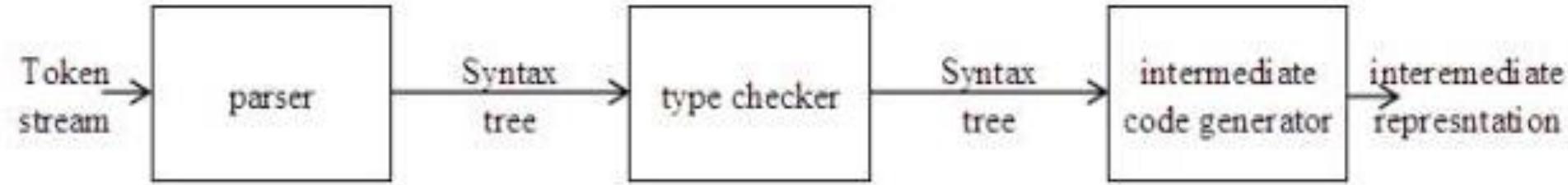
Annotated parse-tree for the input real id1, id2, id3





Presidency University, Bengaluru

TYPE CHECKING



- A compiler must check that the source program follows both syntactic and **semantic conventions** of the source language. This checking, called **static checking**, detects and reports programming errors.
- A typechecker verifies that the type of a construct matches that expected by its context.

For example : arithmetic operator mod in Pascal requires integer operands, so a type checker verifies that the operands of mod have type integer. Type information gathered by a type checker may be needed when code is generated.

Some examples of static checks:

- **Type checks** - A compiler should report an error if an operator is applied to an incompatible operand.

Example: If an array variable and function variable are added together.

- **Flow-of-control checks** - Statements that cause flow of control to leave a construct must have some place to which to transfer the flow of control.

Example: An enclosing statement, such as break, does not exist in switch statement or looping statement.

- **Uniqueness checks** – There is a situation in which an object must be defined exactly once.

Example: int a,b; float b;

- **Name-related checks** – Sometimes, the same name appear two or more times.

Example: In Ada language, a loop or block may have a name that appears at the beginning and end the construct.



Type System

Type Systems

- A type system is a collection of rules that assign types to program constructs (more constraints added to checking the validity of the programs, violation of such constraints indicate errors).
- A language's type system specifies which operations are valid for which types.
- Type systems provide a concise formalization of the semantic checking rules.
- Type rules are defined on the structure of expressions.
- Type rules are language specific.

Type Expressions

- A **type expression** is either a basic type or is formed by applying an operator called a type constructor to a type expression. The sets of basic types and constructors depend on the language to be checked.

The following are some of type expressions:

- A basic type is a type expression. Typical basic types for a language include boolean, char, integer, float, and void(the absence of a value). **type_error** is a special basic type.
- A type constructor applied to type expressions. Constructors include:
 - **Arrays** : If T is a type expression, then array(I, T) is a type expression denoting the type of an array with elements of type T and index set I. I is often a range of integers. Ex. int a[25] ;
 - **Products** : If T_1 and T_2 are type expressions, then their Cartesian product $T_1 \times T_2$ is a type expression. \times associates to the left and that it has higher precedence. Products are introduced for completeness; they can be used to represent a list or tuple of types (e.g., for function parameters).
 - **Records** : A record is a data structure with named fields. A type expression can be formed by applying the record type constructor to the field names and their types.
 - **Pointers** : If T is a type expression, then pointer (T) is a type expression denoting the type "pointer to an object of type T". For example: **int a;** **int *p=&a;**
 - **Functions** : Mathematically, a function maps depends on one set (domain) to another set(range).
Function F : D -> R.A type expression can be formed by using the type constructor \rightarrow for **function** types. We write $s \rightarrow t$ for "function from type s to type t".
- Type expressions may contain variables whose values are themselves **type expressions**.

Static and Dynamic Type Checking

- Type checking is the process of verifying and applying the constraints of types, and it can occur either at compile time (i.e. statically) or at runtime (i.e. dynamically).
- A language is statically-typed if the type of a variable is known at compile time instead of at runtime. Common examples of statically-typed languages such as Ada, C, C++, C#, JADE, Java, Fortran, Haskell, ML, Pascal, and Scala.
- Dynamic type checking is the process of verifying the type safety of a program at runtime. Dynamically-typed languages include Groovy, JavaScript, Lisp, Lua, Objective-C, PHP, Prolog, Python, Ruby, Smalltalk and Tcl.
- A language is strongly typed, if its compiler can guarantee that the programs it accepts will execute without **type errors**. Eg. For integers int array[255];

Error Recovery

- Since type checking has the potential for catching errors in programs. It is important for a **type checker** to do something reasonable when an error is discovered.
- At the very least, the compiler must report the nature and location of the error.
- It is desirable for the type checker to **recover** from errors, so it can check the rest of the input.

Specification of a Type Checker

Specification of a Simple Type Checker

- Specification of a simple type checker for a simple language in which the type of each **identifier** must be declared before the identifier is used.
- The **type checker** is a translation scheme that synthesizes the type of each expression from the types of its subexpressions.
- The type checker can handle **arrays**, **pointers**, **statements**, and **functions**.
- Specification of a simple type checker includes the following:
 - A Simple Language
 - Type Checking of Expressions
 - Type Checking of Statements
 - Type Checking of Functions

A Simple Language

Consider the following grammar:

$P \rightarrow D ; E$

$D \rightarrow D ; D \mid id : T$

$T \rightarrow char \mid integer \mid array [num] of T \mid \uparrow T$

$E \rightarrow literal \mid num \mid id \mid E \text{ mod } E \mid E [E] \mid E \uparrow$

Translation scheme:

$P \rightarrow D ; E$

$D \rightarrow D ; D$

$D \rightarrow id : T \{ \text{addtype} (id.entry, T.type) \}$

$T \rightarrow char \{ T.type := \text{char} \}$

$T \rightarrow integer \{ T.type := \text{integer} \}$

$T \rightarrow \uparrow T1 \{ T.type := \text{pointer}(T1.type) \}$

$T \rightarrow array [num] of T1 \{ T.type := \text{array} (1\dots num.val, T1.type) \}$

In the above language,

- There are two basic types : char and integer ; → type_error is used to signal errors;
- the prefix operator \uparrow builds a pointer type. Example , \uparrow integer leads to the type expression
pointer (integer).

Type Checking of Expressions

- The synthesized attribute type for E gives the type expression **assigned** by the type system to the expression generated by E.
- The following semantic rules say that **constants** represented by the tokens **literal** and **num** have type **char** and **integer**, respectively:

Rule	Semantic Rule
$E \rightarrow \text{literal}$	{ E.type := char }
$E \rightarrow \text{num}$	{ E.type := integer }

- A function **lookup(e)** is used to fetch the type saved in the **symbol-table** entry pointed to by **e**.
- When an **identifier** appears in an expression, its declared type is fetched and assigned to the attribute type;

$E \rightarrow \text{id}$	{ E.type := lookup(id.entry) }
---------------------------	--------------------------------

- The expression formed by applying the **mod operator** to two subexpressions of type integer has type integer; otherwise, its type is **type_error**. The rule is

$E \rightarrow E_1 \text{ mod } E_2$

```
{ E.type := if E1.type = integer and E2.type =  
    integer  
  
    then integer  
  
    else type_error}
```

- In an array reference **E1 [E2]**, the index expression E2 must have type integer, in which case the result is the element type t obtained from the type **array(s, t)** of E1; we make no use of the index set s of the array.

$E \rightarrow E_1 [E_2]$

```
{ E.type := if E2.type = integer and E1.type =  
    array(s,t)  
  
    then t  
  
    else type_error}
```

- Within expressions, the postfix operator \uparrow yields the object pointed to by its operand. The type of $E \uparrow$ is the type \ddagger of the object pointed to by the pointer E:

$E \rightarrow E_1 \ddagger$

```
{ E.type := if E1.type = pointer(t) then t  
  
    else type_error}
```

Type Checking of Statements

- Since language constructs like statements typically do not have values, the special basic type void can be assigned to them.
- If an error is detected within a statement, then the type **type_error** assigned.
- The assignment statement, conditional statement, and while statements are considered for the **type checking**.
- The Sequences of statements are separated by **semicolons**.

Statement	Rule	Semantic Rule
Assignment	$S \rightarrow id := E$	{ $S.type := \text{if } id.type = E.type \text{ then void}$ else type_error }
Conditional	$E \rightarrow \text{if } E \text{ then } S_1$	{ $S.type := \text{if } E.type = \text{Boolean} \text{ then } S_1.type$ else type_error }
While	$E \rightarrow \text{while } E \text{ do } S_1$	{ $S.type := \text{if } E.type = \text{Boolean} \text{ then } S_1.type$ else type_error }
Sequence	$E \rightarrow S_1 ; S_2$	{ $S.type := \text{if } S_1.type = \text{void and } S_2.type = \text{void then void}$ else type_error }

Type Checking of Functions

- The application of a function to an **argument** can be captured by the production

E → E (E)

- An **expression** is the application of one expression to another. The rules for associating type expressions with nonterminal **T** can be augmented by the following production and action to permit function types in declarations.

T → T1' → T2' {T.type := T1.type → T2.type}

- Quotes around the arrow used as a function constructor distinguish it from the arrow used as the **meta** symbol in a production.
- The rule for checking the type of a function application is

E → E1 (E2) { E.type := **if** E2.type = s **and** E1.type = s → t

then t

else typr_error}

TYPE CONVERSIONS

Type Conversions

- Conversion of one data type to another automatically by the compiler is called "Type Conversion".
- For Example, Consider expressions like **x + i**, where **x** is of type float and **i** is of type integer.
- Since the representation of integers and floating-point numbers is different within a computer and different machine instructions are used for operations on integers and floats, the **compiler** may need to convert one of the operands of **+** to ensure that both operands are of the same type when the addition occurs.
- Suppose that integers are converted to floats when necessary, using a **unary** operator (**float**).

For example, the integer 2 is converted to a **float** in the code for the expression `2 * 3.14`:

```
t1 = (float) 2
t2 = t1 * 3.14
```

- The attribute **E.type**, whose value is either integer or float.
- The **rule** associated with $E \rightarrow E_1 + E_2$ builds on the pseudocode

```
if (E1.type = integer and E2.type = integer)      E.type = integer;
else if (E1.type = float and E2. type = integer)   E.type = float;
else if (E1.type = integer and E2. type = float)   E.type = float;
else if (E1.type = float and E2. type = float)     E.type = float;
```

Coercions

- When the type conversion is performed automatically by the **compiler** without the programmer's intervention, the type conversion is referred to as **implicit type conversion**.
- Implicit type conversions, also called **coercions**, are limited in many languages to widening conversions. Conversion is said to be explicit if the programmer must write something to cause the conversion. Explicit conversions are also called casts.

Conversion is said to be **explicit** if the programmer must write something to cause the conversion.

Example

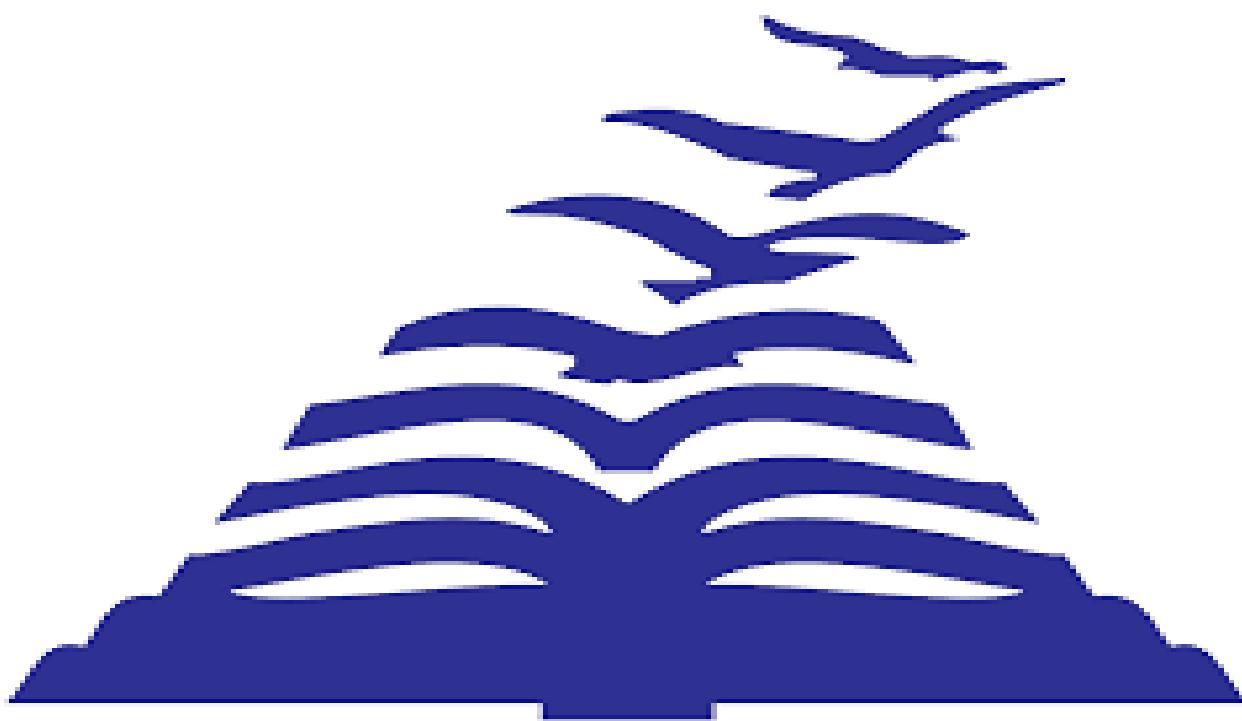
- Consider expressions formed by applying an **arithmetic operator** op to constants and identifiers, as in the grammar. Suppose there are two types - real and integer, with integers converted to reals when necessary. Attribute type of **nonterminal** E can be either integer or real, and the type-checking rules are shown, function `lookup(e)` returns the type saved in the symbol table entry pointed to by e.

Production	Semantic Rule
$E \rightarrow \text{num}$	$E.\text{type} = \text{integer}$
$E \rightarrow \text{num.num}$	$E.\text{type} = \text{real}$
$E \rightarrow \text{id}$	$E.\text{type} = \text{lookup(id.entry)}$
$E \rightarrow E_1 \text{ op } E_2$	$E.\text{type} = \begin{cases} \text{if } (E_1.\text{type} = \text{integer} \text{ and } E_2.\text{type} = \text{integer}) \\ \quad \text{then integer} \\ \text{else if } (E_1.\text{type} = \text{integer} \text{ and } E_2.\text{type} = \text{real}) \\ \quad \text{then real} \\ \text{else if } (E_1.\text{type} = \text{real} \text{ and } E_2.\text{type} = \text{integer}) \\ \quad \text{then real} \\ \text{else if } (E_1.\text{type} = \text{real} \text{ and } E_2.\text{type} = \text{real}) \\ \quad \text{then real} \\ \text{else type_error} \end{cases}$



THANK

YOU



Presidency University, Bengaluru

Intermediate Languages - **THREE ADDRESS STATEMENTS**

Intermediate Code Generation



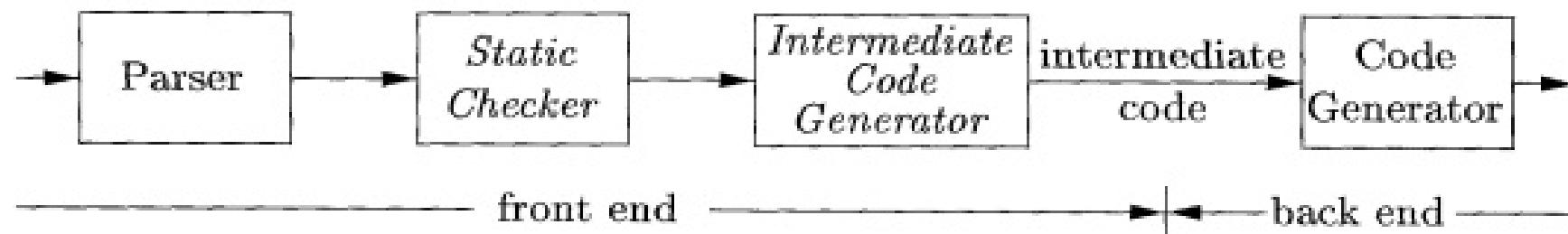
- After syntax and semantic analysis of the source program, many compilers generate an explicit low-level or machine-like intermediate representation.
- In the process of translating a source program into target code, a compiler may construct one or more intermediate representations, which can have a variety of forms.
- This intermediate representation should have two important properties:
 - It should be easy to produce and
 - It should be easy to translate into the target machine

Intermediate Code Generation (ICG)

In the analysis-synthesis model of a compiler, the front end analyzes a source program and creates an intermediate representation, from which the back end generates target code.

Three ways of intermediate representation:

- Syntax tree**
- Postfix notation**
- Three address code**



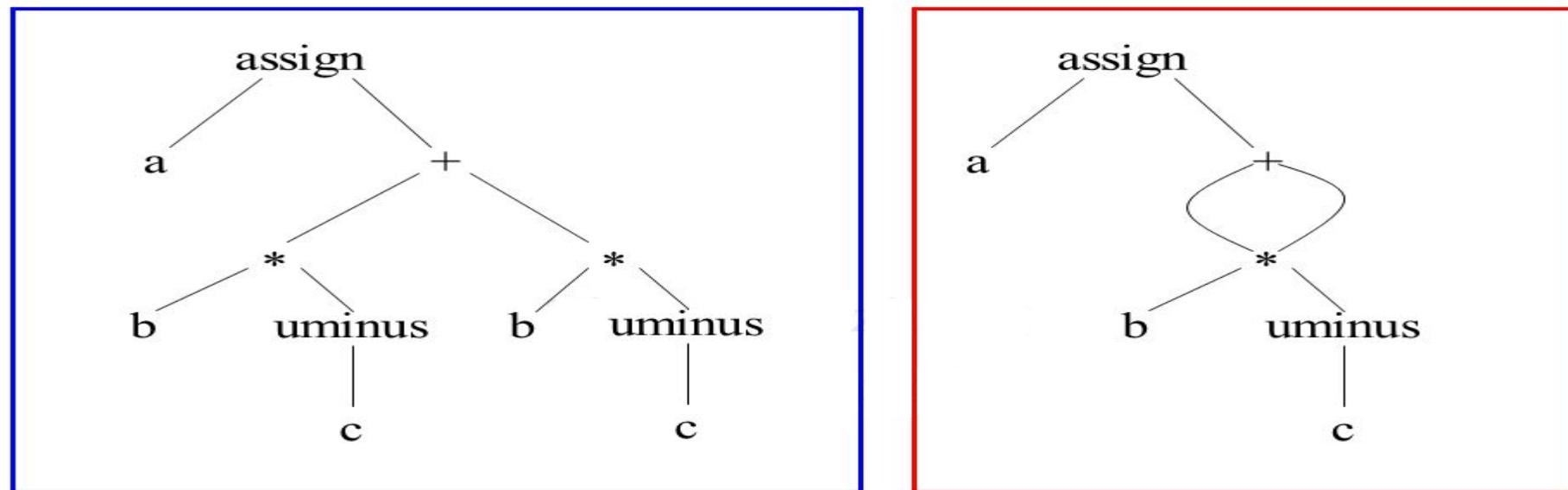
Logical structure of a compiler front end

Intermediate Code Generation (ICG)

Graphical Representations: Syntax Tree

A syntax tree depicts the natural hierarchical structure of a source program. A DAG (Directed Acyclic Graph) gives the same information but in a more compact way because common subexpressions are identified.

For example, constructs the syntax tree and DAG from the input **a := b * - c + b* - c**



Postfix Notation:

Postfix notation is a linearized representation of a syntax tree; it is a list of the nodes of the tree in which a node appears immediately after its children.

The postfix notation for the syntax tree given above is

a b c uminus * b c uminus * + assign

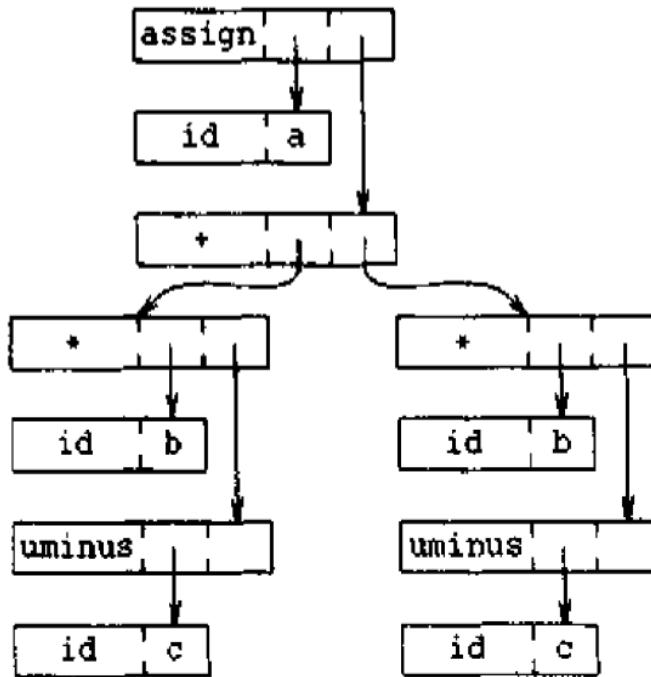
Intermediate Code Generation (ICG)

Syntax-directed definition to produce syntax tree of assignment statements

production	semantic rules
$S \rightarrow id := E$	$S.nptr := mknode('assign', mkleaf(id, id.entry), E.nptr)$
$E \rightarrow E_1 + E_2$	$E.nptr := mknode('+', E_1.nptr, E_2.nptr)$
$E \rightarrow E_1 * E_2$	$E.nptr := mknode('*', E_1.nptr, E_2.nptr)$
$E \rightarrow -E_1$	$E.nptr := mknnode('uminus', E_1.nptr)$
$E \rightarrow (E_1)$	$E.nptr := E_1.nptr$
$E \rightarrow id$	$E.nptr := mkleaf(id, id.entry)$

Two representations of the syntax tree are as follows.

- In (a) each node is represented as a record with a field for its operator and additional fields for pointers to its children.
- In (b), nodes are allocated from an array of records and the index or position of the node serves as the pointer to the node.



0	id	b
1	id	c
2	uminus	1
3	*	0 ?
4	id	b
5	id	c
6	uminus	5
7	*	4 6
8	+	3 7
9	id	a
10	assign	9 8
11		

Three-address code:

One of the mostly used intermediate form called **three-address code**, which **consists of a sequence of assembly-like instructions with three operands per instruction.**

Three-address code is a sequence of statements of the general form **$x := y \text{ op } z$**

where **x, y and z are names, constants, or compiler-generated temporaries;**
op stands for any operator, such as a fixed - or floating-point arithmetic operator, or a logical operator on boolean-valued data.

Thus a source language expression like **$x + y * z$** might be translated into a sequence.

t1 = y * z

t2 = x + t1

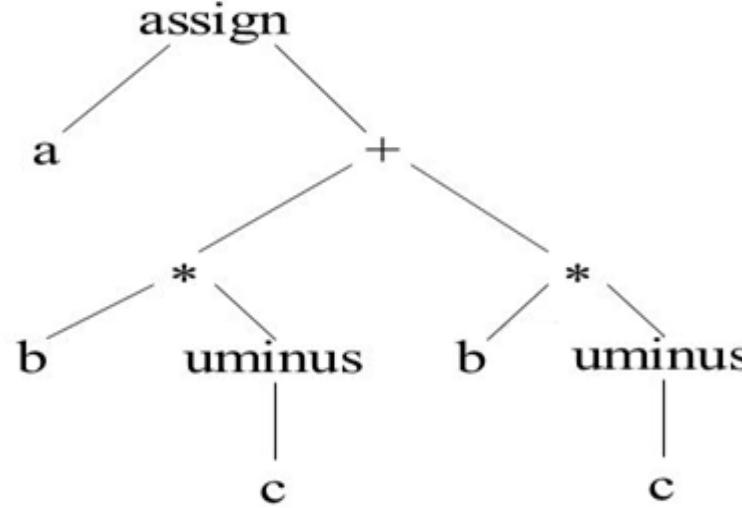
where **t1 and t2 are compiler-generated temporary names.**

Three-address instruction has several important properties:

- **First**, each three-address assignment instruction has at most one operator on the right side.
- **Second**, the compiler must generate a temporary name to hold the value computed by a three-address instruction.
- **Third**, some three-address instructions have fewer than three operands.

Three Address Code

Three-address code corresponding to the Syntax Tree and DAG



```
t1 = minus c
t2 = b * t1
t3 = minus c
t4 = b * t3
t5 = t2 + t4
a = t5
```

```
t1 := -c
t2 := b * t1
t5 := t2 + t2
a := t5
```

The **reason** for the term “**three-address code**” is that each statement usually **contains three addresses**, two for the operands and one for the result.

Types of Three-Address Statements:

1. **Assignment statements** of the form **$x := y \text{ op } z$** , where op is a binary arithmetic or logical operation.
2. **Assignment instructions** of the form **$x := \text{op } y$** , where op is a unary operation. Essential unary operations include unary minus, logical negation, shift operators, and conversion operators.
3. **Copy statements** of the form **$x := y$** where the value of y is assigned to x.
4. The **unconditional jump goto L**. The three-address statement with label L is the next to be executed.
5. **Conditional jumps** such as **$\text{if } x \text{ relop } y \text{ goto L}$** .
6. **param x and call p, n for procedure calls and return y**, where y representing a returned value is optional.

For example,

```
param x1
param x2
param xn
cal 1 p, n
```

7. **Indexed assignments** of the form **$x := y[i]$ and $x[i] := y$** .
8. **Address and pointer assignments** of the form **$x := &y$, $x := *y$ and $*x := y$** .

Implementation of Three-Address Statements

A three-address statement is an **abstract form of intermediate code**. In a compiler, these statements can be implemented as **records with fields for the operator and the operands**.

Three such representations are:

- ❖ **Quadruples**
- ❖ **Triples**
- ❖ **Indirect Triples**

Three Address Code - Implementation

Quadruples

- A **quadruple is a record structure with four fields**, which are, **op, arg1, arg2 and result**.
- The op field contains an internal code for the operator. The three-address statement $x := y \text{ op } z$ is represented by placing y in arg1, z in arg2 and x in result.
- The contents of fields **arg1, arg2 and result are normally pointers to the symbol-entries** for the names represented by these fields. If so, temporary names must be entered into the symbol table as they are created.

	<i>op</i>	<i>arg 1</i>	<i>arg 2</i>	<i>result</i>
(0)	uminus	c		t ₁
(1)	*	b	t ₁	t ₂
(2)	uminus	c		t ₃
(3)	*	b	t ₃	t ₄
(4)	+	t ₂	t ₄	t ₅
(5)	:=	t ₅		a

Three Address Code - Implementation

Triples

- To **avoid entering temporary names into the symbol table**, we might refer to a temporary value by the position of the statement that computes it.
- If we do so, three-address statements can be represented by **records with only three fields: op, arg1 and arg2**.
- The fields arg1 and arg2, for the arguments of op, are either pointers to the symbol table or pointers into the triple structure (for temporary values).
- Since **three fields are used**, this intermediate code **format is known as triples**.

	<i>op</i>	<i>arg 1</i>	<i>arg 2</i>
(14)	uminus	c	
(15)	*	b	(14)
(16)	uminus	c	
(17)	*	b	(16)
(18)	+	(15)	(17)
(19)	assign	a	(18)

Three Address Code - Implementation

Indirect Triples

Another implementation of three-address code is that of **listing pointers to triples**, rather than listing the triples themselves. This implementation is called indirect triples.

	<i>statement</i>
(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)
(4)	(18)
(5)	(19)

Three Address Code - Implementation

Triple Structure for the following statements are

x[i] := y

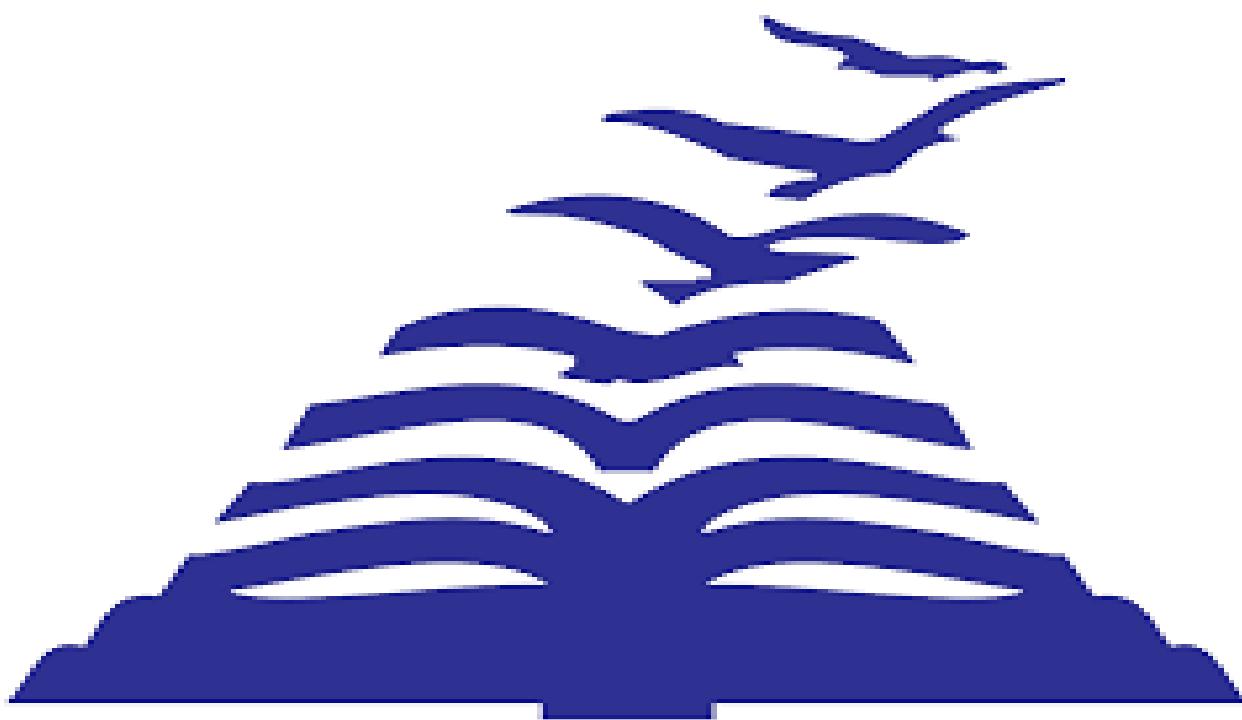
x := y[i]

	OP	Arg1	Arg2
(0)	[]=	x	i
(1)	assign	(0)	y

	OP	Arg1	Arg2
(0)	=[]	y	i
(1)	assign	x	(0)

Write Three Address Code for the following expression-

$$a = b + c + d$$



Presidency University, Bengaluru

TRANSLATION OF DECLARATIONS

Translation of Declarations



- As the **sequence of declarations in a procedure or block** is examined, we can lay out **storage for names local to the procedure**.
- For each **local name**, we create a **symbol-table entry with information** like the **type and the relative address** of the storage for the name.
- The **relative address** consists of an offset from the base of the static data area or the field for local data in an activation record.

Translation of Declarations

Declarations in a Procedure:

The syntax of languages such as C, Pascal and Fortran, allows all the declarations in a single procedure to be processed as a group. In this case, a global variable, say offset, can keep track of the next available relative address.

- ❖ **Nonterminal P** generates a sequence of declarations of the **form id : T.**
- ❖ Before the first declaration is considered, **offset is set to 0**. As each new name is seen, that name is entered in the symbol table with offset equal to the current value of offset, and offset is incremented by the width of the data object denoted by that name.
- ❖ The **procedure enter(name, type, offset)** creates a symbol-table entry for name, gives its type type and relative address offset in its data area.
- ❖ **Attribute type represents a type expression** constructed from the basic types integer and real by applying the type constructors pointer and array. If type expressions are represented by graphs, then attribute type might be a pointer to the node representing a type expression.
- ❖ The **width of an array is obtained by multiplying the width of each element by the number of elements in the array.** The width of each pointer is assumed to be 4.



Translation of Declarations

P → D	{ offset := 0 }
D → D ; D	
D → id : T	{ enter(id.name, T.type, offset); offset := offset + T.width }
T → integer	{ T.type := integer; T.width := 4 }
T → real	{ T.type := real; T.width := 8 }
T → array [num] of T1	{ T.type := array(num.val, T1.type); T.width := num.val X T1.width }
T → ↑T1	{ T.type := pointer (T1.type); T.width := 4 }

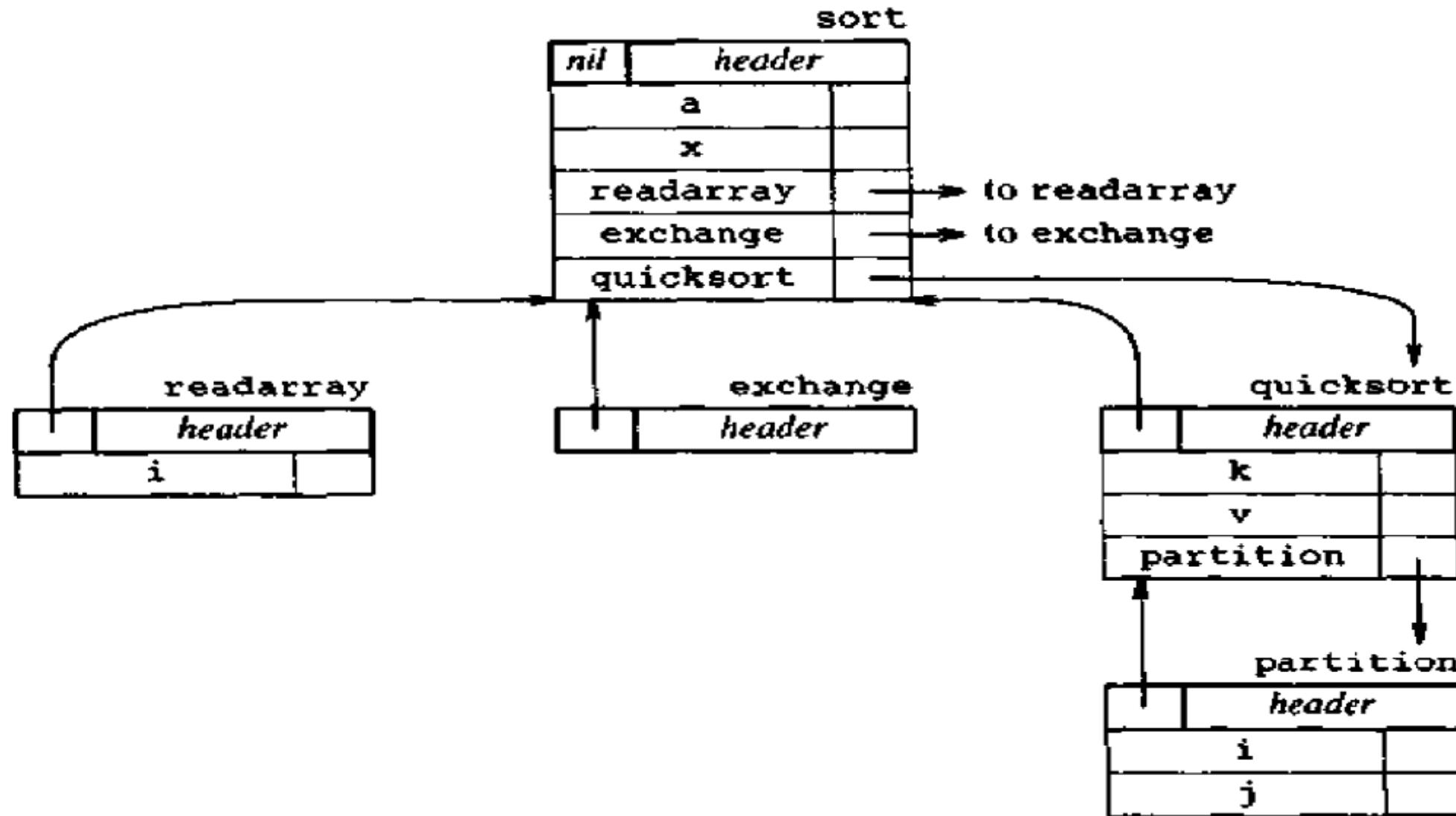
Translation of Declarations

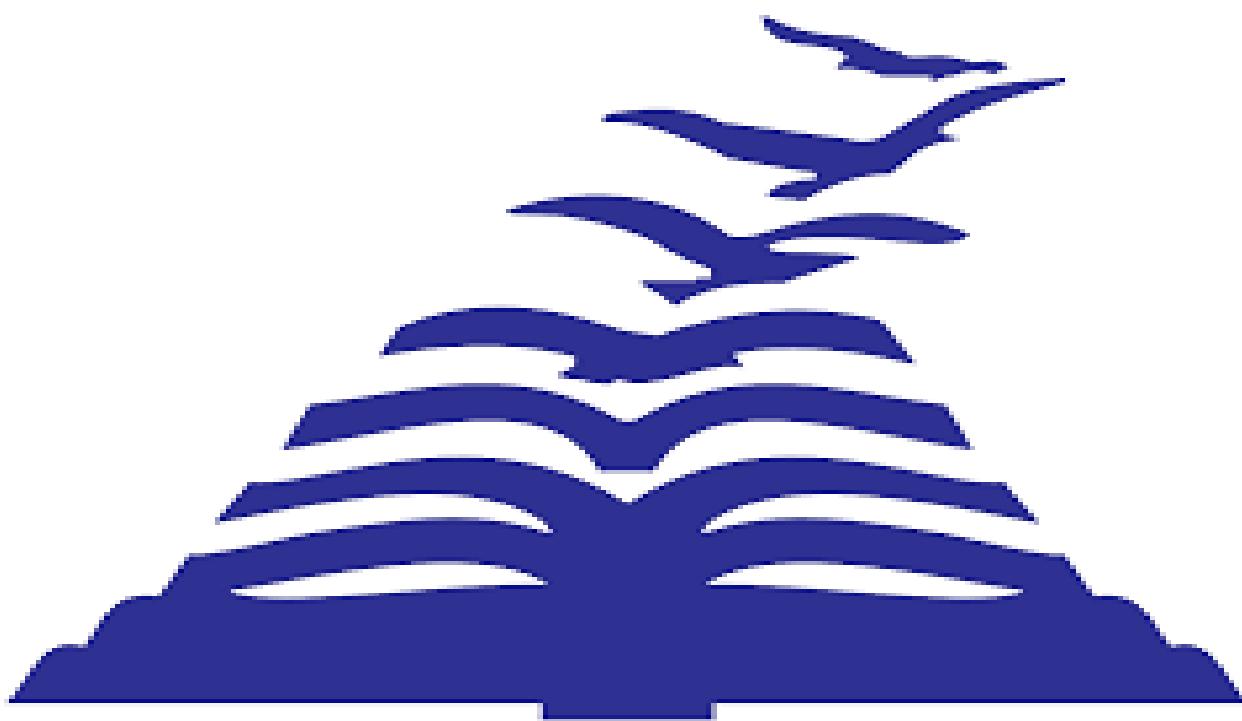
The semantic rules are defined in terms of the following operations:

1. **mktable(previous)** creates a new symbol table and returns a pointer to the new table. The argument previous points to a previously created symbol table, presumably that for the enclosing procedure.
2. **enter(table, name, type, offset)** creates a new entry for name name in the symbol table pointed to by table. Again, enter places type type and relative address offset in fields within the entry.
3. **addwidth(table, width)** records the cumulative width of all the entries in table in the header associated with this symbol table.
4. **enterproc(table, name, newtable)** creates a new entry for procedure name in the symbol table pointed to by table. The argument newtable points to the symbol table for this procedure name.



Translation of Declarations





Presidency University, Bengaluru

TRANSLATION OF ASSIGNMENT STATEMENTS

Translation of Assignment Statements

Translation scheme to produce three-address code for assignments

S → id := E { p := lookup (id.name); if p ≠ nil then emit(p ‘:=’ E.place) else error }

E → E1 + E2 { E.place := newtemp; emit(E.place ‘:=’ E1.place ‘+’ E2.place) }

E → E1 * E2 { E.place := newtemp; emit(E.place ‘:=’ E1.place ‘*’ E2.place) }

E → -E1 { E.place := newtemp; emit (E.place ‘:=’ ‘uminus’ E1.place) }

E → (E1) { E.place := E1.place }

E → id { p := lookup (id.name); if p ≠ nil then E.place := p else error }

Translation of Assignment Statements

Reusing Temporary Names

- The **temporaries used to hold intermediate values in expression calculations** and space has to be allocated to hold their values.
 - Temporaries can be reused by changing newtemp.
 - The code generated by the rules for $E = E1 + E2$ has the general form:
evaluate E1 into t1
evaluate E2 into t2
t := t1 + t2
- * The lifetimes of these temporaries are nested like matching pairs of balanced parentheses.
- * Keep a count c, initialized to zero. Whenever a temporary name is used as an operand, decrement c by 1. Whenever a new temporary name is generated, use \$c and increase c by 1.

Translation of Assignment Statements

Addressing Array Elements:

Elements of an array can be accessed quickly if the elements are stored in a block of consecutive locations. If the **width of each array element is w**, then the i^{th} element of array A begins in location

$$\text{base} + (i - \text{low}) * w$$

where **low is the lower bound** on the subscript and **base is the relative address of the storage allocated for the array**. That is, **base is the relative address of A[low]**.

The expression can be partially evaluated at compile time if it is rewritten

$$i * w + (\text{base} - \text{low} * w)$$

The subexpression **$c = base - low * w$ can be evaluated** when the declaration of the array is seen. We assume that c is saved in the symbol table entry for A, so the relative address of A[i] is obtained by simply adding **$i * w$ to c**.

Translation of Assignment Statements



Address calculation of multi-dimensional arrays:

A two-dimensional array is stored in one of the two forms :

- **Row-major (row-by-row)**
- **Column-major (column-by-column)**

In the case of **row-major form**, the relative address of $A[i_1, i_2]$ can be calculated by the formula

$$\text{base} + ((i_1 - \text{low}_1) \times n_2 + i_2 - \text{low}_2) * w$$

where, **low1 and low2 are the lower bounds** on the values of i_1 and i_2 and **n2 is the number of values that i_2 can take**. That is, if **high2 is the upper bound on the value of i_2** , then

$$n_2 = \text{high}_2 - \text{low}_2 + 1.$$

Assuming that i_1 and i_2 are the only values that are known at compile time, we can rewrite the above expression as

$$((i_1 * n_2) + i_2) * w + (\text{base} - ((\text{low}_1 * n_2) + \text{low}_2) * w)$$

Translation of Assignment Statements

Type conversion within Assignments :

Semantic action for $E \rightarrow E_1 + E_2$

$E.place := newtemp;$

```
if  $E1.type = integer$  and  $E2.type = integer$  then begin emit(  $E.place := E1.place$  'int +'  $E2.place$ );
 $E.type := integer$ 
End
```

```
else if  $E1.type = real$  and  $E2.type = real$  then begin emit(  $E.place := E1.place$  'real +'  $E2.place$ );
 $E.type := real$ 
End
```

```
else if  $E1.type = integer$  and  $E2.type = real$  then begin  $u := newtemp$ ; emit(  $u :=$  'inttoreal'  $E1.place$ );
emit(  $E.place := u$  'real +'  $E2.place$ );  $E.type := real$ 
End
```

```
else if  $E1.type = real$  and  $E2.type = integer$  then begin  $u := newtemp$ ; emit(  $u :=$  'inttoreal'  $E2.place$ );
emit(  $E.place := E1.place$  'real +'  $u$ );  $E.type := real$ 
End
```

else

$E.type := type_error;$

$if (E1.type = integer \text{ and } E2.type = integer)$ $else if (E1.type = float \text{ and } E2.type = integer)$ $else if (E1.type = integer \text{ and } E2.type = float)$ $else if (E1.type = float \text{ and } E2.type = float)$	$E.type = integer;$ $E.type = float;$ $E.type = float;$ $E.type = float;$
---	--

Translation of Assignment Statements



Type conversion within Assignments :

For example, for the input **x := y + i * j**

assuming **x and y have type real**, and **i and j have type integer**, the output would look like

t1 := i int* j

t3 := inttoreal (t1)

t2 := y real+ t3

x := t2



Presidency University, Bengaluru

TRANSLATION OF BOOLEAN EXPRESSION

Translation of Boolean Expression

Boolean expressions have two primary purposes.

- They are used to **compute logical values**,
 - but more often they are used as **conditional expressions** in statements that alter the flow of control, such as if-then-else, or while-do statements.
- Boolean expressions are composed of the boolean operators (and, or, and not) applied to elements that are boolean variables or relational expressions.
- Relational expressions are of the form **E1 relop E2**, where E1 and E2 are arithmetic expressions.

Here we consider boolean expressions generated by the following grammar :

$$E \rightarrow E \text{ or } E \mid E \text{ and } E \mid \text{not}E \mid (E) \mid \text{id} \text{ relop id} \mid \text{true} \mid \text{false}$$

Translation of Boolean Expression

Methods of Translating Boolean Expressions:

There are two principal methods of representing the value of a boolean expression.

They are :

- To encode **true** and **false** numerically and to evaluate a boolean expression analogously to an arithmetic expression. Often, **1 is used to denote true and 0 to denote false.**
(Non-Zero quantity denotes true and Zero denotes false, positive values denotes true and negative values denotes false)
- To implement boolean expressions by flow of control, that is, representing the value of a boolean expression by a position reached in a program. This method is particularly convenient in implementing the boolean expressions in flow-of-control statements, such as the if-then and while-do statements.

For example, given the expression **E1 or E2**, if we determine that **E1 is true, then we can conclude that the entire expression is true without having to evaluate E2.**

Translation of Boolean Expression



Numerical Representation

The translation for **a or b and not c** is the three-address sequence

```
t1 := not c
t2 := b and t1
t3 := a or t2
```

A relational expression such as $a < b$ is equivalent to the conditional statement

```
if a < b then 1 else 0
```

which can be translated into the three-address code sequence :

```
100 : if a < b goto 103
101 : t := 0
102 : goto 104
103 : t := 1
104 :
```



Translation of Boolean Expression

Translation scheme using a numerical representation for Boolean

$E \rightarrow E_1 \text{ or } E_2$	{ E.place := newtemp; emit(E.place ‘:=’ E1.place ‘or’ E2.place) }
$E \rightarrow E_1 \text{ and } E_2$	{ E.place := newtemp; emit(E.place ‘:=’ E1.place ‘and’ E2.place) }
$E \rightarrow \text{not } E_1$	{ E.place := newtemp; emit(E.place ‘:=’ ‘not’ E1.place) }
$E \rightarrow (E_1)$	{ E.place := E1.place }
$E \rightarrow id_1 \text{ relop } id_2$	{ E.place := newtemp; emit(‘if’ id1.place relop.op id2.place ‘goto’ nextstat + 3); emit(E.place ‘:=’ ‘0’); emit(‘goto’ nextstat + 2); emit(E.place ‘:=’ ‘1’) }
$E \rightarrow \text{true}$	{ E.place := newtemp; emit(E.place ‘:=’ ‘1’) }
$E \rightarrow \text{false}$	{ E.place := newtemp; emit(E.place ‘:=’ ‘0’) }



Translation of Boolean Expression

Short-Circuit Code:

- We can also translate a boolean expression into three-address code **without generating code for any of the boolean operators (and, or, and not)** and without having the code necessarily evaluate the entire expression. This style of evaluation is sometimes called “**short-circuit**” or “**jumping**” code.

Translation of $a < b \text{ or } c < d \text{ and } e < f$

100: if a < b goto 103	107: t ₂ := 1
101: t ₁ := 0	108: if e < f goto 111
102: goto 104	109: t ₃ := 0
103: t ₁ := 1	110: goto 112
104: if c < d goto 107	111: t ₃ := 1
105: t ₂ := 0	112: t ₄ := t ₂ and t ₃
106: goto 108	113: t ₅ := t ₁ or t ₄



Flow-of-Control Statements

Consider the translation of boolean expressions into **three address code in the context of if-then, if-then-else, and while-do statements** such as those generated by the following grammar:

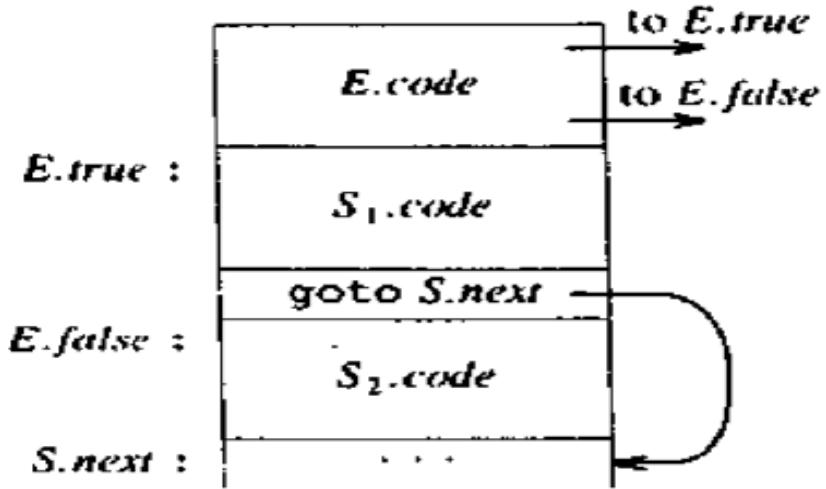
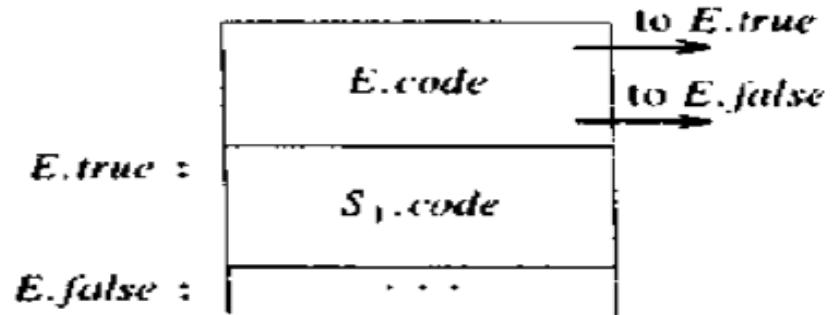
```
S → if E then S1
  | if E then S1 else S2
  | while E do S1
```

In each of these productions, **E is the Boolean expression** to be translated. In the translation, **we assume that a three-address statement can be symbolically labeled, and that the function newlabel returns a new symbolic label each time it is called.**

- **E.true is the label to which control flows if E is true, and E.false is the label to which control flows if E is false.**
- **The semantic rules for translating a flow-of-control statement S allow control to flow from the translation S.code to the three-address instruction immediately following S.code.**
- **S.next is a label that is attached to the first three-address instruction to be executed after the code for S.**

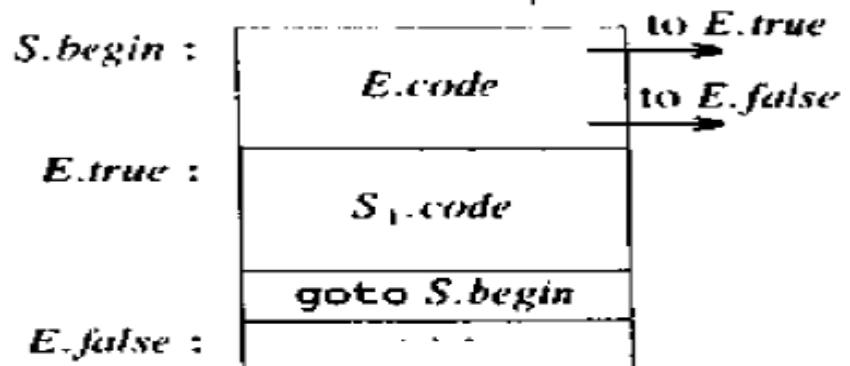


Translation of Boolean Expression



(a) if-then

(b) if-then-else



(c) while-do



Translation of Boolean Expression

Syntax Directed Definition For Flow-of-Control Statements

PRODUCTION	SEMANTIC RULES
$S \rightarrow \text{if } E \text{ then } S_1$	$E.\text{true} := \text{newlabel};$ $E.\text{false} := S.\text{next};$ $S_1.\text{next} := S.\text{next};$ $S.\text{code} := E.\text{code} \parallel$ $\quad \text{gen}(E.\text{true} ':') \parallel S_1.\text{code}$
$S \rightarrow \text{if } E \text{ then } S_1 \text{ else } S_2$	$E.\text{true} := \text{newlabel};$ $E.\text{false} := \text{newlabel};$ $S_1.\text{next} := S.\text{next};$ $S_2.\text{next} := S.\text{next};$ $S.\text{code} := E.\text{code} \parallel$ $\quad \text{gen}(E.\text{true} ':') \parallel S_1.\text{code} \parallel$ $\quad \text{gen}(\text{'goto' } S.\text{next}) \parallel$ $\quad \text{gen}(E.\text{false} ':') \parallel S_2.\text{code}$
$S \rightarrow \text{while } E \text{ do } S_1$	$S.\text{begin} := \text{newlabel};$ $E.\text{true} := \text{newlabel};$ $E.\text{false} := S.\text{next};$ $S_1.\text{next} := S.\text{begin};$ $S.\text{code} := \text{gen}(S.\text{begin} ':') \parallel E.\text{code} \parallel$ $\quad \text{gen}(E.\text{true} ':') \parallel S_1.\text{code} \parallel$ $\quad \text{gen}(\text{'goto' } S.\text{begin})$



Translation of Boolean Expression

PRODUCTION	SEMANTIC RULES
$E \rightarrow E_1 \text{ or } E_2$	$E_1.\text{true} := E.\text{true};$ $E_1.\text{false} := \text{newlabel};$ $E_2.\text{true} := E.\text{true};$ $E_2.\text{false} := E.\text{false};$ $E.\text{code} := E_1.\text{code} \parallel \text{gen}(E_1.\text{false}':') \parallel E_2.\text{code}$
$E \rightarrow E_1 \text{ and } E_2$	$E_1.\text{true} := \text{newlabel};$ $E_1.\text{false} := E.\text{false};$ $E_2.\text{true} := E.\text{true};$ $E_2.\text{false} := E.\text{false};$ $E.\text{code} := E_1.\text{code} \parallel \text{gen}(E_1.\text{true}':') \parallel E_2.\text{code}$
$E \rightarrow \text{not } E_1$	$E_1.\text{true} := E.\text{false};$ $E_1.\text{false} := E.\text{true};$ $E.\text{code} := E_1.\text{code}$
$E \rightarrow (E_1)$	$E_1.\text{true} := E.\text{true};$ $E_1.\text{false} := E.\text{false};$ $E.\text{code} := E_1.\text{code}$
$E \rightarrow \text{id}_1 \text{ relop id}_2$	$E.\text{code} := \text{gen('if' id}_1.\text{place relop.op id}_2.\text{place 'goto' } E.\text{true}) \parallel \text{gen('goto' } E.\text{false)}$
$E \rightarrow \text{true}$	$E.\text{code} := \text{gen('goto' } E.\text{true})$
$E \rightarrow \text{false}$	$E.\text{code} := \text{gen('goto' } E.\text{false})$



Translation of Boolean Expression

Let us again consider the expression

$a < b \text{ or } c < d \text{ and } e < f$

```

if a < b goto Ltrue
goto L1
L1: if c < d goto L2
    goto Lfalse
L2: if e < f goto Ltrue
    goto Lfalse
  
```

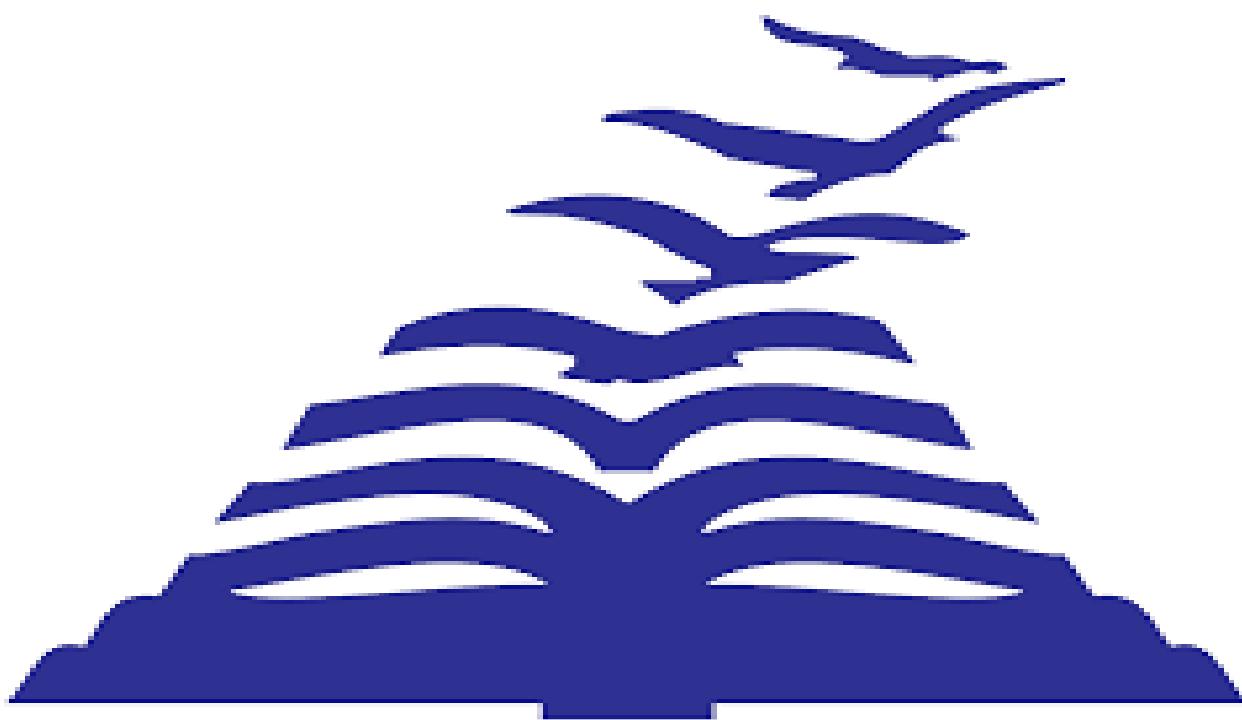
Consider the statement

```

while a < b do
  if c < d then
    x := y + z
  else
    x := y - z
  
```

```

L1: if a < b goto L2
    goto Lnext
L2: if c < d goto L3
    goto L4
L3: t1 := y + z
    x := t1
    goto L1
L4: t2 := y - z
    x := t2
    goto L1
Lnext:
  
```



Presidency University, Bengaluru

Basic Blocks and Flow Graphs

Basic Blocks

A basic block is a **sequence of consecutive statements** in which flow of control **enters at the beginning and leaves at the end without any halt or possibility of branching except at the end.**

The following sequence of three-address statements forms a basic block

t1 := a * a

t2 := a * b

t3 := 2 * t2



Basic Block Construction:

Algorithm: Partition into basic blocks

Input: A sequence of three-address statements

Output: A list of basic blocks with each three-address statement in exactly one block

Method:

1. We first determine the set of leaders, the first statements of basic blocks. The rules we use are of the following:
 - The first statement is a leader.
 - Any statement that is the target of a conditional or unconditional goto is a leader.
 - Any statement that immediately follows a goto or conditional goto statement is a leader.
1. For each leader, its basic block consists of the leader and all statements up to but not including the next leader or the end of the program.

Basic Blocks and Flow Graphs



Construct the Basic Blocks for the following statements

I = 1

While (I < 10)

{

 a = b[i] + c;
 i++;

}

Result = a + c

WSTART: **I = 1**
 if (I < 10) then goto TRUE
 goto EXIT

TRUE: **T1 = b[i]**
 T2 = T1 + c
 a = T2
 T3 = I + 1
 I = T3
 goto WSTART

EXIT: **Result = a + c**

L1: **I = 1**

L2: **if (I < 10) then goto TRUE**

L3: **goto EXIT**

L4: **T1 = b[i]**
 T2 = T1 + c
 a = T2
 T3 = I + 1
 I = T3
 goto WSTART

L5: **Result = a + c**

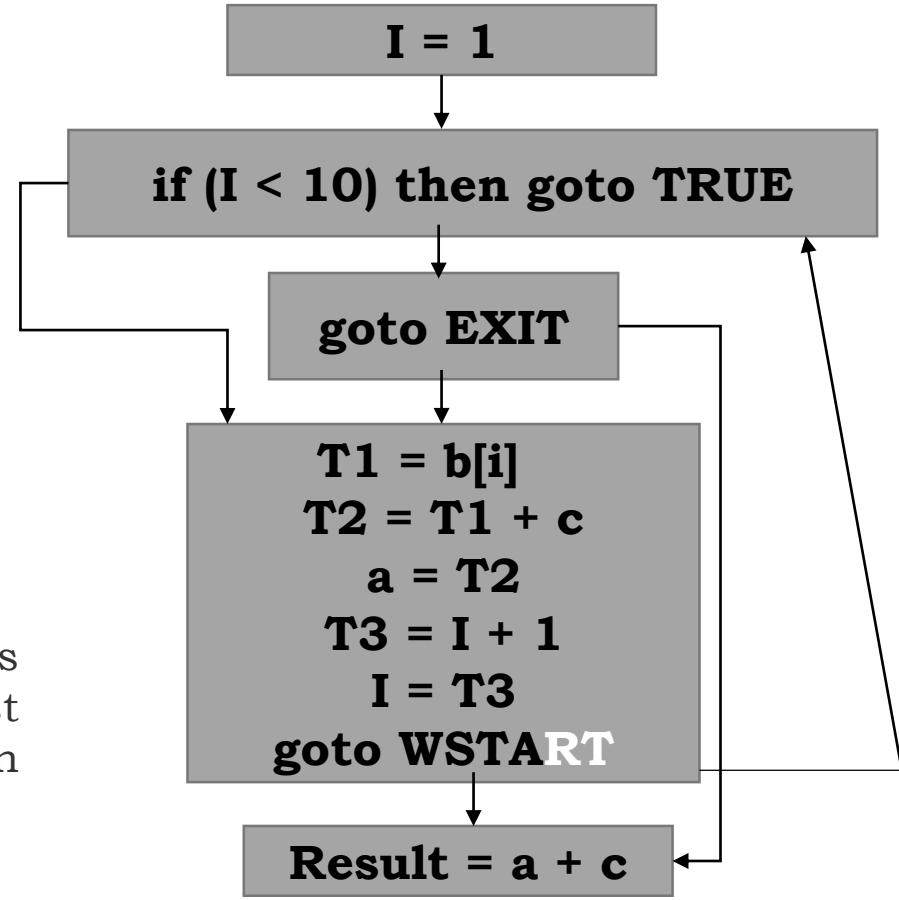
Basic Blocks and Flow Graphs

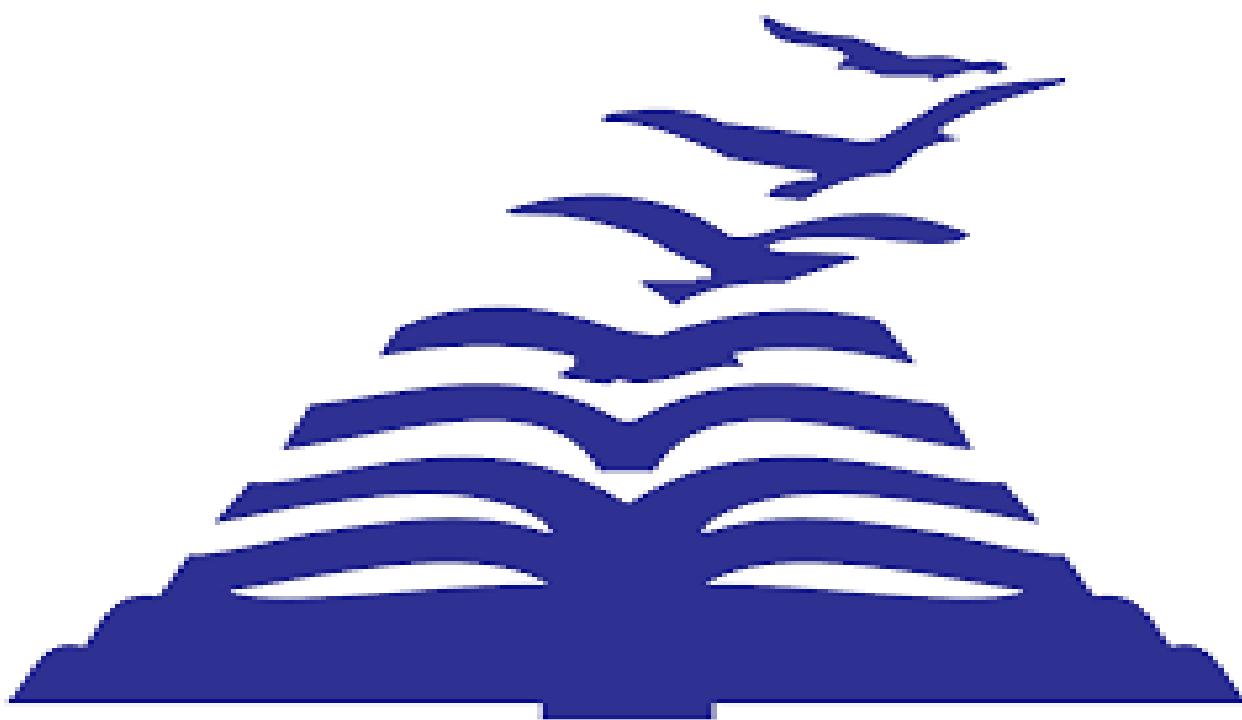
Flow Graphs

Flow graph is a directed graph containing the flow-of-control information for the set of basic blocks making up a program. The nodes of the flow graph are basic blocks.

```
L1:      I = 1
L2:      if (I < 10) then goto TRUE
L3:      goto EXIT
L4:      T1 = b[i]
         T2 = T1 + c
         a = T2
         T3 = I + 1
         I = T3
         goto WSTART
Result = a + c
```

- B1 is the initial node. B2 immediately follows B1, so there is an edge from B1 to B2. The target of jump from last statement of B1 is the first statement B2, so there is an edge from B1 (last statement) to B2 (first statement).
- B1 is the predecessor of B2, and B2 is a successor of B1.





Presidency University, Bengaluru

Principal Sources of OPTIMIZATION

Code Optimization



The **code produced** by the straight forward compiling algorithms can often be made to **run faster or take less space, or both**. This **improvement is achieved by program transformations** that are traditionally called **optimizations**.

Compilers that apply code-improving transformations are called **optimizing compilers**.



The criteria for code improvement transformations

Simply stated, the best program transformations are those that **yield the most benefit for the least effort.**

The transformations provided by an optimizing compiler should have several properties.

They are:

- The transformation must preserve the meaning of programs.
- A transformation must, on the average, speedup programs by a measurable amount.
- The transformation must be worth the effort.

- A transformation of a program is called local if it can be performed by looking only at the statements in a basic block; otherwise, it is called global.
- Many transformations can be performed at both the local and global levels.
- Local transformations are usually performed first.

Function-Preserving Transformations

There are a number of ways in which a compiler can improve a program without changing the function it computes.

Function preserving transformations examples:

- Common sub expression elimination
- Copy propagation
- Dead-code elimination
- Constant folding

Common Sub expressions elimination:

- An occurrence of an expression E is called a common sub-expression if E was previously computed, and the values of variables in E have not changed since the previous computation.
- We can avoid recomputing the expression if we can use the previously computed value.

Copy Propagation:

- Assignments of the form $f := g$ called copy statements, or copies for short. The idea behind the copy-propagation transformation is to use g for f , whenever possible after the copy statement $f := g$.
- Copy propagation means use of one variable instead of another.
- This may not appear to be an improvement, but as we shall see it gives us an opportunity to eliminate f .

Principal Sources of Optimization



Examples for Common Sub expressions elimination and Copy Propagation

Principal Sources of Optimization



Dead-Code Eliminations:

A variable is live at a point in a program if its value can be used subsequently; otherwise, it is dead at that point. A related idea is dead or useless code, statements that compute values that never get used.

Constant folding:

Deducing at compile time that the value of an expression is a constant and using the constant instead is known as constant folding.

Principal Sources of Optimization



Examples for Dead-Code Eliminations and Constant folding:



Loop Optimizations:

In loops, especially in the inner loops, programs tend to spend the bulk of their time. The running time of a program may be improved if the number of instructions in an inner loop is decreased, even if we increase the amount of code outside that loop.

Three techniques are important for loop optimization:

- **Code motion**, which moves code outside a loop;
- **Induction-variable elimination**, which we apply to replace variables from inner loop.
- **Reduction in strength**, which replaces an expensive operation by a cheaper one, such as a multiplication by an addition.

Principal Sources of Optimization



Code Motion:

An important modification that decreases the amount of code in a loop is code motion. This transformation takes an expression that yields the same result independent of the number of times a loop is executed (a loop-invariant computation) and places the expression before the loop.

```
while (i <= limit-2) /* statement does not change limit*/
```

Code motion will result in the equivalent of

```
t= limit-2;
while (i<=t)
```

Principal Sources of Optimization



Induction Variables :

An **induction variable** is a variable that gets increased or decreased by a fixed amount on every iteration of a loop or is a linear function of another induction variable.

A common compiler optimization is to recognize the existence of induction variables and replace them with simpler computations

For example, in the following loop, i and j are induction variables.

```
for (i = 0; i < 10; ++i) {  
    j = 17 * i;  
}
```

```
j = -17;  
for (i = 0; i < 10; ++i) {  
    j = j + 17;  
}
```

Principal Sources of Optimization



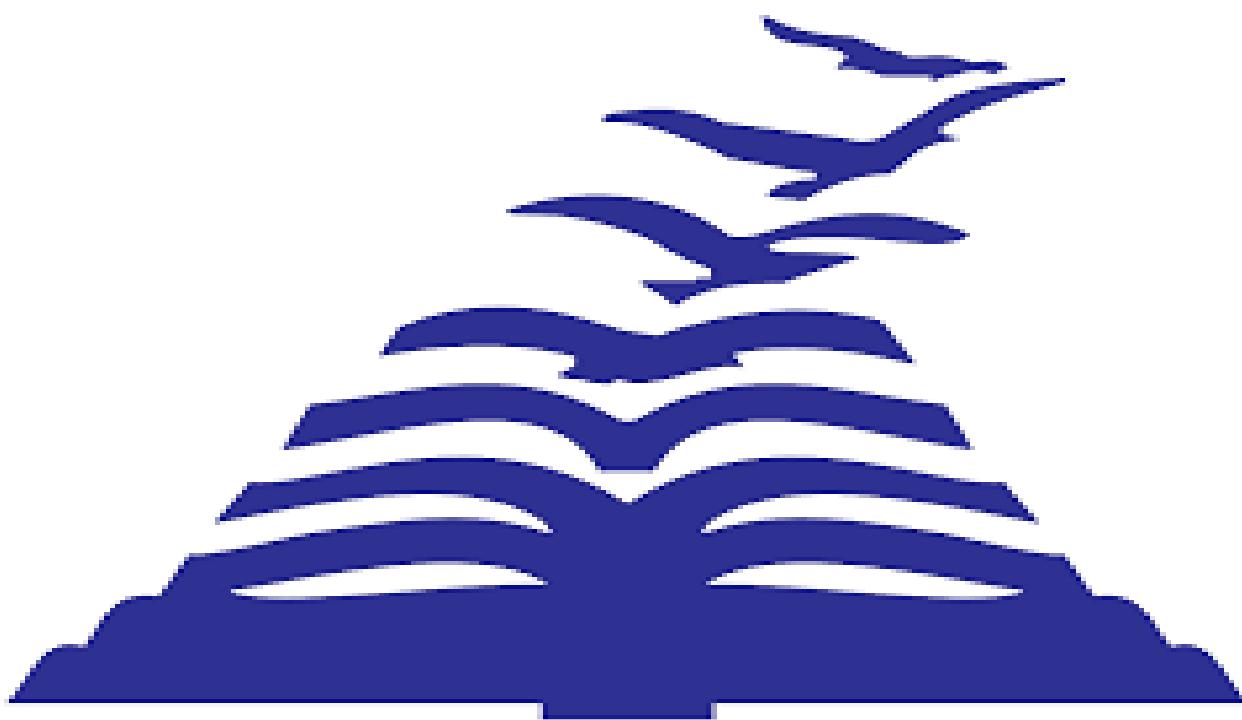
Reduction In Strength:

Reduction in strength replaces expensive operations by equivalent cheaper ones on the target machine.

Certain machine instructions are considerably cheaper than others and can often be used as special cases of more expensive operators.

For example, **x^2 is invariably cheaper to implement as $x * x$ than as a call to an exponentiation routine.**

$2 * X$ is replaced by $X + X$



Presidency University, Bengaluru

Principal Sources of OPTIMIZATION

Code Optimization



The **code produced** by the straight forward compiling algorithms can often be made to **run faster or take less space, or both**. This **improvement is achieved by program transformations** that are traditionally called **optimizations**.

Compilers that apply code-improving transformations are called **optimizing compilers**.



The criteria for code improvement transformations

Simply stated, the best program transformations are those that **yield the most benefit for the least effort.**

The transformations provided by an optimizing compiler should have several properties.

They are:

- The transformation must preserve the meaning of programs.
- A transformation must, on the average, speedup programs by a measurable amount.
- The transformation must be worth the effort.

- A transformation of a program is called local if it can be performed by looking only at the statements in a basic block; otherwise, it is called global.
- Many transformations can be performed at both the local and global levels.
- Local transformations are usually performed first.

Function-Preserving Transformations

There are a number of ways in which a compiler can improve a program without changing the function it computes.

Function preserving transformations examples:

- Common sub expression elimination
- Copy propagation
- Dead-code elimination
- Constant folding

Common Sub expressions elimination:

- An occurrence of an expression E is called a common sub-expression if E was previously computed, and the values of variables in E have not changed since the previous computation.
- We can avoid recomputing the expression if we can use the previously computed value.

Copy Propagation:

- Assignments of the form $f := g$ called copy statements, or copies for short. The idea behind the copy-propagation transformation is to use g for f , whenever possible after the copy statement $f := g$.
- Copy propagation means use of one variable instead of another.
- This may not appear to be an improvement, but as we shall see it gives us an opportunity to eliminate f .

Principal Sources of Optimization



Examples for Common Sub expressions elimination and Copy Propagation

Principal Sources of Optimization



Dead-Code Eliminations:

A variable is live at a point in a program if its value can be used subsequently; otherwise, it is dead at that point. A related idea is dead or useless code, statements that compute values that never get used.

Constant folding:

Deducing at compile time that the value of an expression is a constant and using the constant instead is known as constant folding.

Principal Sources of Optimization



Examples for Dead-Code Eliminations and Constant folding:



Loop Optimizations:

In loops, especially in the inner loops, programs tend to spend the bulk of their time. The running time of a program may be improved if the number of instructions in an inner loop is decreased, even if we increase the amount of code outside that loop.

Three techniques are important for loop optimization:

- **Code motion**, which moves code outside a loop;
- **Induction-variable elimination**, which we apply to replace variables from inner loop.
- **Reduction in strength**, which replaces an expensive operation by a cheaper one, such as a multiplication by an addition.

Principal Sources of Optimization



Code Motion:

An important modification that decreases the amount of code in a loop is code motion. This transformation takes an expression that yields the same result independent of the number of times a loop is executed (a loop-invariant computation) and places the expression before the loop.

```
while (i <= limit-2) /* statement does not change limit*/
```

Code motion will result in the equivalent of

```
t= limit-2;
while (i<=t)
```

Principal Sources of Optimization



Induction Variables :

An **induction variable** is a variable that gets increased or decreased by a fixed amount on every iteration of a loop or is a linear function of another induction variable.

A common compiler optimization is to recognize the existence of induction variables and replace them with simpler computations

For example, in the following loop, i and j are induction variables.

```
for (i = 0; i < 10; ++i) {  
    j = 17 * i;  
}
```

```
j = -17;  
for (i = 0; i < 10; ++i) {  
    j = j + 17;  
}
```

Principal Sources of Optimization



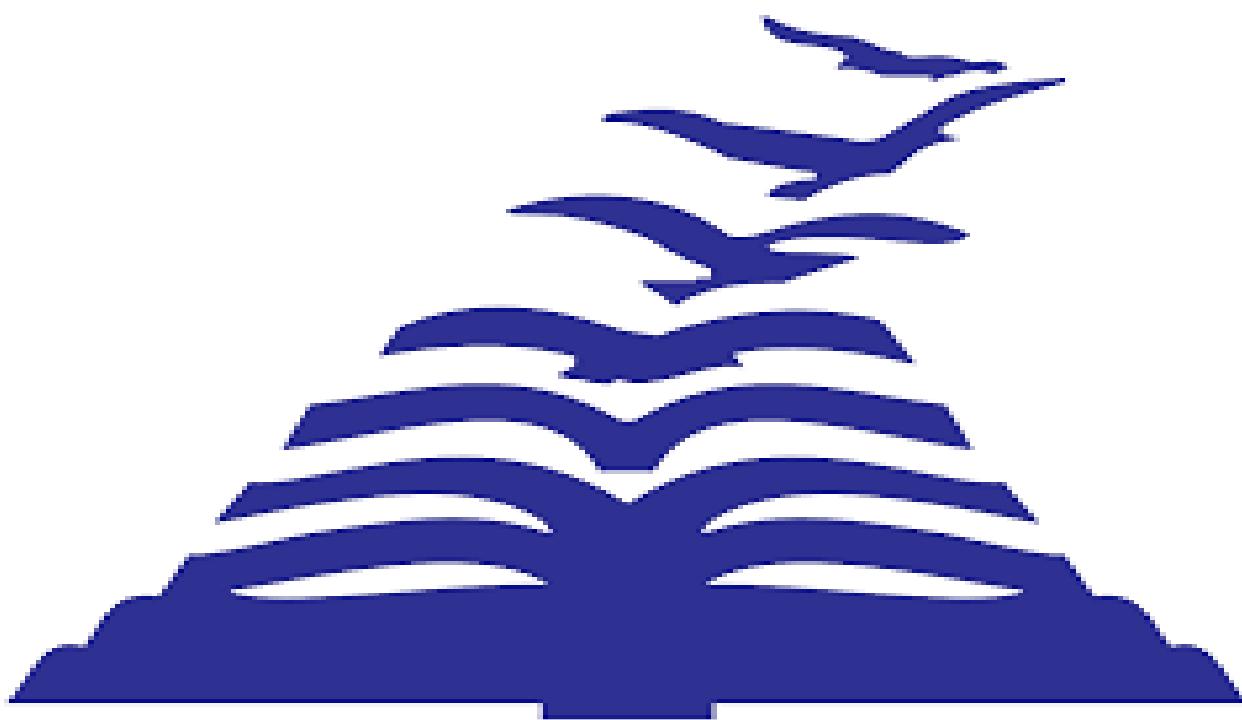
Reduction In Strength:

Reduction in strength replaces expensive operations by equivalent cheaper ones on the target machine.

Certain machine instructions are considerably cheaper than others and can often be used as special cases of more expensive operators.

For example, **x^2 is invariably cheaper to implement as $x * x$ than as a call to an exponentiation routine.**

$2 * X$ is replaced by $X + X$



Presidency University, Bengaluru

Principal Sources of OPTIMIZATION

Code Optimization



The **code produced** by the straight forward compiling algorithms can often be made to **run faster or take less space, or both**. This **improvement is achieved by program transformations** that are traditionally called **optimizations**.

Compilers that apply code-improving transformations are called **optimizing compilers**.



The criteria for code improvement transformations

Simply stated, the best program transformations are those that **yield the most benefit for the least effort.**

The transformations provided by an optimizing compiler should have several properties.

They are:

- The transformation must preserve the meaning of programs.
- A transformation must, on the average, speedup programs by a measurable amount.
- The transformation must be worth the effort.

- A transformation of a program is called local if it can be performed by looking only at the statements in a basic block; otherwise, it is called global.
- Many transformations can be performed at both the local and global levels.
- Local transformations are usually performed first.

Function-Preserving Transformations

There are a number of ways in which a compiler can improve a program without changing the function it computes.

Function preserving transformations examples:

- Common sub expression elimination
- Copy propagation
- Dead-code elimination
- Constant folding

Common Sub expressions elimination:

- An occurrence of an expression E is called a common sub-expression if E was previously computed, and the values of variables in E have not changed since the previous computation.
- We can avoid recomputing the expression if we can use the previously computed value.

Copy Propagation:

- Assignments of the form $f := g$ called copy statements, or copies for short. The idea behind the copy-propagation transformation is to use g for f , whenever possible after the copy statement $f := g$.
- Copy propagation means use of one variable instead of another.
- This may not appear to be an improvement, but as we shall see it gives us an opportunity to eliminate f .

Principal Sources of Optimization



Examples for Common Sub expressions elimination and Copy Propagation

Principal Sources of Optimization



Dead-Code Eliminations:

A variable is live at a point in a program if its value can be used subsequently; otherwise, it is dead at that point. A related idea is dead or useless code, statements that compute values that never get used.

Constant folding:

Deducing at compile time that the value of an expression is a constant and using the constant instead is known as constant folding.

Principal Sources of Optimization



Examples for Dead-Code Eliminations and Constant folding:



Loop Optimizations:

In loops, especially in the inner loops, programs tend to spend the bulk of their time. The running time of a program may be improved if the number of instructions in an inner loop is decreased, even if we increase the amount of code outside that loop.

Three techniques are important for loop optimization:

- **Code motion**, which moves code outside a loop;
- **Induction-variable elimination**, which we apply to replace variables from inner loop.
- **Reduction in strength**, which replaces an expensive operation by a cheaper one, such as a multiplication by an addition.

Principal Sources of Optimization



Code Motion:

An important modification that decreases the amount of code in a loop is code motion. This transformation takes an expression that yields the same result independent of the number of times a loop is executed (a loop-invariant computation) and places the expression before the loop.

```
while (i <= limit-2) /* statement does not change limit*/
```

Code motion will result in the equivalent of

```
t= limit-2;
while (i<=t)
```

Principal Sources of Optimization



Induction Variables :

An **induction variable** is a variable that gets increased or decreased by a fixed amount on every iteration of a loop or is a linear function of another induction variable.

A common compiler optimization is to recognize the existence of induction variables and replace them with simpler computations

For example, in the following loop, i and j are induction variables.

```
for (i = 0; i < 10; ++i) {  
    j = 17 * i;  
}
```

```
j = -17;  
for (i = 0; i < 10; ++i) {  
    j = j + 17;  
}
```

Principal Sources of Optimization



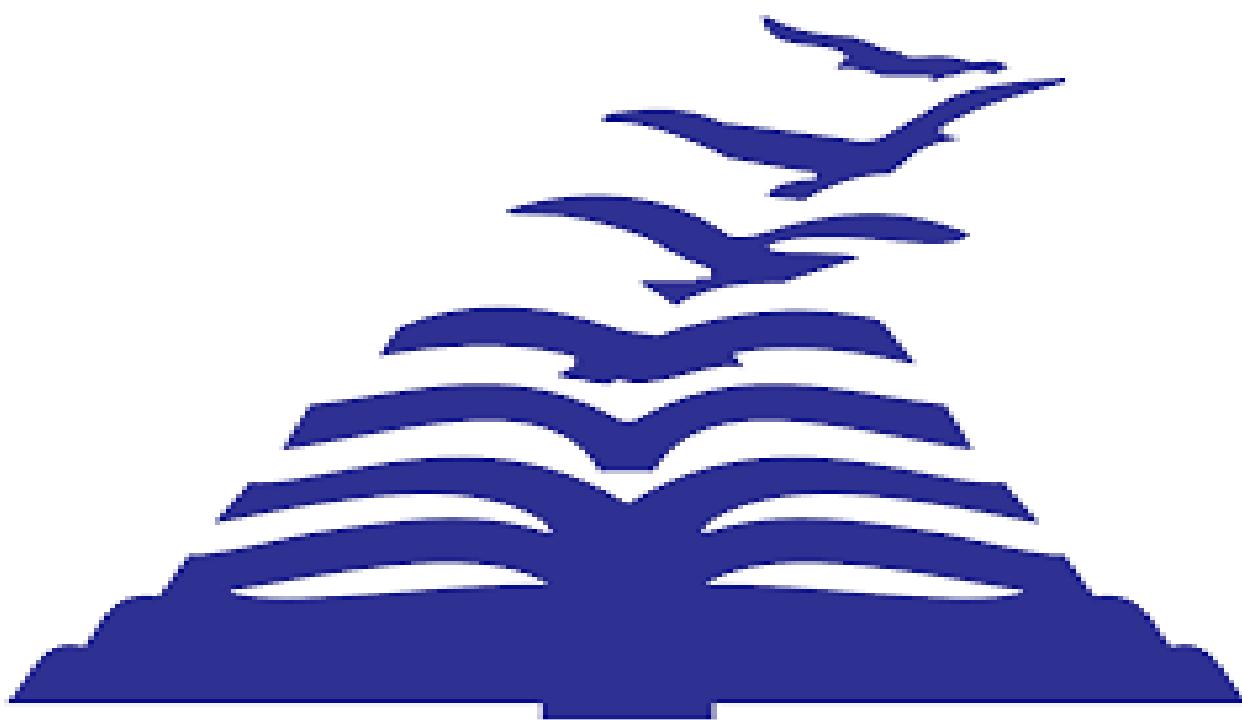
Reduction In Strength:

Reduction in strength replaces expensive operations by equivalent cheaper ones on the target machine.

Certain machine instructions are considerably cheaper than others and can often be used as special cases of more expensive operators.

For example, **x^2 is invariably cheaper to implement as $x * x$ than as a call to an exponentiation routine.**

$2 * X$ is replaced by $X + X$



Presidency University, Bengaluru

Peephole Optimization

Peephole Optimization



- A statement-by-statement code-generations strategy often produces target code that contains redundant instructions and suboptimal constructs. The quality of such target code can be improved by applying “optimizing” transformations to the target program.
- A simple but effective technique for improving the target code is peephole optimization, a method for trying to improving the performance of the target program by examining a short sequence of target instructions (called the peephole) and replacing these instructions by a shorter or faster sequence, whenever possible.



Characteristics of peephole optimizations:

- Redundant-instructions elimination
- Flow-of-control optimizations
- Algebraic simplifications
- Use of machine idioms
- Unreachable



Redundant Loads And Stores:

If we see the instructions sequence

- (1) MOV R0,a
- (2) MOV a,R0

we can delete instructions (2) because whenever (2) is executed. (1) will ensure that the value of a is already in register R0.

Unreachable Code:

Another opportunity for peephole optimizations is the removal of unreachable instructions. An unlabeled instruction immediately following an unconditional jump may be removed. This operation can be repeated to eliminate a sequence of instructions.

Peephole Optimization



```
define debug 1
....
If ( debug )
{
    Print debugging information
}
```

In the intermediate representations the if-statement may be translated as:

```
If debug =1 goto L1 goto L2
L1: print debugging information L2: ..... (a)
```

One obvious peephole optimization is to eliminate jumps over jumps. Thus no matter what the value of debug; (a) can be replaced by:

```
If debug ≠1 goto L2
    Print debugging information
L2: ..... (b)
```

Peephole Optimization



Flows-Of-Control Optimizations:

The unnecessary jumps can be eliminated in either the intermediate code or the target code by the following types of peephole optimizations. We can replace the jump sequence

goto L1

....

L1: gotoL2 (d)

by the sequence

goto L2

....

L1: goto L2

If there are now no jumps to L1, then it may be possible to eliminate the statement L1:goto L2 provided.



Algebraic Simplification:

Only a few algebraic identities occur frequently enough that it is worth considering implementing them. For example, statements such as

x := x+0 or

x := x * 1

are often produced by straightforward intermediate code-generation algorithms, and they can be eliminated easily through peephole optimization.



Reduction in Strength:

Reduction in strength replaces expensive operations by equivalent cheaper ones on the target machine. Certain machine instructions are considerably cheaper than others and can often be used as special cases of more expensive operators.

$$\mathbf{X}^2 \rightarrow \mathbf{X}^* \mathbf{X}$$

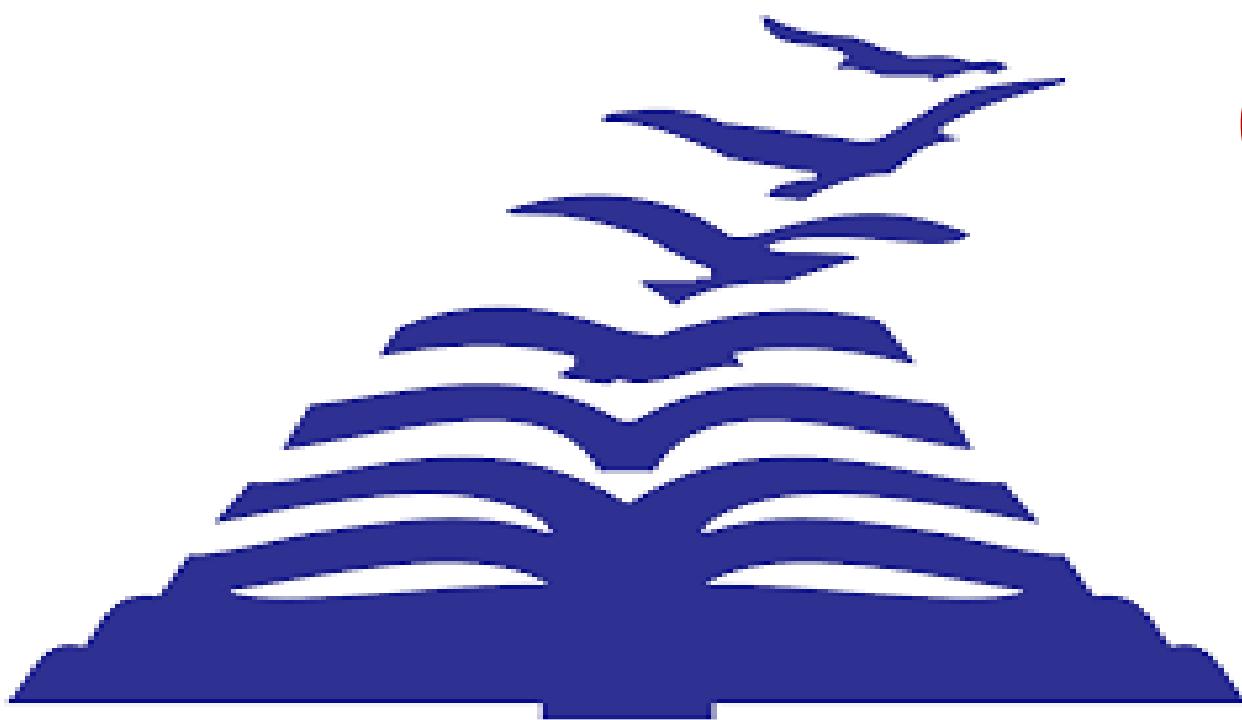


Use of Machine Idioms:

The target machine may have hardware instructions to implement certain specific operations efficiently. For example, some machines have auto-increment and auto-decrement addressing modes. These add or subtract one from an operand before or after using its value. The use of these modes greatly improves the quality of code when pushing or popping a stack, as in parameter passing. These modes can also be used in code for statements like $i := i + 1$.

$i := i + 1$ \rightarrow **$i++$**

$i := i - 1$ \rightarrow **$i--$**



Presidency University, Bengaluru

OPTIMIZATION OF BASIC BLOCKS

OPTIMIZATION OF BASIC BLOCKS



- There are no. of code-improving transformation for basic blocks.
- Two types of basic block optimization mainly used. These are as follows:
 - **Structure-Preserving Transformations**
 - **Algebraic Transformations**

Structure preserving transformations:

The primary Structure-Preserving Transformation on basic blocks is as follows:

- Common sub-expression elimination
- Dead code elimination
- Renaming of temporary variables
- Interchange of two independent adjacent statements

Algebraic transformations:

- Reduction in Strength



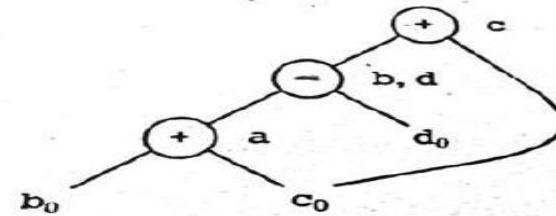
Structure-Preserving Transformations

Common sub expressions can be detected by noticing, as a new node m is about to be added, whether there is an existing node n with the same children, in the same order, and with the same operator. If so, n computes the same value as m and may be used in its place.

A dag for the basic block

- $a = b + c$
- $b = a - d$
- $c = b + c$
- $d = a - d$

- $a = b + c$
- $d = a - d$
- $c = d + c$



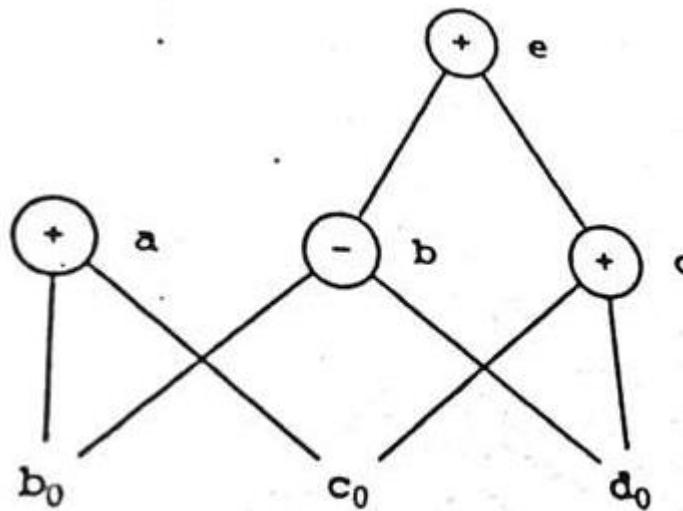
Dag for basic block

In the above expression, the second and forth expression computed the same expression.



Structure-Preserving Transformations

- $a = b + c$
- $b = b - d$
- $c = c + d$
- $e = b + c$





The Use of Algebraic Identities

- Algebraic identities another important class of optimization on basic blocks.
- The simple algebraic transformations that one might try during optimization.
- For examples apply arithmetic identities, such as
 - $x + 0 = 0 + x = x$
 - $x - 0 = x$
 - $x * 1 = 1 * x = x$
 - $x / 1 = x$

The Use of Algebraic Identities

- **Reduction in strength**

Another class of algebraic optimization includes reduction in strength, that is, **replacing a more expensive operator by a cheaper one** as in

- $x * * 2 = x * x$
- $2 * x = x + x$
- $x / 2 = x * 0.5$

- **Constant Folding** (Other kind of optimization for basic blocks)

Evaluate constant expressions at compile time and replace the constant expressions by their values.

2 * 3.14 can be replaced by **6.28**.

OPTIMIZATION OF BASIC BLOCKS



- The dag-construction process also can help us apply more general algebraic transformations such as **commutative** and **associative**.
- For example, suppose $*$ is commutative;
$$x * y = y * x$$
- Before create a new node labeled $*$ with left child x and right child y , check whether such a node already exists, then check for a node having operator $*$, let child y and right child x .

OPTIMIZATION OF BASIC BLOCKS



- The **relational operators** \leq , \geq , $<$, $>$, $=$, and \neq sometimes generate unexpected common sub expressions.

- For example

$X - Y$ and $X > Y$

- Associative laws may also be applied to expose common sub expressions.

For example

- $a = b + c$
- $e = c + d + b$

Intermediate node generated

- $t1 = b + c$
- $a = t1$
- $t2 = c + d$
- $t3 = t2 + b$
- $e = t4$

$t1 = b + c$
 $a = t1$
 $t2 = t1 + d$
 $e = t2$

DAG Representation in Compiler Design

DAG Representation

- DAG stands for Directed Acyclic Graph
- Syntax tree and DAG both, are graphical representations. Syntax tree does not find the common sub expressions whereas DAG can
- Another usage of DAG is the application of optimization technique in the basic block

Characteristics of DAG are:

- All internal nodes store operator values
- External or leaf nodes are identifiers or variable names or constants

Algorithm for Construction of DAG

There are three possible cases to construct DAG on three address code:

Case 1: $x = y \text{ op } z$

Case 2: $x = \text{op } y$

Case 3: $x = y$

DAG can be constructed as follows:

STEP 1:

If y is undefined then create a node with label y . Similarly create a node with label z .

STEP 2:

For case 1, create a node with label op whose left child is node y , and node z will be the right child. Also, check for any common sub expressions. For case 2, determine if a node is labeled op . such node will have a child node y . for case 3 node n will be node y .

STEP 3:

Delete x from list of identifiers for node x . append x to the list of attached identifiers for node n found in step 2.

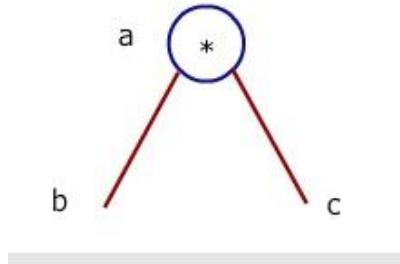
Example:

Consider the following three address code statements.

```
a = b * c  
d = b  
e = d * c  
b = e  
f = b + c  
g = f + d
```

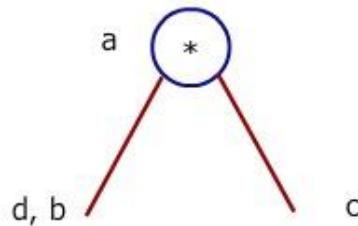
Step 1

Consider the first statement, i.e., $a = b * c$. Create a leaf node with label b and c as left and right child respectively and parent of it will be $*$. Append resultant variable a to the node $*$.



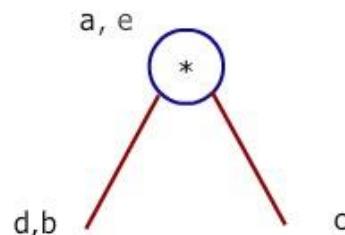
Step 2

For second statement, i.e., $d = b$, node b is already created. So, append d to this node.



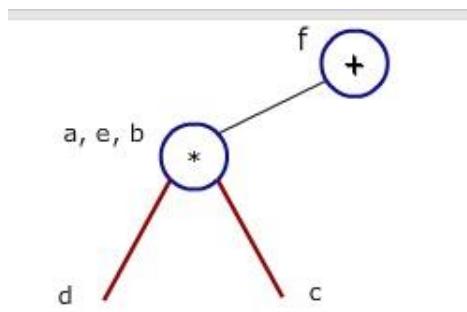
Step 3

For third statement $e = d * c$, the nodes for d , c and $*$ are already created, Node e is not created, so append node e to node $*$.



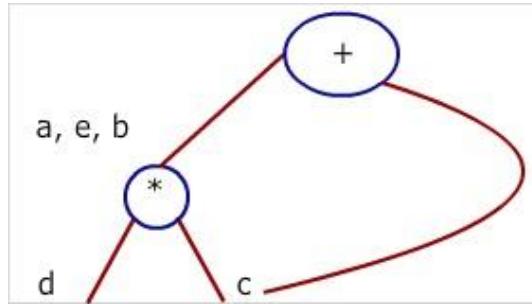
Step 4

For fourth statement $b = e$, append b to node e .



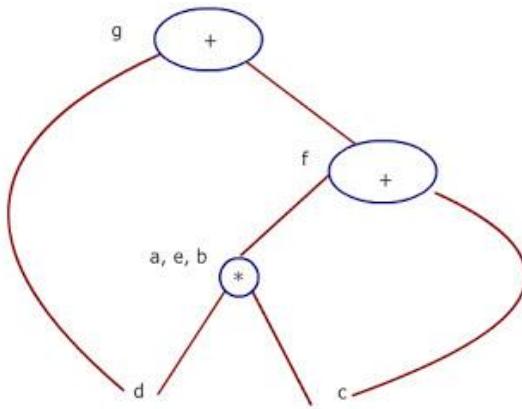
Step 5

For fifth statement $f = b + c$, create a node for operator $+$ whose left child b and right child c and append f to newly created node $+$



Step 6

For last statement $g = f + d$, create a node for operator $+$ whose left child d and right child f and append g to newly created node $+$.



DAG Applications:

- Determines the common sub expressions
- Determines the names used inside the block, and the names that are computed outside the block
- Determines which statements of the block could have their computed value outside the block
- Code may be represented by a DAG describing the inputs and outputs of each of the arithmetic operations performed within the code; this representation allows the compiler to perform common subexpression elimination efficiently
- Several programming languages describe systems of values that are related to each other by a directed acyclic graph. When one value changes, its successors are recalculated; each value is evaluated as a function of its predecessors in the DAG