Compiler Design and Construction (CSC 352)

By

Bhupendra Singh Saud

for

B. Sc. Computer Science & Information Technology

Course Contents

Unit 1:

1.1 Introduction to compiling: Compilers, Analysis of source program, the phases of compiler, compiler-construction tools. **4 hrs**

1.2 A Simple One-Pass Compiler: Syntax Definition, Syntax directed translation, Parsing, Translator for simple expression, Symbol Table, Abstract Stack Machines. **5 hrs**

Unit 2:

2.1 Lexical Analysis: The role of the lexical analyzer, Input buffering, Specification of tokens, Recognition of tokens, Finite Automata, Conversion Regular Expression to an NFA and then to DFA, NFA to DFA, State minimization in DFA, Flex/lex introduction. **8 Hrs**

2.2 Syntax Analysis: The role of parser, Context frees grammars, Writing a grammars, Top-down parsing, Bottom-up parsing, error recovery mechanism, LL grammar, Bottom up parsing-Handles, shift reduced parsing, LR parsers-SLR,LALR,LR,LR/LALR Grammars, parser generators. **10 Hrs**

Unit 3:

3.1 Syntax Directed Translation: Syntax-directed definition, Syntax tree and its construction, Evaluation of S-attributed definitions, L-attributed, Top-down translation, Recursive evaluators. **5 Hrs**

3.2 Type Checking: Type systems, Specification of a simple type checker, Type conversions equivalence of type expression, Type checking Yacc/Bison. **3 Hrs**

Unit 4:

4.1 Intermediate Code Generation: Intermediate languages, three address code, Declarations, Assignments Statements, Boolean Expressions, addressing array elements, case statements, Back patching, procedure calls. 4 **Hrs**

4.2 Code Generation and optimization: Issues in design of a code generator, the target machine, Run –time storage management, Basic blocks and flow graphs, next use information's, a simple code generator, Peephole organization, generating code from dags. **6 Hrs**

Subject: Compiler Design and Construction **FM: 60**

Time: 3 hours **PM: 24**

Candidates are required to give their answer in their own words as for as practicable.
Attempt all the questions.
Every question contains equal marks.

1. What do mean by compiler? How source program analyzed? Explain in brief.
2. Discuss the role of symbol table in compiler design.
3. Convert the regular expression '0+ (1+0)*00' first into NFA and then into DFA using Thomson's and Subset Construction methods.
4. Consider the grammar:
   a. S→(L) | a
   b. L→L, S | S
5. Consider the grammar
   a. C→AB
   b. A →a
   c. B→ a

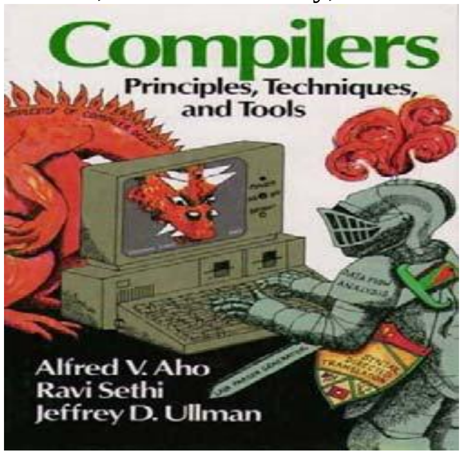
   Calculate the canonical LR (0) items.
6. Describe the inherited and synthesized attributes of grammar using an example.
7. Write the type expressions for the following types.
   - An array of pointers to real, where the array index range from 1 to 100.
   - Function whose domains are function from characters and whose range is a Pointer of integer.
8. What do you mean by intermediate code? Explain the role of intermediate code in compiler design.
9. What is operation of simple code generator? Explain.
10. Why optimization is often required in the code generated by simple code generator? Explain the unreachable code optimization.

Prerequisites

- Introduction to Automata and Formal Languages
- Introduction to Analysis of Algorithms and Data Structures
- Working knowledge of C/C++
- Introduction to the Principles of Programming Languages, Operating System & Computer Architecture is plus

Resources

Text Book: Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986



What is Compiler?

A compiler is a translator software program that takes its input in the form of program written in one particular programming language (source language) and produce the output in the form of program in another language (object or target language).

- A compiler is a translator that converts the high-level language into the machine language.
- High-level language is written by a developer and machine language can be understood by the processor.
- Compiler is used to show errors to the programmer.
- The main purpose of compiler is to change the code written in one language without changing the meaning of the program.
- When you execute a program which is written in HLL programming language then it executes into two parts.
- In the first part, the source program compiled and translated into the object program (low level language).
- In the second part, object program translated into the target program through the assembler.

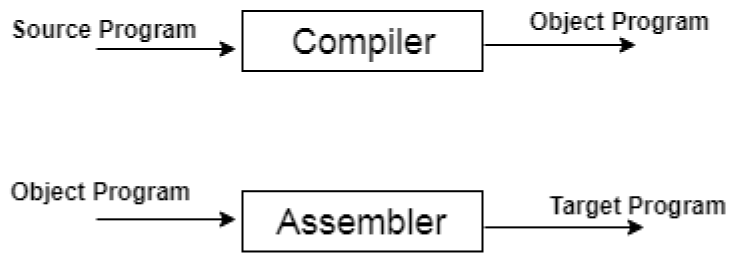


Fig: Execution process of source program in Compiler

Phases of a Compiler

The compilation process contains the sequence of various phases. Each phase takes source program in one representation and produces output in another representation. Each phase takes input from its previous stage. It also involves the symbol table and error handler. There are two major parts of a compiler **Analysis** and **Synthesis.**

Analysis part

In analysis part, an intermediate representation is created from the given source program.
This part is also called front end of the compiler. This part consists of mainly following four phases:

- Lexical Analysis
- Syntax Analysis
- Semantic Analysis and
- Intermediate code generation

Synthesis part

In synthesis part, the equivalent target program is created from intermediate representation of the program created by analysis part. This part is also called back end of the compiler. This part consists of mainly following two phases:

- Code Optimization and
- Final Code Generation

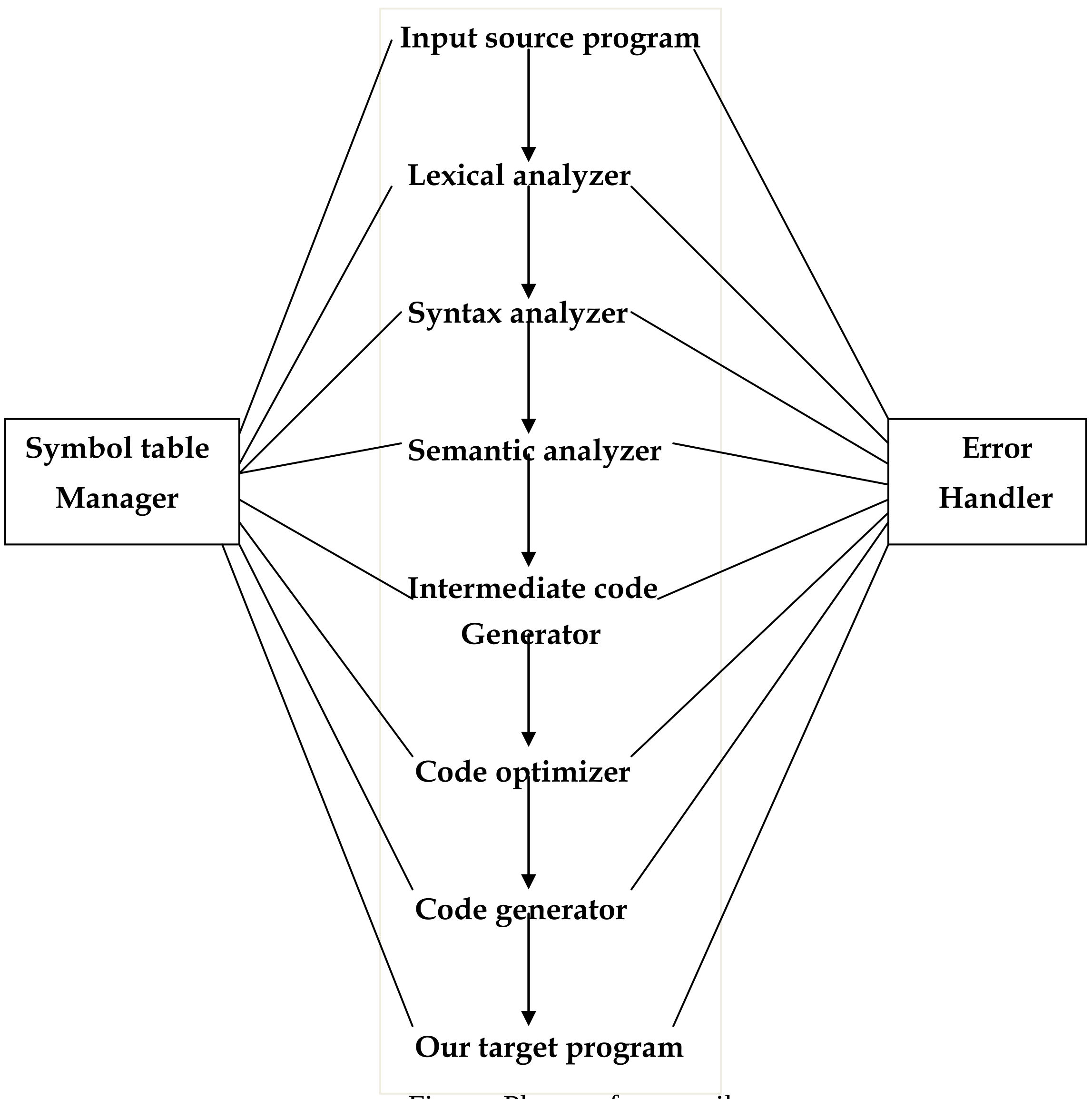
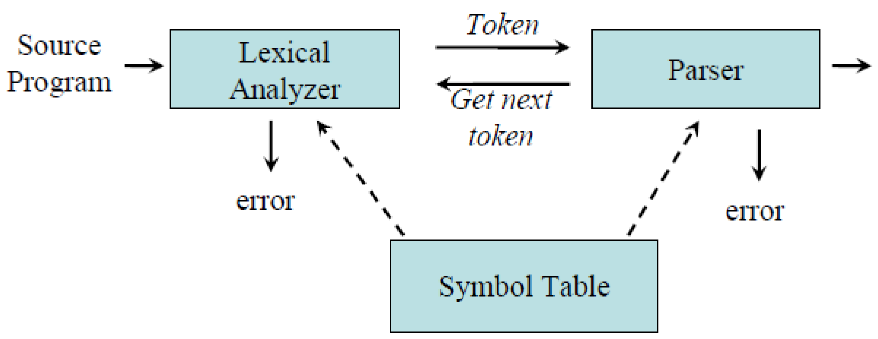


Figure: Phases of a compiler

1. Lexical Analysis (or Scanning)

Lexical analysis or scanning is the process where the source program is read from left-to-right and grouped into tokens. Tokens are sequences of characters with a collective meaning. In any programming language tokens may be constants, operators, reserved words, punctuations etc.

The Lexical Analyzer takes a source program as input, and produces a stream of tokens as output. Normally a lexical analyzer doesn't return a list of tokens; it returns a token only when the parser asks a token from it. Lexical analyzer may also perform other auxiliary operation like removing redundant white space, removing token separator (like semicolon) etc. In this phase only few limited errors can be detected such as illegal characters within a string, unrecognized symbols etc. Other remaining errors can be detected in the next phase called syntax analyzer.



Example:

```
While(i>0)
        i=i-2;
```

Tokens	**description**
while	while keyword
(	left parenthesis
i	identifier
>	greater than symbol
0	integers constant
)	right parenthesis
i	identifier
=	Equals
i	identifier
-	Minus
2	integers constant
;	Semicolon

The main purposes of lexical analyzer are:

- It is used to analyze the source code.
- Lexical analyzer is able to remove the comments and the white space present in the expression.
- It is used to format the expression for easy access i.e. creates tokens.
- It begins to fill information in SYMBOL TABLE.

2. Syntax Analyzer (or Parsing)

The second phase of the compiler phases is syntax Analyzer, once lexical analysis is completed the generation of lexemes and mapped to the token, then parser takes over to check whether the sequence of tokens is grammatically correct or not, according to the rules that define the syntax of the source language.

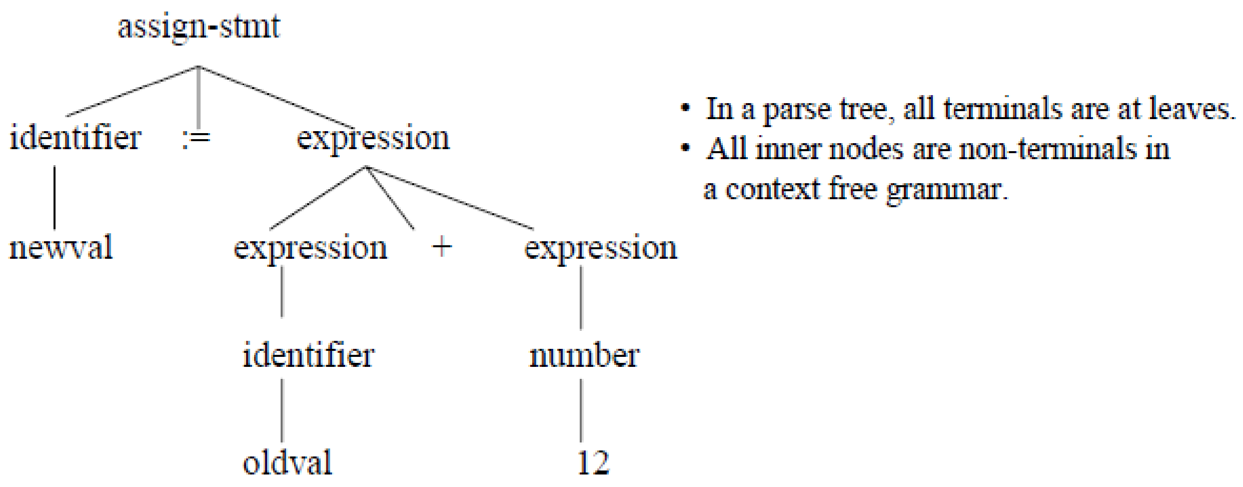
The main purposes of Syntax analyzer are:

- Syntax analyzer is capable analyzes the tokenized code for structure.

- This is able to tags groups with type information.

A **Syntax Analyzer** creates the syntactic structure (generally a parse tree) of the given source program. Syntax analyzer is also called the **parser.** Its job is to analyze the source program based on the definition of its syntax. It is responsible for creating a parse-tree of the source code.

Ex: newval := oldval + 12


assign-stmt
identifier
:=
expression
newval
expression
+
expression
identifier
number
oldval
12
• In a parse tree, all terminals are at leaves.
• All inner nodes are non-terminals in a context free grammar.

The syntax of a language is specified by a **context free grammar** (CFG).

The rules in a CFG are mostly recursive.

A syntax analyzer checks whether a given program satisfies the rules implied by a CFG or not.

– If it satisfies, the syntax analyzer creates a parse tree for the given program.

3. Semantic Analyzer

The next phase of the semantic analyzer is the semantic analyzer and it performs a very important role to check the semantics rules of the expression according to the source language. The previous phase output i.e. syntactically correct expression is the input of the semantic analyzer. Semantic analyzer is required when the compiler may require performing some additional checks such as determining the type of expressions and checking that all statements are correct with respect to the typing rules, that variables have been properly declared before they are used, that functions are called with the proper number of parameters etc. This semantic analyzer phase is carried out using information from the parse tree and the symbol table.

The parsing phase only verifies that the program consists of tokens arranged in a syntactically valid combination. Now semantic analyzer checks whether they form a sensible set of instructions in the programming language or not. Some examples of the things checked in this phase are listed below:

- The type of the right side expression of an assignment statement should match the type of the left side ie. in the expression ***newval = oldval + 12,*** The type of the expression *(oldval+12)* must match with type of the variable *newval.*
- The parameter of a function should match the arguments of a function call in both number and type.
- The variable name used in the program must be unique etc.

The main purposes of Semantic analyzer are:

- It is used to analyze the parsed code for meaning.
- Semantic analyzer fills in assumed or missing information.
- It tags groups with meaning information.

Important techniques that are used for Semantic analyzer:

- The specific technique used for semantic analyzer is Attribute Grammars.
- Another technique used by the semantic analyzer is Ad hoc analyzer.

4. Intermediate code generator

If the program syntactically and semantically correct then intermediate code generator generates a simple machine independent intermediate language. The intermediate language should have two important properties:

- It should be simple and easy to produce.
- It should be easy to translate to the target program

Some compiler may produce an explicit intermediate codes representing the source program. These intermediate codes are generally machine (architecture) independent. But the level of intermediate codes is close to the level of machine codes.

Example: A = b + c * d / f
Solution
Intermediate code for above example

```
T1 = c * d
T2 = T1 / f
T3 = b + T2
A = T3
```

The main purposes of Intermediate code generation are:

- This phase is used to generate the intermediate code of the source code.

Important techniques that are used for Intermediate code generations:

- Intermediate code generation is done by the use of Three address code generation.

Code Optimization

Optimization is the process of transforming a piece of code to make more efficient (either in terms of time or space) without changing its output or side effects. The process of removing unnecessary part of a code is known as code optimization. Due to code optimization process it decreases the time and space complexity of the program. i.e

Detection of redundant function calls
Detection of loop invariants
Common sub-expression elimination
Dead code detection and elimination

```
b  := 0
t1 := a + b
t2 := c * t1
a  := t2
```

$$\Downarrow$$

```
a := c*a
b := 0
```

The main purposes of Code optimization are:

- It examines the object code to determine whether there are more efficient means of execution.

Important techniques that are used for lexical analyzer:

- Loop unrolling.
- Common-sub expression elimination
- Operator reduction etc.

Code Generation

It generates the assembly code for the target CPU from an optimized intermediate representation of the program.

Ex: Assume that we have an architecture with instructions whose at least one of its operands is a machine register.

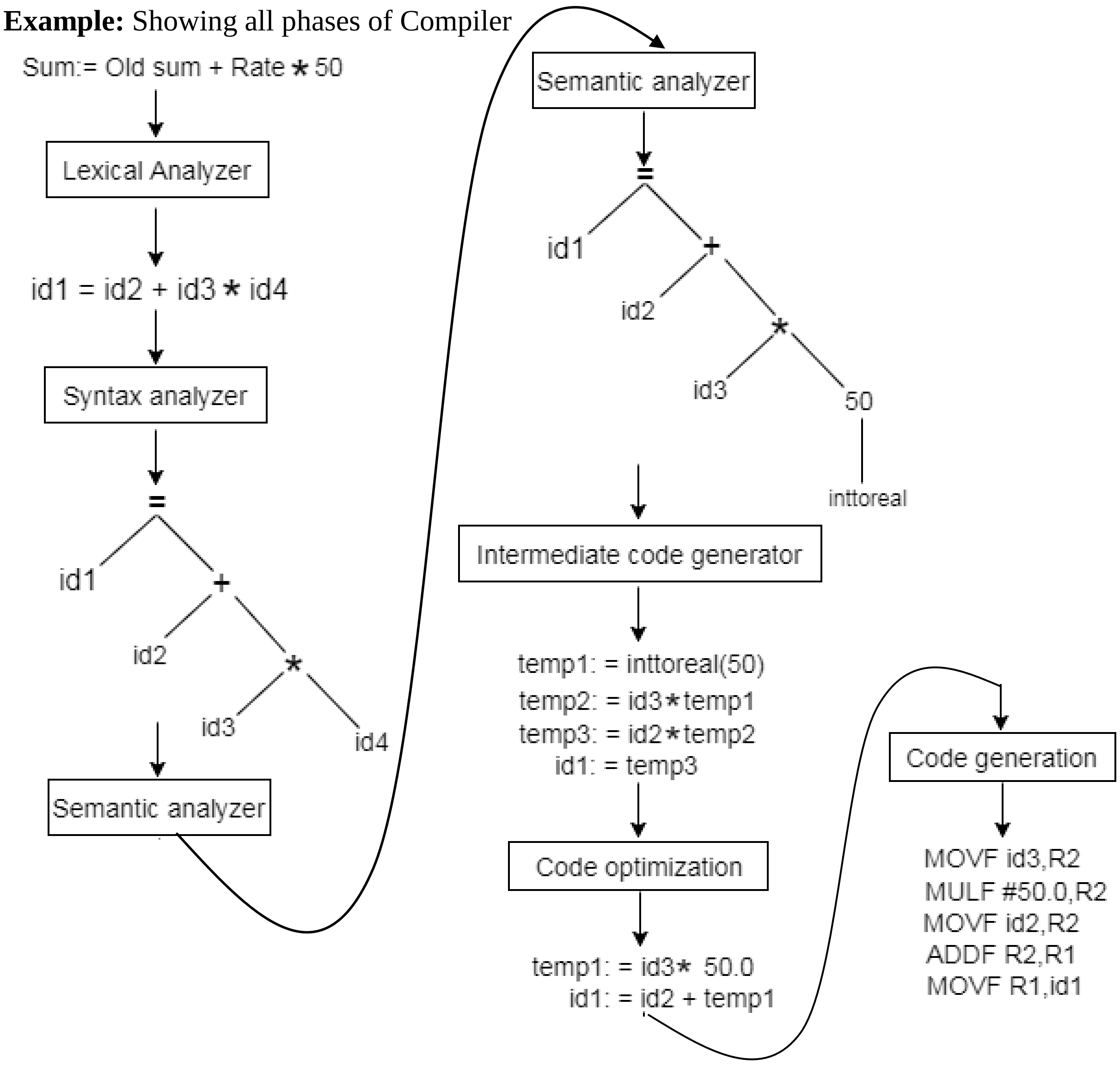
A = b + c * d / f

$$\Downarrow$$

```
MOVE    c, R1
MULT    d, R1
DIV     f, R1
ADD     b, R1
```

MOVE R1, A

Example: Showing all phases of Compiler



One pass VS Multi-pass compiler

Each individual unique step in compilation process is called a phase such as lexical analysis, syntax analysis, semantic analysis and so on. Different phases can be combined into one or more than one group. These each group is called passes. If all the phases are combined into a single group then this is called as one pass compiler otherwise more than one pass constitute the multi-pass compiler.

One pass compiler	Multi-pass compiler
1. In a one pass compiler all the phases are combined into one pass. 2. Here intermediate representation of source program is not created. 3. It is faster than multi-pass compiler.	1. In multi-pass compiler different phases of compiler are grouped into multiple phases. 2. Here intermediate representation of source program is created. 3. It is slightly slower than one pass compiler.

4. It is also called narrow compiler. 5. Pascal's compiler is an example of one pass compiler. 6. A single-pass compiler takes more space than the multi-pass compiler	4. It is also called wide compiler. 5. C++ compiler is an example of multi-pass compiler. 6. A multi-pass compiler takes less space than the multi-pass compiler because in multi-pass compiler the space used by the compiler during one pass can be reused by the subsequent pass.

Multi-pass Compiler

- Multi pass compiler is used to process the source code of a program several times.
- In the first pass, compiler can read the source program, scan it, extract the tokens and store the result in an output file.
- In the second pass, compiler can read the output file produced by first pass, build the syntactic tree and perform the syntactical analysis. The output of this phase is a file that contains the syntactical tree.
- In the third pass, compiler can read the output file produced by second pass and check that the tree follows the rules of language or not. The output of semantic analysis phase is the annotated tree syntax.
- This pass is going on, until the target output is produced.

One-pass Compiler

- One-pass compiler is used to traverse the program only once. The one-pass compiler passes only once through the parts of each compilation unit. It translates each part into its final machine code.
- In the one pass compiler, when the line source is processed, it is scanned and the token is extracted.
- Then the syntax of each line is analyzed and the tree structure is build. After the semantic part, the code is generated.
- The same process is repeated for each line of code until the entire program is compiled.

Compiler Construction Tools

For the construction of a compiler, the compiler writer uses different types of software tools that are known as compiler construction tools. These tools make use of specialized languages for specifying and implementing specific components, and most of them use sophisticated algorithms. The tools should hide the details of the algorithm used and produce component in such a way that they can be easily integrated into the rest of the compiler. Some of the most commonly used compiler construction tools are:

- **Scanner generators:** They automatically produce lexical analyzers or scanners. Example: flex, lex, etc
- **Parser generators:** They produce syntax analyzers or parsers. Example: bison, yacc etc.
- **Syntax-directed translation engines:** They produce a collection of routines, which traverses the parse tree and generates the intermediate code.
- **Code generators:** They produce a code generator from a set of rules that translates the intermediate language instructions into the equivalent machine language instructions for the target machine.
- **Data-flow analysis engines:** They gather the information about how the data is transmitted from one part of the program to another. For code optimization, data-flow analysis is a key part.
- **Compiler-construction toolkits:** They provide an integrated set of routines for construction of the different phases of a compiler.

Symbol Tables

Symbol tables are data structures that are used by compilers to hold information about source-program constructs. The information is collected incrementally by the analysis phase of a compiler and used by the synthesis phases to generate the target code. Entries in the symbol table contain information about an identifier such as its type, its position in storage, and any other relevant information. Symbol tables typically need to support multiple declarations of the same identifier within a program. The lexical analyzer can create a symbol table entry and can return token to the parser, say ***id***, along with a pointer to the lexeme. Then the parser can decide whether to use a previously created symbol table or create new one for the identifier.

The basic operations defined on a symbol table include

- **allocate** – to allocate a new empty symbol table
- **free** – to remove all entries and free the storage of a symbol table
- **insert** – to insert a name in a symbol table and return a pointer to its entry
- **lookup** – to search for a name and return a pointer to its entry
- **set_attribute** – to associate an attribute with a given entry
- **get_attribute** – to get an attribute associated with a given entry

Other operations can be added depending on requirement

- For example, a **delete** operation removes a name previously inserted

Possible entries in a symbol table:

- Name: a string.
- Attribute:
    - ✓ Reserved word
    - ✓ Variable name
    - ✓ Type name
    - ✓ Procedure name
    - ✓ Constant name
    - ✓ _ _ _ _ _ _ _
- Data type
- Scope information: where it can be used.
- Storage allocation, size…
- ………………

Example: Let's take a portion of a program as below:

```
void fun ( int A, float B)
{
        int D, E;
        D = 0;
        E = A / round (B);
        if (E > 5)
        {
                Print D
        }
}
```

Its symbol table is created as below:

Symbol	**Token**	**Data type**	**Initialization?**
Fun	Id	Function name	No
A	Id	Int	Yes
B	Id	Float	Yes
D	Id	Int	No
E	Id	Int	No

Symbol	**Token**	**Data type**	**Initialization?**
Fun	Id	Function name	No
A	Id	Int	Yes
B	Id	Float	Yes
D	Id	Int	Yes
E	Id	Int	Yes

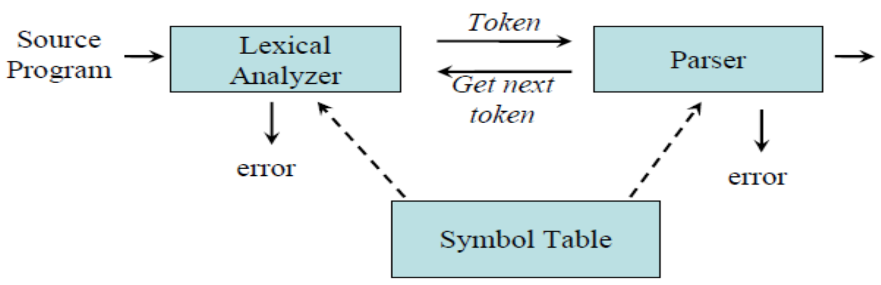
Error handling in compiler

Error detection and reporting of errors are important functions of the compiler. Whenever an error is encountered during the compilation of the source program, an error handler is invoked. Error handler generates a suitable error reporting message regarding the error encountered. The error reporting message allows the programmer to find out the exact location of the error. Errors can be encountered at any phase of the compiler during compilation of the source program for several reasons such as:

- In lexical analysis phase, errors can occur due to misspelled tokens, unrecognized characters, etc. These errors are mostly the typing errors.
- In syntax analysis phase, errors can occur due to the syntactic violation of the language.
- In intermediate code generation phase, errors can occur due to incompatibility of operands type for an operator.
- In code optimization phase, errors can occur during the control flow analysis due to some unreachable statements.
- In code generation phase, errors can occurs due to the incompatibility with the computer architecture during the generation of machine code. For example, a constant created by compiler may be too large to fit in the word of the target machine.
- In symbol table, errors can occur during the bookkeeping routine, due to the multiple declaration of an identifier with ambiguous attributes.

Lexical Analysis

The lexical analysis is the first phase of a compiler where a lexical analyzer acts as an interface between the source program and the rest of the phases of compiler. It reads the input characters of the source program, groups them into lexemes, and produces a sequence of tokens for each lexeme. The tokens are then sent to the parser for syntax analysis. Normally a lexical analyzer doesn't return a list of tokens; it returns a token only when the parser asks a token from it. Lexical analyzer may also perform other auxiliary operation like removing redundant white space, removing token separator (like semicolon) etc.



Example:

newval := oldval + 12

tokens:	newval	identifier
	:=	assignment operator
	oldval	identifier
	+	add operator

12	a number

Put information about identifiers into the symbol table.
Regular expressions are used to describe tokens (lexical constructs).
A (Deterministic) Finite State Automaton (DFA) can be used in the implementation of a lexical analyzer.

Tokens, Patterns, Lexemes

A *token* is a logical building block of language. They are the sequence of characters having a collective meaning.
Example: identifier, keywords, integer constants, string constant etc

A sequence of input characters that make up a single token is called a lexeme.
A token can represent more than one lexeme. The token is a general class in which lexeme belongs to.
Example: The token "String constant" may have a number of lexemes such as "bh", "sum", "area", "name" etc.
Thus lexeme is the particular member of a token which is a general class of lexemes.

Patterns are the rules for describing whether a given lexeme belonging to a token or not. Regular expressions are widely used to specify patterns.

Lexemes are said to be a sequence of characters (alphanumeric) in a token. There are some predefined rules for every lexeme to be identified as a valid token. These rules are defined by grammar rules, by means of a pattern. A pattern explains what can be a token, and these patterns are defined by means of regular expressions.
In programming language, keywords, constants, identifiers, strings, numbers, operators and punctuations symbols can be considered as tokens.

Attributes of Tokens

When a token represents more than one lexeme, lexical analyzer must provide additional information about the particular lexeme. This additional information is called as the *attribute* of the token.
For simplicity, a token may have a single attribute which holds the required information for that token.
Example: the tokens and the associated attribute for the following statement.

```
A=B*C+2
<id, pointer to symbol table entry for A>
<assig operator>
<id, pointer to symbol table entry for B>
<mult_op>
<id, pointer to symbol table entry for C>
<add_op>
<num, integer value 2>
```

Input Buffering

- Reading character by character from secondary storage is slow process and time consuming as well. It is necessary to look ahead several characters beyond the lexeme for a pattern before a match can be announced.
- One technique is to read characters from the source program and if pattern is not matched then push look ahead character back to the source program.
- This technique is time consuming.
- Use buffer technique to eliminate this problem and increase efficiency.

Many times, a scanner has to look ahead several characters from the current character in order to recognize the token.

For example *int* is keyword in C, while the term *inp* may be a variable name. When the character *'i'* is encountered, the scanner cannot decide whether it is a keyword or a variable name until it reads two more characters.

In order to efficiently move back and forth in the input stream, input buffering is used.

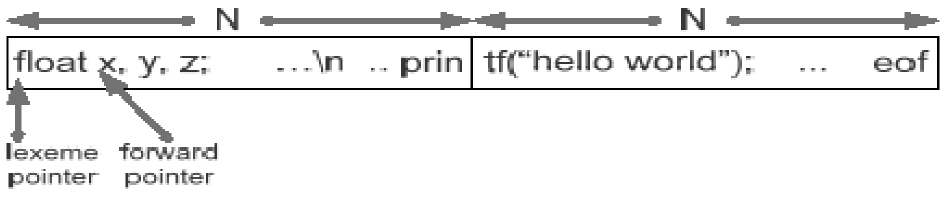


Fig: - An input buffer in two halves

Here, we divide the buffer into two halves with N-characters each.

Rather than reading character by character from file we read N input character at once.

If there are fewer than N characters in input eof marker is placed.

There are two pointers (see in above fig.) the portion between lexeme pointer and forward pointer is current lexeme. Once the match for pattern is found, both the pointers points at the same place and forward pointer is moved.

The forward pointer performs tasks like below:

```
If forward at end of first half then,
        Reload second half
        Forward++
end if
else if forward at end of second half then,
        Reload first half
        Forward=start of first half
end else if
else
        forward++
```

Specifications of Tokens

Regular expressions are an important notation for specifying patterns. Each pattern matches a set of strings, so regular expressions will serve as names for sets of strings.

Alphabets

Any finite set of symbols {0,1} is a set of binary alphabets, {0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F} is a set of Hexadecimal alphabets, {a-z, A-Z} is a set of English language alphabets.

Strings

Any finite sequence of alphabets is called a string. Length of the string is the total number of occurrence of alphabets, e.g., the length of the string newsummitcollege is 16 and is denoted by |newsummitcollege| = 16. A string having no alphabets, i.e. a string of zero length is known as an empty string and is denoted by ε (epsilon).

Special Symbols

A typical high-level language contains the following symbols:-

Arithmetic Symbols	Addition(+), Subtraction(-), Modulo(%), Multiplication(*), Division(/)
Punctuation	Comma(,), Semicolon(;), Dot(.), Arrow(->)
Assignment	=
Special Assignment	+=, /=, *=, -=
Comparison	==, !=, <, <=, >, >=
Preprocessor	#
Location Specifier	&
Logical	&, &&, |, ||, !
Shift Operator	>>, >>>, <<, <<<

Language

A language is considered as a finite set of strings over some finite set of alphabets. Computer languages are considered as finite sets, and mathematically set operations can be performed on them. Finite languages can be described by means of regular expressions.

Kleene Closure

Kleene closure of an alphabet A denoted by A* is set of all strings of any length (0 also) possible from A. Mathematically $A^* = A^0 \cup A^1 \cup A^2 \cup \cdots \cdots$ For any string, w over alphabet A, $w \in A^*$

Notations

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

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

Recognition of tokens

To recognize tokens lexical analyzer performs following steps:

a. Lexical analyzers store the input in input buffer.
b. The token is read from input buffer and regular expressions are built for corresponding token
c. From these regular expressions finite automata is built. Usually NFA is built.
d. For each state of NFA, a function is designed and each input along the transitional edges corresponds to input parameters of these functions.
e. The set of such functions ultimately create lexical analyzer program.

Regular Expressions

Regular expressions are the algebraic expressions that are used to describe tokens of a programming language.

Examples

Given the alphabet A = {0, 1}

1. 1(1+0)*0 denotes the language of all string that begins with a '1' and ends with a '0'.
2. (1+0)*00 denotes the language of all strings that ends with 00 (binary number multiple of 4)
3. (01)*+ (10)* denotes the set of all stings that describe alternating 1s and 0s
4. (0* 1 0* 1 0* 1 0*) denotes the string having exactly three 1's.
5. 1*(0+ ε)1*(0+ ε) 1* denotes the string having at most two 0's
6. (A | B | C |………| Z | a | b | c |………| z | _ |). ((A | B | C |………| Z | a | b | c |………| z | _ |) (1 | 2 |…………….| 9))* denotes the regular expression to specify the identifier like in C. [TU]
7. (1+0)* 001 (1+0)* denotes string having substring 001

Regular Definitions

To write regular expression for some languages can be difficult, because their regular expressions can be quite complex. In those cases, we may use *regular definitions*.

The regular definition is a sequence of definitions of the form,

$d1 \rightarrow r1$

$d2 \rightarrow r2$

…………….

$d_n \rightarrow r_n$

Where d_i is a distinct name and r_i is a regular expression over symbols in $\Sigma \cup$ {d1, d2... di-1}

Where, Σ = Basic symbol and

{d1, d2... di-1} = previously defined names.

Regular Definitions: Examples

Regular definition for specifying identifiers in a programming language like C

letter → A | B | C |………| Z | a | b | c |………| z

underscore →'_'

digit →0 | 1 | 2 |…………….| 9

id → (letter | underscore).(letter | underscore | digit)*

If we are trying to write the regular expression representing identifiers without using regular definition, it will be complex.

(A | B | C |………| Z | a | b | c |………| z | _ |). ((A | B | C |………| Z | a | b | c |………| z | _ |) (1 | 2 |…………….| 9))*

Exercise

Write regular definition for specifying floating point number in a programming language like C

Sol^n: digit →0 | 1 | 2 |…………….| 9

fnum→ digit * (.digit$^+$)

Write regular definitions for specifying an integer array declaration in language like C

Sol^n: letter → A | B | C |………| Z | a | b | c |………| z

underscore →'_'

digit → 1 | 2 |…………….| 9

array → (letter | underscore).(letter | underscore | digit)* ([digit$^+$.0*])$^+$

Design of a Lexical Analyzer

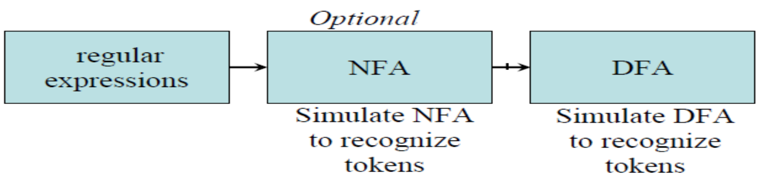
First, we define regular expressions for tokens; then we convert them into a DFA to get a lexical analyzer for our tokens.

Algorithm1:

Regular Expression → NFA → DFA (two steps: first to NFA, then to DFA)

Algorithm2:

Regular Expression → DFA (directly convert a regular expression into a DFA)


Optional
regular expressions
NFA
DFA
Simulate NFA to recognize tokens
Simulate DFA to recognize tokens

Non-Deterministic Finite Automaton (NFA)

FA is non deterministic, if there is more than one transition for each (state, input) pair. It is slower recognizer but it make take less spaces. An NFA is a 5-tuple (*S*, Σ, δ, *s*0, *F*) where

S is a finite set of *states*

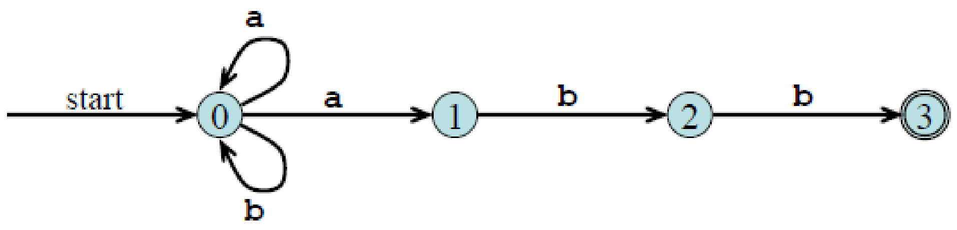
Σ is a finite set of symbols

δ is a *transition function*

$s_0 \in S$ is the *start state*

$F \subseteq S$ is the set of *accepting* (or *final*) *states*

A NFA accepts a string x, if and only if there is a path from the starting state to one of accepting states.



$$S = \{0,1,2,3\}$$
$$\Sigma = \{a,b\}$$
$$s_0 = 0$$
$$F = \{3\}$$

Fig: - NFA for regular expression (a + b)*a b b

ε- NFA

In NFA if a transition made without any input symbol is called ε-NFA.

Here we need ε-NFA because the regular expressions are easily convertible to ε-NFA.

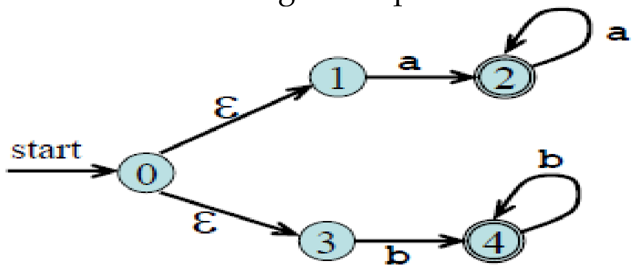


Fig: - ε-NFA for regular expression aa* +bb*

Deterministic Finite Automaton (DFA)

DFA is a special case of NFA. There is only difference between NFA and DFA is in the transition function. In NFA transition from one state to multiple states take place while in DFA transition from one state to only one possible next state take place.

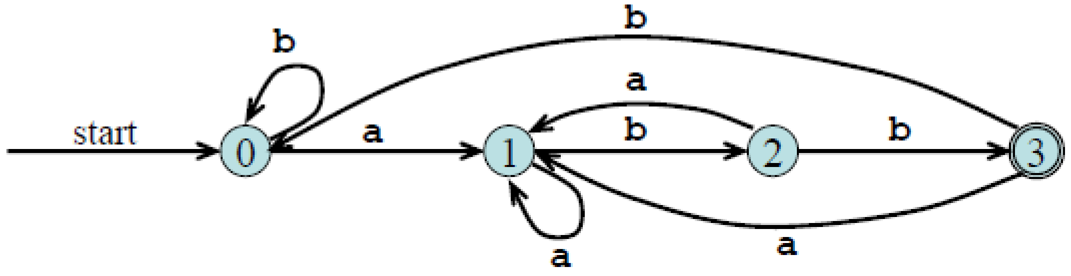


Fig:-DFA for regular expression (**a**+**b**)***abb**

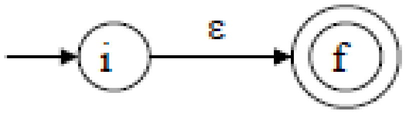
Conversion: Regular Expression to NFA

Thomson's Construction

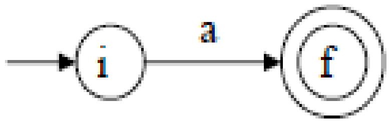
Thomson's Construction is simple and systematic method.
It guarantees that the resulting NFA will have exactly one final state, and one start state.
Method:

- First parse the regular expression into sub-expressions
- Construct NFA's for each of the basic symbols in regular expression (r)
- Finally combine all NFA's of sub-expressions and we get required NFA of given regular expression.

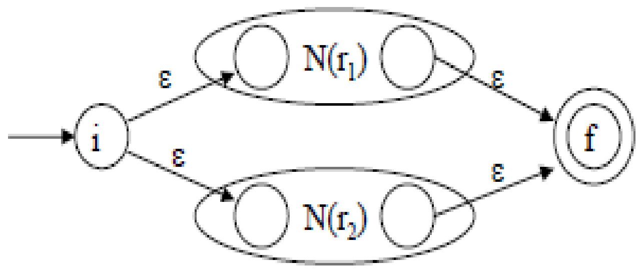
1. To recognize an empty string ε



2. To recognize a symbol a in the alphabet Σ

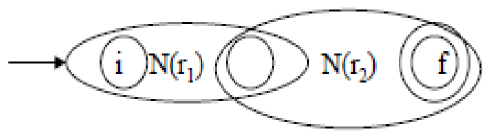


3. If N (r1) and N (r2) are NFAs for regular expressions r1 and r2
   a. For regular expression r1 + r2



$$N(r_1 + r_2)$$

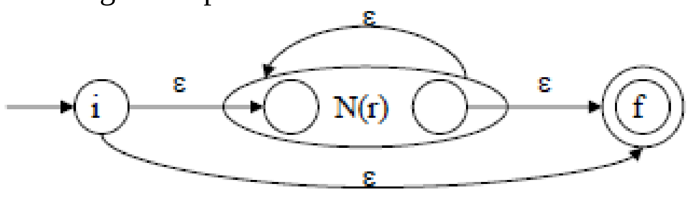
   b. For regular expression r1 r2



The start state of $N(r_1)$ becomes the start state of $N(r_1r_2)$ and final state of $N(r_2)$ become final state of $N(r_1r_2)$

$N(r_1\ r_2)$

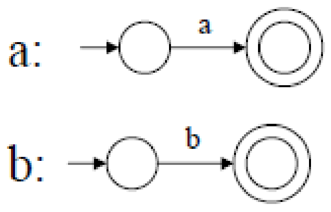
c. For regular expression r*

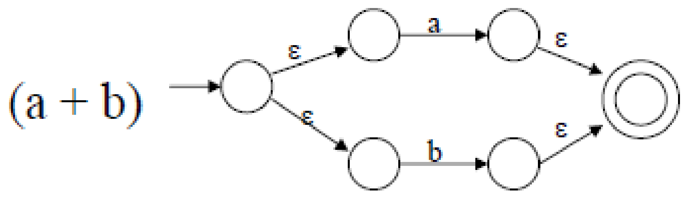


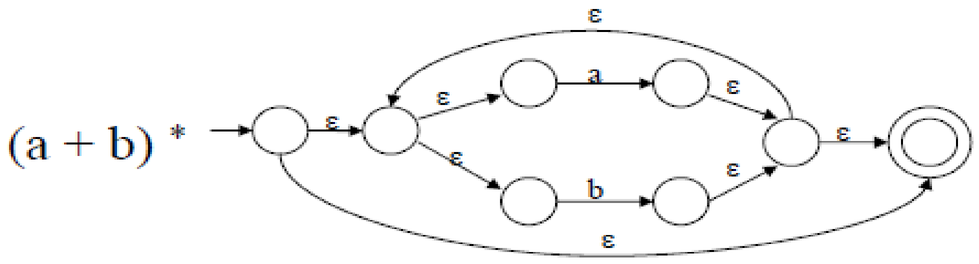
$N\ (r^*)$

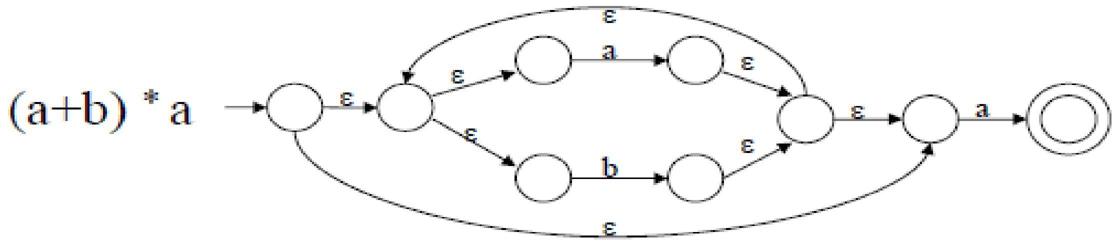
Using rule 1 and 2 we construct NFA's for each basic symbol in the expression, we combine these basic NFA using rule 3 to obtain the NFA for entire expression.

Example: - NFA construction of RE **(a + b) * a**









Conversion from NFA to DFA

Subset Construction Algorithm

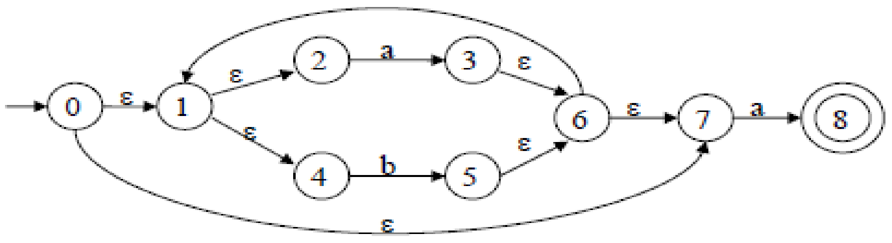
```
Put ε-closure (s0) as an unmarked state in to Dstates
    While there is an unmarked state T in Dstates do
        mark T
        for each input symbol a ∈ Σ do
            U = ε-closure (move (T, a))
            if U is not in Dstates then
                Add U as an unmarked state to Dstates
            end if
            Dtran[T, a] = U
        end do
    end do
```

The algorithm produces:

Dstates is the set of states of the new DFA consisting of sets of states of the NFA

Dtran is the transition table of the new DFA

Subset Construction Example (NFA to DFA) [(a+b)*a]



S_0 = ε-closure({0}) = {0,1,2,4,7} S_0 into *Dstates* as an unmarked state

⇓ mark S_0

ε-closure(move(S_0,a)) = ε-closure({3,8}) = {1,2,3,4,6,7,8} = S_1 S_1 into *Dstates*

ε-closure(move(S_0,b)) = ε-closure({5}) = {1,2,4,5,6,7} = S_2 S_2 into *Dstates*

Dtran[S_0,a] 🡐 S_1 *Dtran*[S_0,b] 🡐 S_2

⇓ mark S_1

ε-closure(move(S_1,a)) = ε-closure({3,8}) = {1,2,3,4,6,7,8} = S_1

ε-closure(move(S_1,b)) = ε-closure({5}) = {1,2,4,5,6,7} = S_2

Dtran [S_1,a] 🡐 S_1 *Dtran* [S_1,b] 🡐 S_2

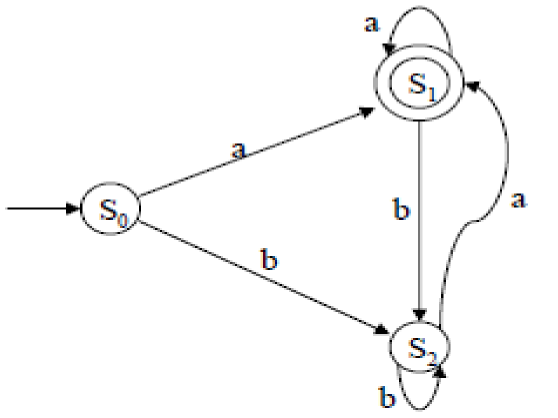
⇓ mark S_2

ε-closure(move(S_2,a)) = ε-closure({3,8}) = {1,2,3,4,6,7,8} = S_1

ε-closure(move(S_2,b)) = ε-closure({5}) = {1,2,4,5,6,7} = S_2

Dtran[S_2,a] 🡐 S_1 *Dtran*[S_2,b] 🡐 S_2

S0 is the start state of DFA since 0 is a member of S0= {0, 1, 2, 4, 7}
S1 is an accepting state of DFA since 8 is a member of S1 = {1, 2, 3, 4, 6, 7, 8}



This is final DFA

Exercise
Convert the following regular expression first into NFA and then into DFA

1. 0+ (1+0)*00
2. zero →0; one →1; bit → zero + one; bits → bit*
3. aa*+bb*
4. (a+b)*abb

Conversion from RE to DFA Directly

Important States

A state S of an NFA without ε- transition is called the important state if,

$$move(\{s\},a) \neq \varnothing$$

In an optimal state machine all states are important states

Augmented Regular Expression

When we construct an NFA from the regular expression then the final state of resulting NFA is not an important state because it has no transition. Thus to make important state of the accepting state of NFA we introduce an 'augmented' character (#) to a regular expression r.
This resulting regular expression is called the augmented regular expression of original expression r.

Conversion steps:

1. Augment the given regular expression by concatenating it with special symbol #
   I.e. r →(r) #
2. Create the syntax tree for this augmented regular expression
   In this syntax tree, all alphabet symbols (plus # and the empty string) in the augmented regular expression will be on the leaves, and all inner nodes will be the operators in that augmented regular expression.

3. Then each alphabet symbol (plus #) will be numbered (position numbers)
4. Traverse the tree to construct functions *nullable*, *firstpos*, *lastpos*, and *followpos*
5. Finally construct the DFA from the *followpos*

Rules for calculating nullable, firstpos and lastpos:

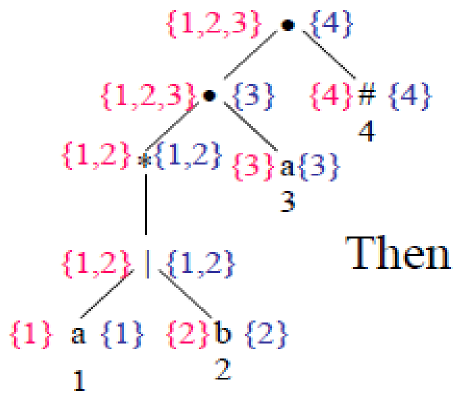
node n	nullable(n)	firstpos(n)	lastpos(n)
is leaf labeled ε	true	Φ	Φ
is leaf labeled with position i	false	{i} (position of leaf node)	{i}
n $\mid$ c_1 c_2	nullable(c_1) or nullable(c_2)	firstpos(c_1) ∪ firstpos(c_2)	lastpos(c_1) ∪ lastpos(c_2)
n • c_1 c_2	nullable(c_1) and nullable(c_2)	**if** (nullable(c_1)) **then** firstpos(c_1) ∪ firstpos(c_2) **else** firstpos(c_1)	**if** (nullable(c_2)) **then** lastpos(c_1) ∪ lastpos(c_2) **else** lastpos(c_2)
n * c_1	true	firstpos(c_1)	lastpos(c_1)

Algorithm to evaluate followpos

```
for each node n in the tree do
    if n is a cat-node with left child c1 and right child c2 then
        for each i in lastpos(c1) do
            followpos(i) := followpos(i) ∪ firstpos(c2)
        end do
    else if n is a star-node
        for each i in lastpos(n) do
            followpos(i) := followpos(i) ∪ firstpos(n)
        end do
    end if
end do
```

How to evaluate followpos: Example

For regular expression: **(a | b) * a #**



Then we can calculate followpos

followpos(1) = {1,2,3}
followpos(2) = {1,2,3}
followpos(3) = {4}
followpos(4) = {}

After we calculate follow positions, we are ready to create DFA for the regular expression.

Conversion from RE to DFA Example1

Note: - the start state of DFA is firstpos(root)

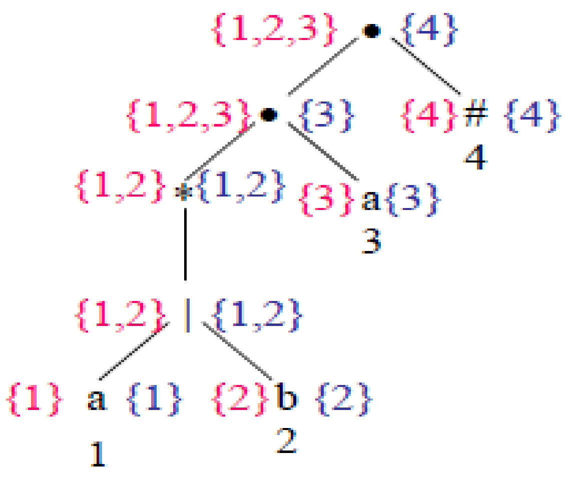
the accepting states of DFA are all states containing the position of #

Convert regular expression (a | b) * a into DFA

Its augmented regular expression is;

$(\underset{1}{a} \mid \underset{2}{b})^* \ \underset{3}{a} \ \underset{4}{\#}$

The syntax tree is:



Now we calculate followpos,

followpos(1)={1,2,3}
followpos(2)={1,2,3}
followpos(3)={4}
followpos(4)={}

S_1=firstpos(root)={1,2,3}

mark S_1

for a: followpos(1) $\cup$followpos(3)={1,2,3,4}=S_2 *move(S_1,a)=S_2*

for b: followpos(2)={1,2,3}=S_1 *move(S_1,b)=S_1*

mark S_2

for a: followpos(1) $\cup$followpos(3)={1,2,3,4}=S_2 *move(S_2,a)=S_2*

for b: followpos(2)={1,2,3}=S_1 *move(S_2,b)=S_1*

Now

start state: S_1

accepting states: {S_2}

Note:- Accepting states=states containing position of # ie 4.

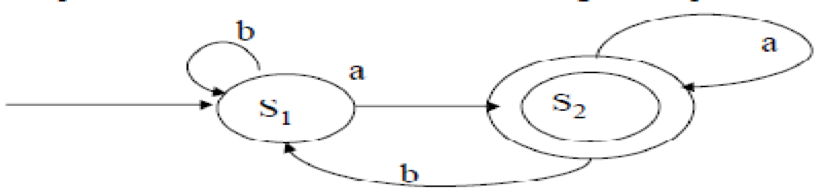


Fig: Resulting DFA of given regular expression

Conversion from RE to DFA

Example2

For RE---- (a | ε) b c* #
```
                  1       2 3 4
```
followpos(1)={2}

followpos(2)={3,4}

followpos(3)={3,4}

followpos(4)={}

S1=firstpos(root)={1,2}

$\downarrow$ *mark S1*

for a: followpos(1)={2}=S2 *move(S1,a)=S2*

for b: followpos(2)={3,4}=S3 *move(S1,b)=S3*

$\downarrow$ *mark S2*

for b: followpos(2)={3,4}=S3 *move(S2,b)=S3*

$\downarrow$ *mark S3*

for c: followpos(3)={3,4}=S3 *move(S3,c)=S3*

Start state: S1

Accepting states: {S3}

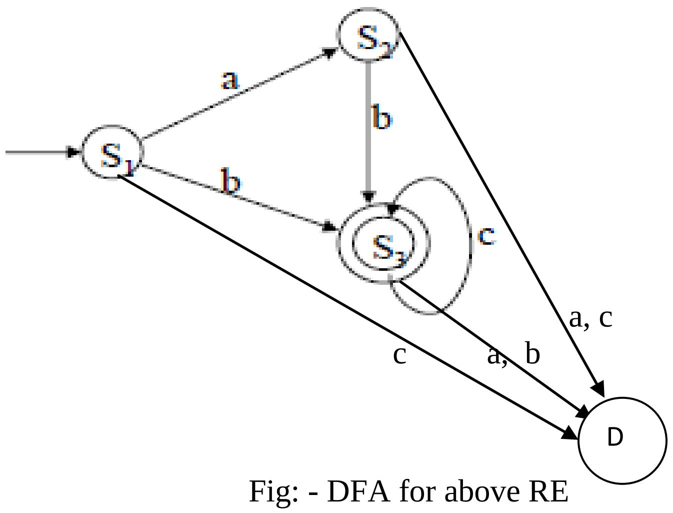


Fig: - DFA for above RE

State minimization in DFA

Partition the set of states into two groups:

– G1: set of accepting states

– G2: set of non-accepting states

For each new group G:

– partition G into subgroups such that states s1 and s2 are in the same group if for all input symbols a, states s1 and s2 have transitions to states in the same group.

Start state of the minimized DFA is the group containing the start state of the original DFA.

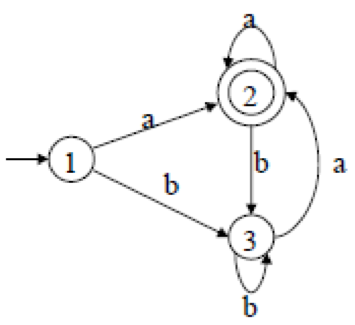
Accepting states of the minimized DFA are the groups containing the accepting states of the original DFA.

Procedure

1. So partition the set of states into two partition a) set of accepting states and b) set of non-accepting states.
2. Split the partition on the basis of distinguishable states and put equivalent states in a group
3. To split we process the transition from the states in a group with all input symbols. If the transition on any input from the states in a group is on different group of states then they are distinguishable so remove those states from the current partition and create groups.
4. Process until all the partition contains equivalent states only or has single state.

State Minimization in DFA

Example1:



$G_1 = \{2\}$

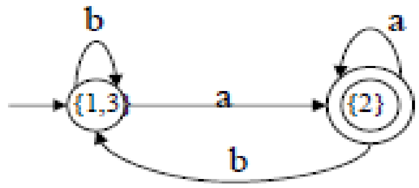
$G_2 = \{1,3\}$

G_2 cannot be partitioned because

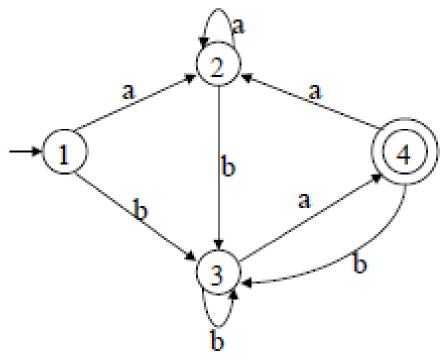
move(1,a)=2 move(1,b)=3

move(3,a)=2 move(2,b)=3

So, the minimized DFA (with minimum states)



Example 2:

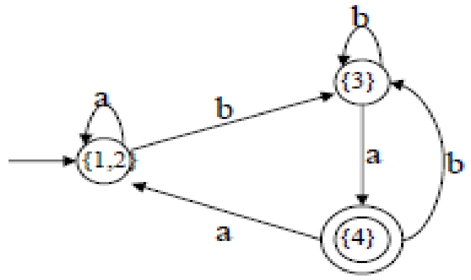


$$\begin{tikzcd}[column sep=small]
\text{Groups:} & \{1,2,3\} \arrow[dl] \arrow[dr] & \{4\} \\
\{1,2\} & & \{3\}
\end{tikzcd}$$

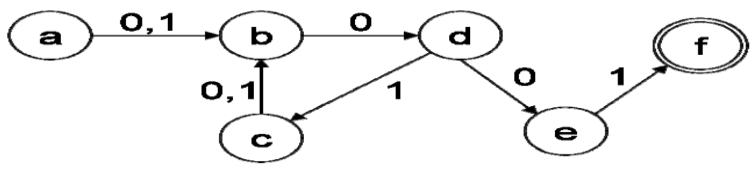
no more partitioning

a	b
1->2	1->3
2->2	2->3
3->4	3->3

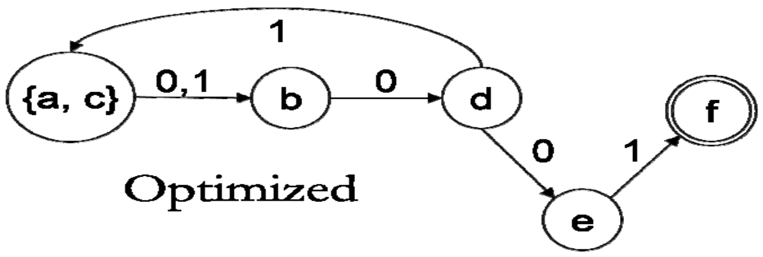
So minimized DFA is:



Example 3:

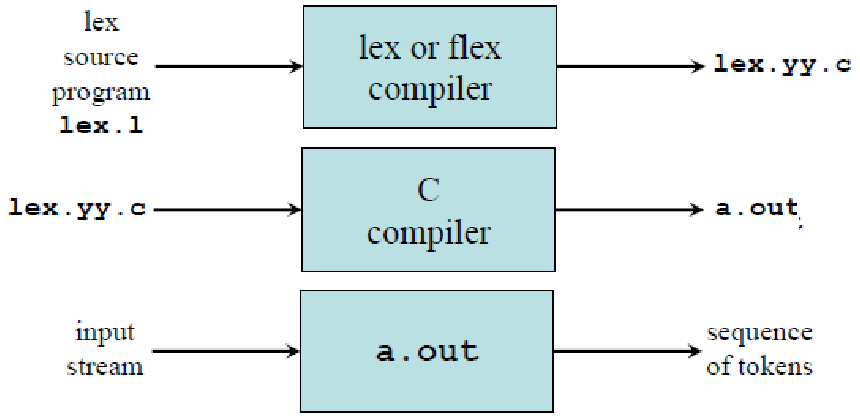


Partition 1: {{a, b, c, d, e}, {f}} with input 0; a→b and b→d, c→b, d→e all transatction in same group with input 1; e→f (different group) so e is distinguishable from others.
Partition 2: {{a, b, c, d}, {e}, {f}} with input 0; d→e
Partition 3: {{a, b, c}, {d}, {e}, {f}} with input 0; b→d
Partition 4: {{a, c}, {b}, {d}, {e}, {f}} with both 0 and 1 a, c→b so no split is possible here, a and c are equivalent.



Flex: Language for Lexical Analyzer

Systematically translate regular definitions into C source code for efficient scanning. Generated code is easy to integrate in C applications



Flex: An introduction

Flex is a tool for generating scanners. A scanner is a program which recognizes lexical patterns in text. The flex program reads the given input files, or its standard input if no file names are given, for a description of a scanner to generate. The description is in the form of pairs of regular expressions and C code, called rules. flex generates as output a C source file, 'lex.yy.c' by default, which defines a routine yylex(). This file can be compiled and linked with the flex runtime library to produce an executable. When the executable is run, it analyzes its input for occurrences of the regular expressions. Whenever it finds one, it executes the corresponding C code.

Flex specification:

A *flex specification* consists of three parts:
Regular definitions, C declarations in **%{ %}**
%%
Translation rules
%%
User-defined auxiliary procedures

The *translation rules* are of the form:

```
p1      {action1}
p2      {action2}
……………………..
pn      { actionn }
```

In all parts of the specification comments of the form **/* comment text */** are permitted.

Regular definitions:

It consist two things:'\'

– Any C code that is external to any function should be in %{ …….. %}
– Declaration of simple name definitions i.e specifying regular expression e.g

```
DIGIT      [0-9]
ID          [a-z][a-z0-9]*
```

The subsequent reference is as {DIGIT}, {DIGIT}+ or {DIGIT}*

Translation rules:

Contains a set of regular expressions and actions (C code) that are executed when the scanner matches the associated regular expression e.g

```
{ID}          printf("%s", getlogin());
```

Any code that follows a regular expression will be inserted at the appropriate place in the recognition procedure *yylex()*
Finally the user code section is simply copied to *lex.yy.c*

Practice

• Get familiar with FLEX
1. Try sample*.lex
2. Command Sequence:

```
flex sample*.lex
gcc lex.yy.c -lfl
./a.out
```

Flex operators and Meaning

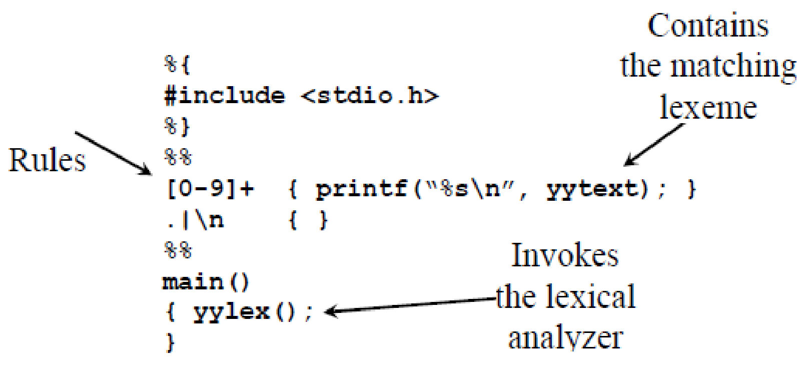
x	match the character **x**
\\.	match the character **.**
"*string*"	match contents of string of characters

.	match any character except newline
^	match beginning of a line
$	match the end of a line
[xyz]	match one character **x**, **y**, or **z** (use \ to escape **-**)
[^xyz]	match any character except **x**, **y**, and **z**
[a-z]	match one of **a** to **z**
*r****	closure (match zero or more occurrences)
r+	positive closure (match one or more occurrences)
*r***?**	optional (match zero or one occurrence)
*r*1*r*2	match *r*1 then *r*2 (concatenation)
*r*1|*r*2	match *r*1 or *r*2 (union)
(*r* **)**	grouping
*r*1\r2	match *r*1 when followed by *r*2
{*d***}**	match the regular expression defined by *d*
'r{2,5}'	anywhere from two to five r's
'r{2,}'	two or more r's
'r{4}'	exactly 4 r's

Flex Global Function, Variables & Directives

yylex() is the scanner function that can be invoked by the parser
yytext extern char *yytext; is a global char pointer holding the currently matched lexeme.
yyleng extern int yyleng; is a global int that contains the length of the currently matched lexeme.
ECHO copies yytext to the scanner's output
REJECT directs the scanner to proceed on to the "second best" rule which matched the input
yymore() tells the scanner that the next time it matches a rule, the corresponding token should be appended onto the current value of *yytext* rather than replacing it.
yyless(n) returns all but the first n characters of the current token back to the input stream, where they will be rescanned when the scanner looks for the next match
unput(c) puts the character c back onto the input stream. It will be the next character scanned
input() reads the next character from the input stream
YY_FLUSH_BUFFER flushes the scanner's internal buffer so that the next time the scanner attempts to match a token; it will first refill the buffer.

Flex Example1



```
%{
#include <stdio.h>
%}
%%
[0-9]+   { printf("%s\n", yytext); }
.|\n     { }
%%
main()
{ yylex();
}
```

Example2

```
/* Description: Count the number of characters and the number of lines
* from standard input
* Usage:
        (1) $ flex sample2.lex
        * (2) $ gcc lex.yy.c -lfl
        * (3) $ ./a.out
        * stdin> whatever you like
        * stdin> Ctrl-D
* Questions: Is it ok if we do not indent the first line?
* What will happen if we remove the second rule?
*/
        int num_lines = 0, num_chars = 0;
%%
        \n      ++num_lines; ++num_chars;
        .        ++num_chars;
%%
main()
{
        yylex();
        printf("# of lines = %d, # of chars = %d\n", num_lines, num_chars);
}
```

==

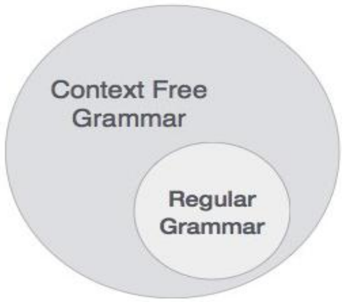
unit: 2

Syntax Analysis

Syntax analysis or parsing is the second phase of a compiler. In this chapter, we shall learn the basic concepts used in the construction of a parser.

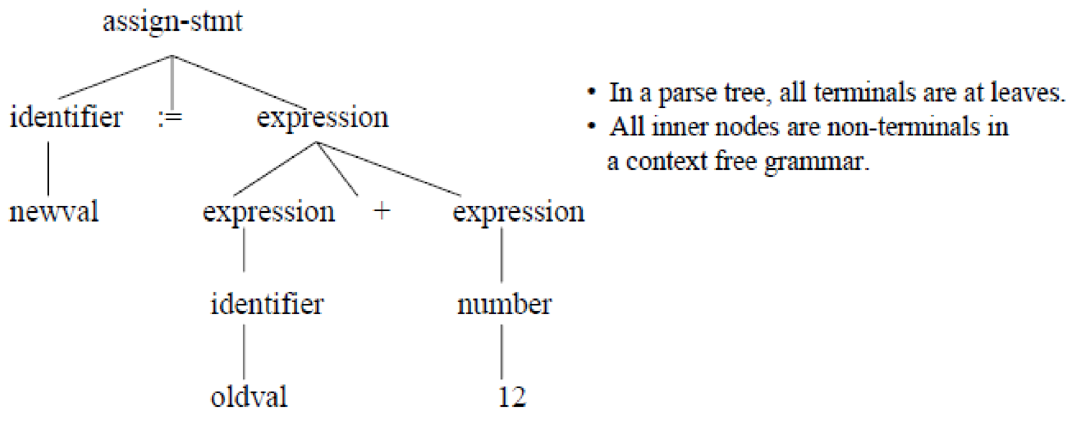
We have seen that a lexical analyzer can identify tokens with the help of regular expressions and pattern rules. But a lexical analyzer cannot check the syntax of a given sentence due to the limitations of the regular expressions. Regular expressions cannot check balancing tokens, such as parenthesis. Therefore, this phase uses context-free grammar (CFG), which is recognized by push-down automata.

CFG, on the other hand, is a superset of Regular Grammar, as depicted below:



A **Syntax Analyzer** creates the syntactic structure (generally a parse tree) of the given source program. Syntax analyzer is also called the **parser.** Its job is to analyze the source program based on the definition of its syntax. It works in lock-step with the lexical analyzer and is responsible for creating a parse-tree of the source code.

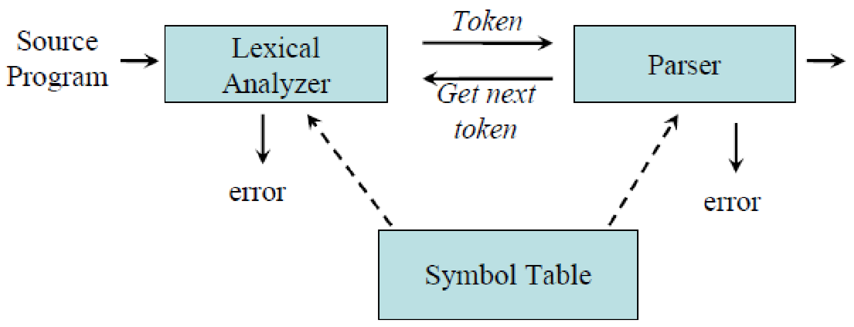
Ex: newval: = oldval + 12


assign-stmt
identifier
:=
expression
newval
expression
+
expression
identifier
number
oldval
12
• In a parse tree, all terminals are at leaves.
• All inner nodes are non-terminals in a context free grammar.

The syntax of a language is specified by a **context free grammar** (CFG).

The rules in a CFG are mostly recursive.

A syntax analyzer checks whether a given program satisfies the rules implied by a CFG or not.



Context-Free Grammars

Generally most of the programming languages have recursive structures that can be defined by

Context free grammar (CFG).Context-free grammar is a 4-tuple $G = (N, T, P, S)$ where

- T is a finite set of tokens (*terminal* symbols)
- N is a finite set of *non-terminals*
- P is a finite set of *productions* of the form

$$\alpha \rightarrow \beta$$

Where,

$$\alpha \in (N \cup T)^* \, N \, (N \cup T)^* \text{ and } \beta \in (N \cup T)^*$$

- $S \in N$ is a designated *start symbol*

Programming languages usually have recursive structures that can be defined by a context-free grammar (CFG).

CFG: Notational Conventions

Terminals are denoted by lower-case letters and symbols (single atoms) and **bold** strings (tokens)

$a, b, c, \ldots \in T$

specific terminals:

0, **1**, **id**, +

Non-terminals are denoted by *lower-case italicized* letters or upper-case letters symbols

$A, B, C \ldots \in N$

specific non-terminals:

expr, *term*, *stmt*

Production rules are of the form

$A \rightarrow \alpha$:, that is read as "A can produce α"

Strings comprising of both terminals and non-terminals are denoted by greek letters

α , β , etc

Derivations

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

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

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

Left-most derivation

If the sentential form of an input is scanned and replaced from left to right, it is called left-most derivation.

Right-most derivation

If we scan and replace the input with production rules, from right to left, it is known as right-most derivation.

Example

Production rules:

E → E + E

E → E * E

E → id

Input string: id + id * id

The left-most derivation is:

E → E * E

E → E + E * E

E → id + E * E

E → id + id * E

E → id + id * id

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

The right-most derivation is:

E → E + E

E → E + E * E

E → E + E * id

E → E + id * id

E → id + id * id

Parse Trees

A parse tree is a graphical depiction of a derivation. It is convenient to see how strings are derived from the start symbol. The start symbol of the derivation becomes the root of the parse tree. Let us see this by an example from the last topic.

We take the left-most derivation of a + b * c

The left-most derivation is:

E → E * E

E → E + E * E

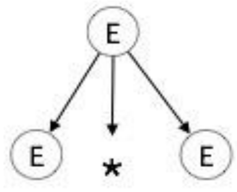
E → id + E * E

E → id + id * E

E → id + id * id

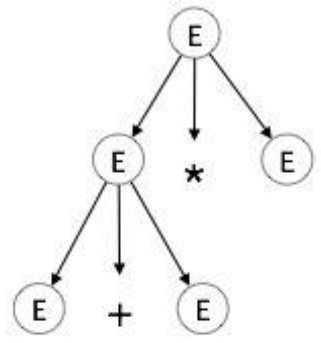
Step 1:

E → E * E

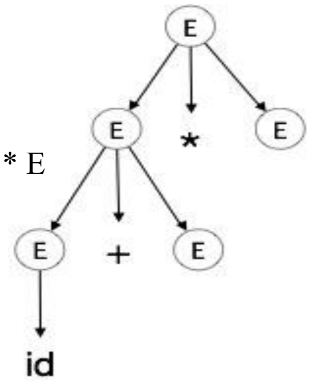


Step 2:

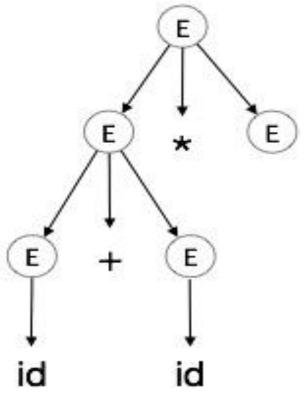
E → E + E * E



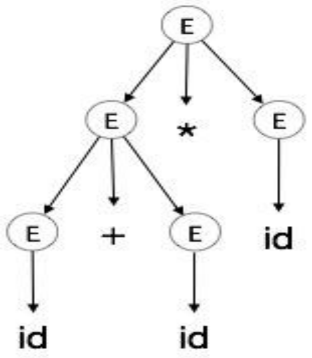
Step 3:
E → id + E * E



Step 4:
E → id + id * E



Step 5:
E → id + id * id



In a parse tree:

- All leaf nodes are terminals.
- All interior nodes are non-terminals.
- In-order traversal gives original input string.

A parse tree depicts associativity and precedence of operators. The deepest sub-tree is traversed first; therefore the operator in that sub-tree gets precedence over the operator which is in the parent nodes.

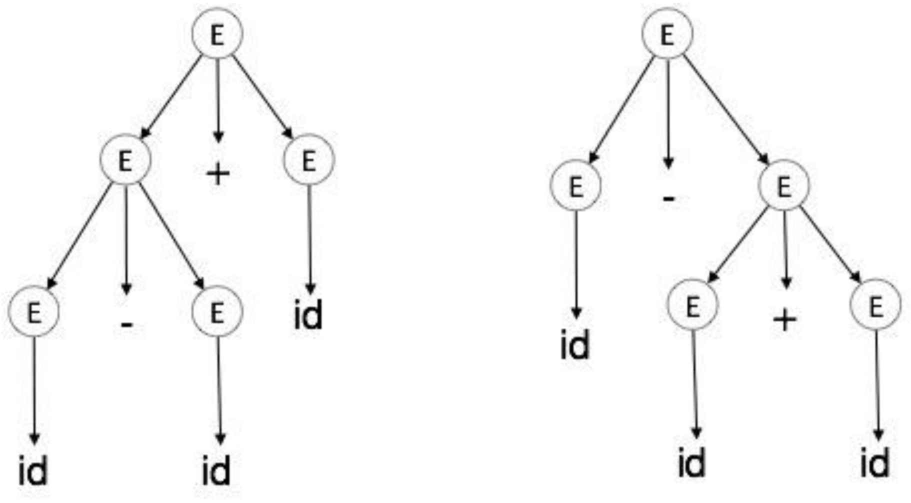
Ambiguity of a grammar

A grammar G is said to be ambiguous if it has more than one parse tree (left or right derivation) for at least one string. Also A grammar G is said to be ambiguous if there is a string w∈L(G) for which we can construct more than one parse tree rooted at start symbol of the production.

Example

E → E + E
E → E – E
E → id

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



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

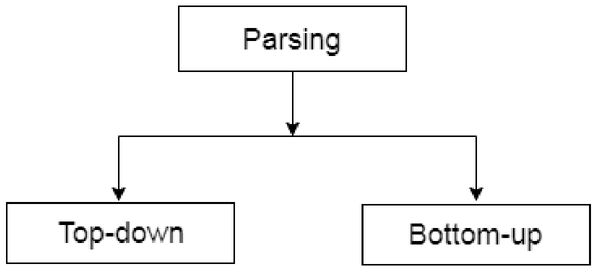
Example 2: let us consider a CFG:

$E \to E + E|E * E|(E)|-E|id$ Test whether this grammar is ambiguous or not.

Parsing

Parser is a compiler that is used to break the data into smaller elements coming from lexical analysis phase. A parser takes input in the form of sequence of tokens and produces output in the form of parse tree. Parsing is of two types: top down parsing and bottom up parsing.

Given a stream of input tokens, *parsing* involves the process of reducing them to a non-terminal. Parsing can be either *top-down* or *bottom-up*.



Top-down parsing involves generating the string starting from the first non-terminal and repeatedly applying production rules.

Bottom-up parsing involves repeatedly rewriting the input string until it ends up in the first non-terminal of the grammar.

Top-Down Parsing

The parse tree is created top to bottom.

Top-down parser

– Recursive-Descent Parsing
– Predictive Parsing

Recursive-Descent Parsing

• Backtracking is needed (If a choice of a production rule does not work, we backtrack to try other alternatives.)
• It is a general parsing technique, but not widely used.
• Not efficient

It tries to find the left-most derivation

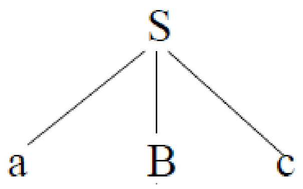
Example: Consider the grammar,

$S \to aBc$

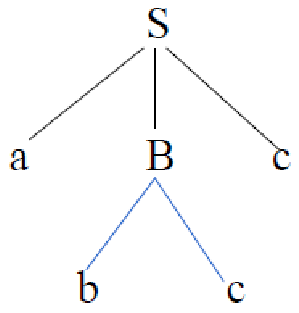
$B \to bc|b$

and the input string "abc" parsed using Recursive-Descent parsing.

Step 1: The first rule $S \to aBc$ to parse S

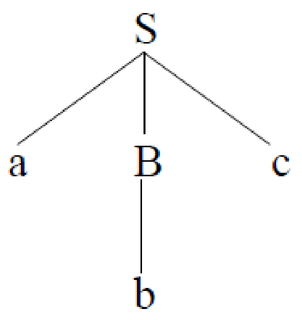


Step 2: The next non-terminal is B and is parsed using production $B \to bc$ as,



Fails

Step 3: Which is false and now backtrack and use production $B \to b$ to parse for B



Method: let input w = abc, initially create the tree of single node S. The left most node a match the first symbol of w, so advance the pointer to b and consider the next leaf B. Then expand B using first choice bc. There is match for b and c, and advanced to the leaf symbol c of S, but there is no match in input, report failure and go back to B to find another alternative b that produce match.

Left Recursion

A grammar becomes left-recursive if it has any non-terminal 'A' whose derivation contains 'A' itself as the left-most symbol. Left-recursive grammar is considered to be a problematic situation for top-down parsers. Top-down parsers start parsing from the Start symbol, which in itself is non-terminal. So, when the parser encounters the same non-terminal in its derivation, it becomes hard for it to judge when to stop parsing the left non-terminal and it goes into an infinite loop.

A grammar is ***left recursive*** if it has a non-terminal A such that there is a derivation.
$A \rightarrow A\alpha$ For some string α
Top-down parsing techniques **cannot** handle left-recursive grammars.
So, we have to convert our left-recursive grammar into an equivalent grammar which is not left-recursive.

Example: Immediate Left-Recursion

$$A \rightarrow A\alpha \mid \beta$$

$$\Downarrow \quad \text{Eliminate immediate left recursion}$$

$$A \rightarrow \beta A'$$
$$A' \rightarrow \alpha A' \mid \epsilon$$

In general,

$$A \rightarrow A\,\alpha_1 \mid \dots \mid A\,\alpha_m \mid \beta_1 \mid \dots \mid \beta_n \qquad \text{where } \beta_1 \dots \beta_n \text{ do not start with A}$$

$$\Downarrow \qquad \text{eliminate immediate left recursion}$$

$$A \rightarrow \beta_1 A' \mid \dots \mid \beta_n A'$$
$$A' \rightarrow \alpha_1 A' \mid \dots \mid \alpha_m A' \mid \varepsilon \qquad \text{an equivalent grammar}$$

Immediate Left-Recursion - Example

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow id \mid (E)$

$\Downarrow$ eliminate immediate left recursion

$E \rightarrow T\,E'$

$E' \rightarrow +T\,E' \mid \varepsilon$

$T \rightarrow F\,T'$

$T' \rightarrow *F\,T' \mid \varepsilon$

$F \rightarrow id \mid (E)$

Non-Immediate Left-Recursion

By just eliminating the immediate left-recursion, we may not get a grammar which is not left-recursive.

$S \rightarrow Aa \mid b$

$A \rightarrow Sc \mid d$ This grammar is not immediately left-recursive, but it is still left-recursive.

$\underline{S} \Rightarrow Aa \Rightarrow \underline{S}ca$ or

$\underline{A} \Rightarrow Sc \Rightarrow \underline{A}ac$ causes to a left-recursion

So, we have to eliminate all left-recursions from our grammar

So, the resulting equivalent grammar which is not left-recursive is:

$S \rightarrow Aa \mid b$

$A \rightarrow bdA' \mid fA'$

$A' \rightarrow cA' \mid adA' \mid \varepsilon$

Left-Factoring

If more than one grammar production rules have a common prefix string, then the top-down parser cannot make a choice as to which of the production it should take to parse the string in hand.

When a no terminal has two or more productions whose right-hand sides start with the same grammar symbols, then such a grammar is not LL(1) and cannot be used for predictive parsing. This grammar is called left factoring grammar.

Eg:

Replace productions

$A \rightarrow \alpha\,\beta_1 \mid \alpha\,\beta_2 \mid \ldots \mid \alpha\,\beta_n \mid \gamma$

with

$A \rightarrow \alpha\,A' \mid \gamma$

$A' \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$

Hint: taking α common from the each production.

Example: Eliminate left factorial from following grammar:

S→iEiS | iEiSiS | a
B→b

Solution: **S→iEiSS' | a**
S'→iS
B→b

Predictive Parsing

A *predictive parser* tries to predict which production produces the least chances of a backtracking and infinite looping.

When re-writing a non-terminal in a derivation step, a predictive parser can uniquely choose a production rule by just looking the current symbol in the input string.

Two variants:

– Recursive (recursive-descent parsing)
– Non-recursive (table-driven parsing)

Non-Recursive Predictive Parsing

Non-Recursive predictive parsing is a table-driven parser.

Given an LL(1) grammar $G = (N, T, P, S)$ construct a table $M[A,a]$ for $A \in N$, $a \in T$ and use a *driver program* with a *stack.*

A table driven predictive parser has an input buffer, a stack, a parsing table and an output stream.

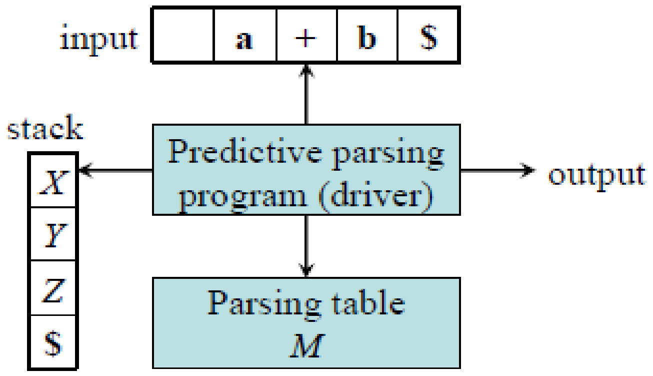


Fig model of a non-recursive predictive parser

Input buffer:

It contains the string to be parsed followed by a special symbol $.

Stack:

A stack contains a sequence of grammar symbols with $ on the bottom. Initially it contains the symbol $.

Parsing table:

It is a two dimensional array M [A, a] where 'A' is non-terminal and 'a' is a terminal symbol.

Output stream:

A production rule representing a step of the derivation sequence of the string in the input buffer.

Algorithm

Input : a string w.

Output: if w is in L(G), a leftmost derivation of w; otherwise error

```
1. Set ip to the first symbol of input stream
2. Set the stack to $S where S is the start symbol of the grammar
3. repeat
   Let X be the top stack symbol and a be the symbol pointed by ip
   If X is a terminal or $ then
         if X = a then pop X from the stack and advance ip
         else error()
   else            /* X is a non-terminal */
         if M[X, a] = X → Y1, Y2, ..., ..., Yk  then
                   pop X from stack
                   push Yk, Yk-1, ..., ..., Y1 onto stack (with Y1 on top)
                   output the production X → Y1, Y2, ...,  Yk
         else error()
4. until X = $       /* stack is empty */
```

Example: Given a grammar,

$S \rightarrow aBa$

$B \rightarrow bB \mid \varepsilon$

Input: abba

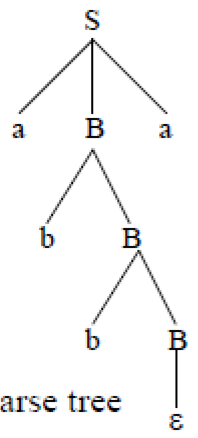
<u>stack</u>	<u>input</u>	<u>output</u>
$S	abba$	$S \rightarrow aBa$
$aBa	abba$	
$aB	bba$	$B \rightarrow bB$
$aBb	bba$	
$aB	ba$	$B \rightarrow bB$
$aBb	ba$	
$aB	a$	$B \rightarrow \varepsilon$
$a	a$	
$	$	accept, successful completion

	a	b	$
S	$S \rightarrow aBa$		
B	$B \rightarrow \varepsilon$	$B \rightarrow bB$	

LL(1) Parsing Table

Outputs: $S \rightarrow aBa$ $\quad B \rightarrow bB$ $\quad B \rightarrow bB$ $\quad B \rightarrow \varepsilon$

Derivation(left-most): $S \Rightarrow aBa \Rightarrow abBa \Rightarrow abbBa \Rightarrow abba$



parse tree

Constructing LL(1) Parsing Tables

- Eliminate left recursion from grammar
- Eliminate left factor of the grammar

$$\begin{tikzcd}\text{a grammar} \arrow[r, "\text{eliminate left recursion}"'] & {} \arrow[r, "\text{eliminate left factor}"'] & \text{a grammar suitable for predictive parsing (a LL(1) grammar)}\end{tikzcd}$$

To compute LL (1) parsing table, at first we need to compute FIRST and FOLLW functions.

Compute FIRST

FIRST(α) is a set of the terminal symbols which occur as first symbols in strings derived from α where α is any string of grammar symbols.

If α derives to ϵ, then ϵ is also in FIRST (α).

Algorithm for calculating First set

Look at the definition of FIRST(α) set:

- if α is a terminal, then FIRST(α) = { α }.
- if α is a non-terminal and α → Ɛ is a production, then FIRST(α) = { Ɛ }.
- if α is a non-terminal and α → $\boldsymbol{\gamma}1\ \boldsymbol{\gamma}2\ \boldsymbol{\gamma}3 \ldots \boldsymbol{\gamma}n$ and any FIRST($\boldsymbol{\gamma}$) contains t then t is in FIRST(α).

Example:

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

FIRST(F) = { (,id}	FIRST(TE') = { (,id}
FIRST(T') = {*, ε}	FIRST(+TE') = {+}
FIRST(T) = { (,id}	FIRST(ε) = {ε}
FIRST(E') = {+, ε}	FIRST(FT') = { (,id}
FIRST(E) = { (,id}	FIRST(*FT') = {*}
	FIRST(ε) = {ε}
	FIRST((E)) = { (}
	FIRST(id) = {id}

Compute FOLLOW

FOLLOW (A) is the set of the terminals which occur immediately after (follow) the *non-terminal A* in the strings derived from the starting symbol.

– a terminal a is in FOLLOW(A) if $S \overset{*}{\Rightarrow} \alpha A a \beta$

– \$ is in FOLLOW(A) if $S \overset{*}{\Rightarrow} \alpha A$

Algorithm for calculating Follow set:

- if α is a start symbol, then FOLLOW() = \$
- if α is a non-terminal and has a production α → AB, then FIRST(B) is in FOLLOW(A) except Ɛ.
- if α is a non-terminal and has a production α → AB, where B Ɛ, then FOLLOW(A) is in FOLLOW(α).

Compute FOLLOW: Example

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

FOLLOW(E) = { \$,) }
FOLLOW(E') = { \$,) }
FOLLOW(T) = { +,), \$ }
FOLLOW(T') = { +,), \$ }
FOLLOW(F) = {+, *,), \$ }

Constructing LL(1) Parsing Tables

If we can always choose a production uniquely by using FIRST and FOLLOW functions then this is called LL(1) parsing where the first L indicates the reading direction (Left-to –right) and second L indicates the derivation order (left) and 1 indicates that there is a one-symbol look-ahead. The grammar that can be parsed using LL(1) parsing is called an LL(1) grammar.

Algorithm

```
Input: LL(1) Grammar G
Output: Parsing Table M
for each production rule $A \to \alpha$ of a grammar G
    for each terminal a in FIRST($\alpha$)
            add $A \to \alpha$ to M[A,a]
            If $\varepsilon$ in FIRST($\alpha$) then
                    for each terminal a in FOLLOW(A)
                            add $A \to \alpha$ to M[A,a]
            If $\varepsilon$ in FIRST($\alpha$) and $ in FOLLOW(A) then
                    add $A \to \alpha$ to M[A,$]
```

Constructing LL(1) Parsing Tables: Example1

$E \to T E'$

$E' \to + T E' \mid \varepsilon$

$T \to F T'$

$T' \to * F T' \mid \varepsilon$

$F \to (E) \mid id$

Solution:

FIRST(F) = { (,id}
FIRST(T') = {*, ε}
FIRST(T) = { (,id}
FIRST(E') = {+, ε}
FIRST(E) = { (,id}
FIRST(TE') = { (,id}
FIRST(+TE') = {+}
FIRST(ε) = {ε}
FIRST(FT') = { (,id}
FIRST(*FT') = {*}
FIRST(ε) = {ε}
FIRST((E)) = { (}
FIRST(id) = {id}

FOLLOW(E)={$, FIRST (')')}={$,)}
FOLLOW(E')={ FOLLOW(E)}={$,)}
FOLLOW(T)={FIRST(E'), FOLLOW(E')}={+,), $}
FOLLOW(T')={ FOLLOW(T)}={+,), $}
FOLLOW(F)={ FOLLOW(T'),FIRST(T')except ? }={+,), $, *}

Non-terminals	Terminal Symbols					
	+	*	(	)	id	$
E			$E \to TE'$		$E \to TE'$	

E’	$E' \rightarrow +TE'$			$E' \rightarrow \varepsilon$		$E' \rightarrow \varepsilon$
T			$T \rightarrow FT'$		$T \rightarrow FT'$	
T’	$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \varepsilon$		$T' \rightarrow \varepsilon$
F			$F \rightarrow (E)$		$F \rightarrow id$	

Constructing LL(1) Parsing Tables: Example2

$S \rightarrow iEtSS' | a$
$S' \rightarrow eS | \in$
$E \rightarrow b$

Construct LL(1) parsing table for this grammar.
Solution:

FIRST(S)={i, a}	FOLLOW(S)={ FIRST(S’)}={e, $}
FIRST(S’)={e, ∈ }	FOLLOW(S’)={FOLLOW(S)}={e, $}
FIRST(E)={b}	FOLLOW(E)={FIRST(tSS’)}={t}
FIRST(iEtSS’)={i}	FIRST(b)={b}
FIRST(a)={a}	
FIRST(eS)={e}	
FIRST(∈)={∈ }	Construct table itself.

[Q] Produce the predictive parsing table for [HW]

a. $S \rightarrow 0\,S\,1 \mid 0\,1$
b. The prefix grammar $S \rightarrow +\,S\,S \mid *\,S\,S \mid a$

LL(1) Grammars

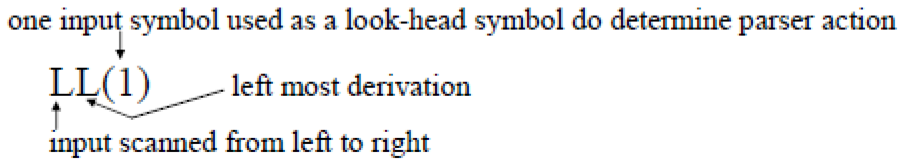
A grammar whose parsing table has no multiply-defined entries is said to be LL(1) grammar.

What happen when a parsing table contains multiply defined entries?

– The problem is ambiguity

A left recursive, not left factored and ambiguous grammar cannot be a LL(1) grammar (i.e. left recursive, not left factored and ambiguous grammar may have multiply – defined entries in parsing table)

Properties of LL(1) Grammars


one input symbol used as a look-head symbol do determine parser action
LL(1)
left most derivation
input scanned from left to right

A grammar G is LL(1) if and only if the following conditions hold for two distinctive production rules $A \rightarrow \alpha$ and $A \rightarrow \beta$

1. Both α and β cannot derive strings starting with same terminals.
2. At most one of α and β can derive to ε.
3. If β can derive to ε, then α cannot derive to any string starting with a terminal in FOLLOW(A).

Exercise:
Q. For the grammar,
$S \rightarrow [C]S \mid \epsilon$
$C \rightarrow \{A\}C \mid \epsilon$
$A \rightarrow A(\,) \mid \epsilon$ Construct the predictive top down parsing table (LL (1) parsing table)

Conflict in LL (1):

When a single symbol allows several choices of production for non-terminal N then we say that there is a conflict on that symbol for that non-terminal.
Example: Show that given grammar is not LL(1).
$S \rightarrow aA \mid bAc$
$A \rightarrow c \mid \varepsilon$
Solution:

FIRST(S)={a, b} FIRST(A)={C, ε}
FIRST(aA)={a} FIRST(bAc)={b} FIRST(c)={c} FIRST(ε)={ε}

FOLLOW(S)={ $ }
FOLLOW(A)={$, c }

	a	b	c	$
S	S→aA	S→bAc		
A			A →c A →ε	A →ε

A conflict emerges when the parser gets c token to which it does not know the rule to apply. So this grammar is not LL(1).

Bottom-Up Parsing

Bottom up parsing is used to construct a parse tree for an input string. In the bottom up parsing, the parsing starts with the input symbol and construct the parse tree up to the start symbol by tracing out the rightmost derivations of string in reverse. Bottom up parsing is classified in to various parsing. These are as follows:

1. Shift-Reduce Parsing
2. Operator Precedence Parsing
3. Table Driven LR Parsing
   a. LR(1)
   b. SLR(0)
   c. CLR (1)
   d. LALR(1)

Reduction

The process of replacing a substring by a non-terminal in bottom-up parsing is called reduction. It is a reverse process of production.

Eg: $S \rightarrow aA$

Here, if replacing aA by S then such a grammar is called reduction.

Shift-Reduce Parsing

The process of reducing the given input string into the starting symbol is called shift-reduce parsing.

$$\text{A string} \xrightarrow[\text{Reduced to}]{} \text{the starting symbol}$$

Example:

$$S \rightarrow aABb$$
$$A \rightarrow aA \mid a$$
$$B \rightarrow bB \mid b$$

$$S \underset{rm}{\Rightarrow} aABb \underset{rm}{\Rightarrow} aAbb \underset{rm}{\Rightarrow} aaAbb \underset{rm}{\Rightarrow} aaabb$$

input string: aaabb
aaAbb
aAbb $\Downarrow$ reduction
aABb
S

Handle

A substring that can be replaced by a non-terminal when it matches its right sentential form is called a **handle**.

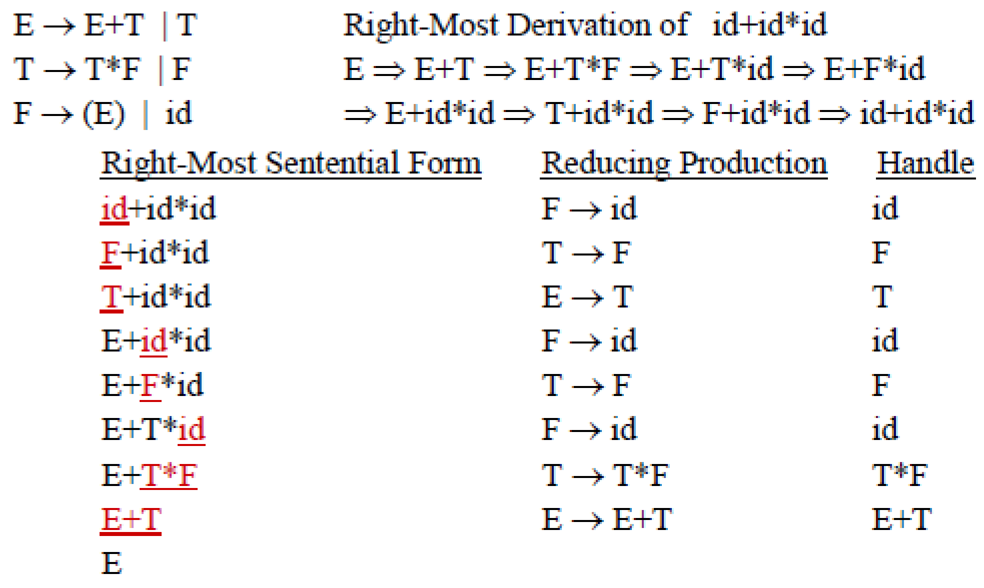
If the grammar is unambiguous, then every right-sentential form of the grammar has exactly one handle.

Example 1: Let's take a grammar

$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow (E) \mid id$

Right Sentential Form	Handle	Reducing Production
id1 * id2	id1	$F \rightarrow id$
F * id2	F	$T \rightarrow F$
T * id2	id2	$F \rightarrow id$
T * F	T * F	$E \rightarrow T * F$

Example 2: A Shift-Reduce Parser with Handle



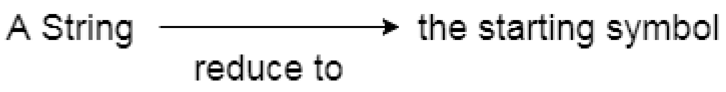
$E \rightarrow E+T \mid T$
$T \rightarrow T*F \mid F$
$F \rightarrow (E) \mid id$

Right-Most Derivation of id+id*id
$E \Rightarrow E+T \Rightarrow E+T*F \Rightarrow E+T*id \Rightarrow E+F*id$
$\Rightarrow E+id*id \Rightarrow T+id*id \Rightarrow F+id*id \Rightarrow id+id*id$

Right-Most Sentential Form	Reducing Production	Handle
$\underline{id}+id*id$	$F \rightarrow id$	id
$\underline{F}+id*id$	$T \rightarrow F$	F
$\underline{T}+id*id$	$E \rightarrow T$	T
$E+\underline{id}*id$	$F \rightarrow id$	id
$E+\underline{F}*id$	$T \rightarrow F$	F
$E+T*\underline{id}$	$F \rightarrow id$	id
$E+\underline{T*F}$	$T \rightarrow T*F$	T*F
$\underline{E+T}$	$E \rightarrow E+T$	E+T
E		

Stack Implementation of Shift-Reduce Parser

- Shift reduce parsing is a process of reducing a string to the start symbol of a grammar.
- Shift reduce parsing uses a stack to hold the grammar and an input tape to hold the string.
- Sift reduce parsing performs the two actions: shift and reduce. That's why it is known as shift reduces parsing.
- At the shift action, the current symbol in the input string is pushed to a stack.
- At each reduction, the symbols will replaced by the non-terminals. The symbol is the right side of the production and non-terminal is the left side of the production.



The stack holds the grammar symbols and input buffer holds the string w to be parsed.

```
1. Initially stack contains only the sentinel $, and input buffer contains the input string w$.
2. While stack not equal to $S or not error and input not $ do
        (a) While there is no handle at the top of stack, do shift input buffer and push the symbol
        onto stack
        (b) If there is a handle on top of stack, then pop the handle and reduce the handle with its
        non-terminal and push it onto stack
3. Done
```

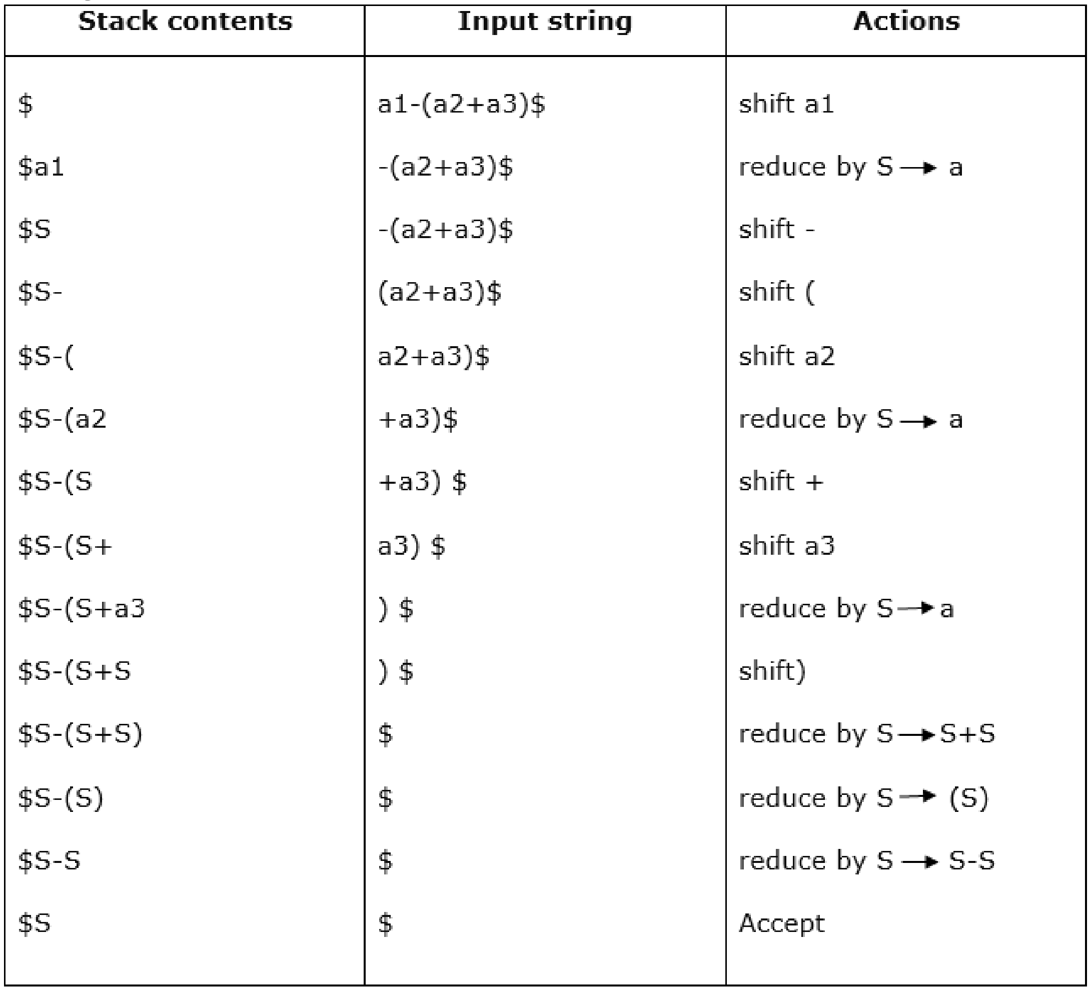
<u>Parser Actions:</u>

1. Shift: The next input symbol is shifted onto the top of the stack.
2. Reduce: Replace the handle on the top of the stack by the non-terminal.
3. Accept: Successful completion of parsing.
4. Error: Parser discovers a syntax error, and calls an error recovery routine

Example 1: Use the following grammar

$S \rightarrow S+S$
$S \rightarrow S-S$
$S \rightarrow (S)$
$S \rightarrow a$

Input string: a1-(a2+a3)
Parsing Table:



Stack contents	Input string	Actions
$	a1-(a2+a3)$	shift a1
$a1	-(a2+a3)$	reduce by S → a
$S	-(a2+a3)$	shift -
$S-	(a2+a3)$	shift (
$S-(	a2+a3)$	shift a2
$S-(a2	+a3)$	reduce by S → a
$S-(S	+a3) $	shift +
$S-(S+	a3) $	shift a3
$S-(S+a3	) $	reduce by S→a
$S-(S+S	) $	shift)
$S-(S+S)	$	reduce by S→S+S
$S-(S)	$	reduce by S → (S)
$S-S	$	reduce by S → S-S
$S	$	Accept

There are two main categories of shift reduce parsing as follows:

1. Operator-Precedence Parsing
2. LR-Parser

Operator precedence parsing

Operator precedence grammar is kinds of shift reduce parsing method. It is applied to a small class of operator grammars. A grammar is said to be operator precedence grammar if it has two properties:

1. No R.H.S. of any production has a∈.
2. No two non-terminals are adjacent.

Operator precedence can only established between the terminals of the grammar. It ignores the non-terminal. There are the three operator precedence relations:

1. a ⋗ b means that terminal "a" has the higher precedence than terminal "b".

2. a ⋖ b means that terminal "a" has the lower precedence than terminal "b".
3. a ≐ b means that the terminal "a" and "b" both have same precedence.

Precedence table:

	+	*	(	)	id	$
+	⋗	⋖	⋖	⋗	⋖	⋗
*	⋗	⋗	⋖	⋗	⋖	⋗
(	⋖	⋖	⋖	≐	⋖	X
)	⋗	⋗	X	⋗	X	⋗
id	⋗	⋗	X	⋗	X	⋗
$	⋖	⋖	⋖	X	⋖	X

Parsing Action

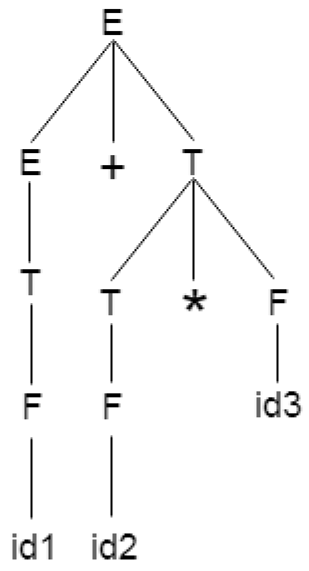
- Both end of the given input string, add the $ symbol.
- Now scan the input string from left right until the ⋗ is encountered.
- Scan towards left over all the equal precedence until the first left most ⋖ is encountered.
- Everything between left most ⋖ and right most ⋗ is a handle.
- $ on $ means parsing is successful.

Example: Let's take a grammar:

$E \rightarrow E+T/T$
$T \rightarrow T*F/F$
$F \rightarrow id$

Given string: w = id + id * id

Let us consider a parse tree for it as follows:



On the basis of above tree, we can design following operator precedence table

	E	T	F	id	+	*	$
E	X	X	X	X	≐	X	⋗
T	X	X	X	X	⋗	≐	⋗
F	X	X	X	X	⋗	⋗	⋗
id	X	X	X	X	⋗	⋗	⋗
+	X	≐	⋖	⋖	X	X	X
*	X	X	≐	⋖	X	X	X
$	⋖	⋖	⋖	⋖	X	X	X

Now let us process the string with the help of the above precedence table:

$ ⋖ id1 ⋗ + id2 * id3 $

$ ⋖ F ⋗ + id2 * id3 $

$ ⋖ T ⋗ + id2 * id3 $

$ ⋖ E ≐ + ⋖ id2 ⋗ * id3 $

$ ⋖ E ≐ + ⋖ F ⋗ * id3 $

$ ⋖ E ≐ + ⋖ T ≐ * ⋖ id3 ⋗ $

$ ⋖ E ≐ + ⋖ T ≐ * ≐ F ⋗ $

$ ⋖ E ≐ + ≐ T ⋗ $

$ ⋖ E ≐ + ≐ T ⋗ $

$ ⋖ E ⋗ $

Accept.

Conflicts in Shift-Reduce Parsing

Some grammars cannot be parsed using shift-reduce parsing and result in *conflicts*. There are two kinds of shift-reduce conflicts:

shift/reduce conflict:

Here, the parser is not able to decide whether to shift or to reduce.

Example:

A $\rightarrow$ ab | abcd

the stack contains \$ab, and

the input buffer contains cd\$, the parser cannot decide whether to reduce \$ab to \$A or to shift two more symbols before reducing.

reduce/reduce conflict:

Here, the parser cannot decide which sentential form to use for reduction.

For example

A $\rightarrow$ bc

B $\rightarrow$abc and the stack contains \$abc, the parser cannot decide whether to reduce it to \$aA or to \$B.

LR Parser

LR parsing is one type of bottom up parsing. It is used to parse the large class of grammars. In the LR parsing, "L" stands for left-to-right scanning of the input. "R" stands for constructing a right most derivation in reverse. "K" is the number of input symbols of the look ahead used to make number of parsing decision. LR parsing is divided into four parts: LR (0) parsing, SLR parsing, CLR parsing and LALR parsing.

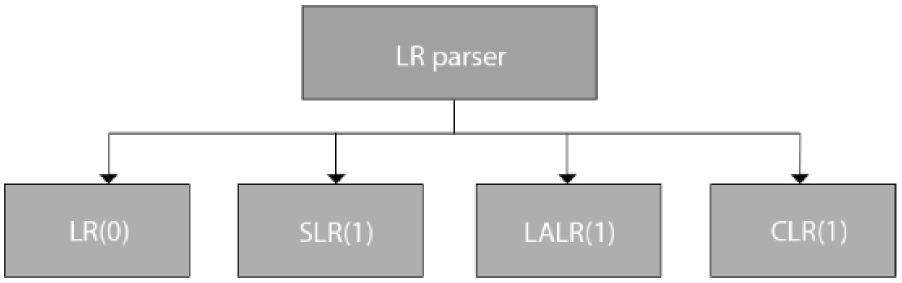
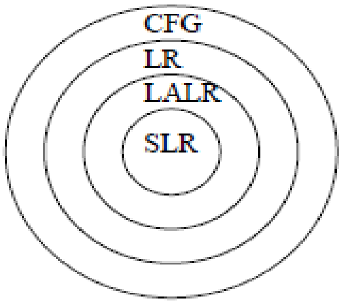


Fig: Types of LR parser

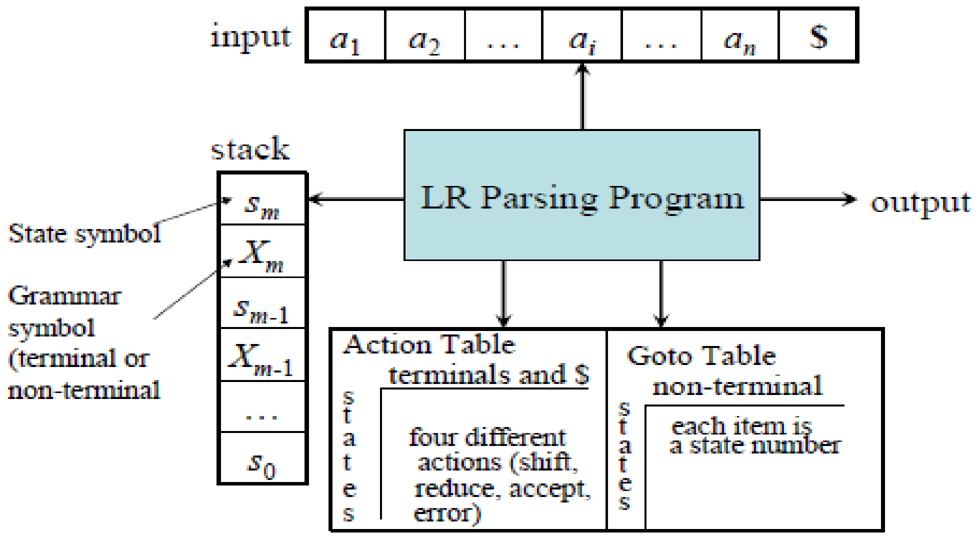
LR-Parsers cover wide range of grammars.

- SLR – simple LR parser
- LR – most general LR parser
- LALR – intermediate LR parser (look-head LR parser)

SLR, LR and LALR work same (they used the same algorithm), only their parsing tables are different.



LR Parsers: General Structure



Constructing SLR Parsing Tables

For constructing a SLR parsing table of given grammar we need,
To construct the canonical LR(0) collection of the grammar, which uses the 'closure' operation and 'goto' operation.

LR(0) Item:

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

A $\rightarrow$aBb yields the following 4 possible LR(0) items which are:
A $\rightarrow$**.**aBb
A $\rightarrow$a**.**Bb
A $\rightarrow$aB.b
A $\rightarrow$aBb**.**

Note: - The production A $\rightarrow\epsilon$ generates only one LR(0) item, A $\rightarrow$**.**

Canonical LR(0) collection:

An LR (0) item is a production G with dot at some position on the right side of the production. LR(0) items is useful to indicate that how much of the input has been scanned up to a given point in the process of parsing. In the LR (0), we place the reduce node in the entire row. A collection of sets of LR(0) items is called canonical LR(0) collection. To construct canonical LR(0) collection for a grammar we require augmented grammar and closure & goto functions.

Augmented grammar:

If G is a grammar with start symbol S, then the augmented grammar G' of G is a grammar with a new start symbol S' and production S' →S

Eg: the grammar,

$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow (E) \mid \textbf{id}$

Its augmented grammar is;

E' →E
$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow (E) \mid \textbf{id}$

The Closure Operation:

Say I is a set of items and one of these items is A→α·Bβ. This item represents the parser having seen α and records that the parser might soon see the remainder of the RHS. For that to happen the parser must first see a string derivable from B. Now consider any production starting with B, say B→γ. If the parser is to making progress on A→α·Bβ, it will need to be making progress on one such B→·γ. Hence we want to add all the latter productions to any state that contains the former. We formalize this into the notion of closure.

Definition:

If ***I*** is a set of LR(0) items for a grammar G, then ***closure(I)*** is the set of LR(0) items constructed from ***I*** by the two rules:

1. Initially, every LR(0) item in ***I*** is added to closure(I).
2. If A → α**.**Bβ is in closure(I) and B→γ is a production rule of G then add B→.γ in the closure(I) repeat until no more new LR(0) items added to closure(I).

The Closure Operation: Example

Consider a grammar:

$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow (E) \mid \textbf{id}$

Its augmented grammar is;

E' →E

$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow (E) \mid \mathbf{id}$

Now closure (E' →E) contains the following items:

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

The goto Operation:

If I is a set of LR(0) items and X is a grammar symbol (terminal or non-terminal), then goto(I,X) is defined as follows:

If A → α.Xβ in I then every item in **closure({A →** α**X**.β**})** will be in goto(I,X).

Example:

I = { E' → .E, E → .E+T, E → .T, T → .T*F, T → .F, F → .(E), F → .id }
goto(I,E) = *closure*({[*E'* → *E* •, *E* → *E* • + *T*]}) = { E' → E., E → E.+T }
goto(I,T) = { E → T., T → T.*F }
goto(I,F) = {T → F. }
goto(I,( ) = *closure*({[*F* →(•E)]})
= { F → (.E), E → .E+T, E → .T, T → .T*F, T → .F, F → .(E), F → .id }
goto(I,id) = { F → id. }

Construction of canonical LR(0) collection

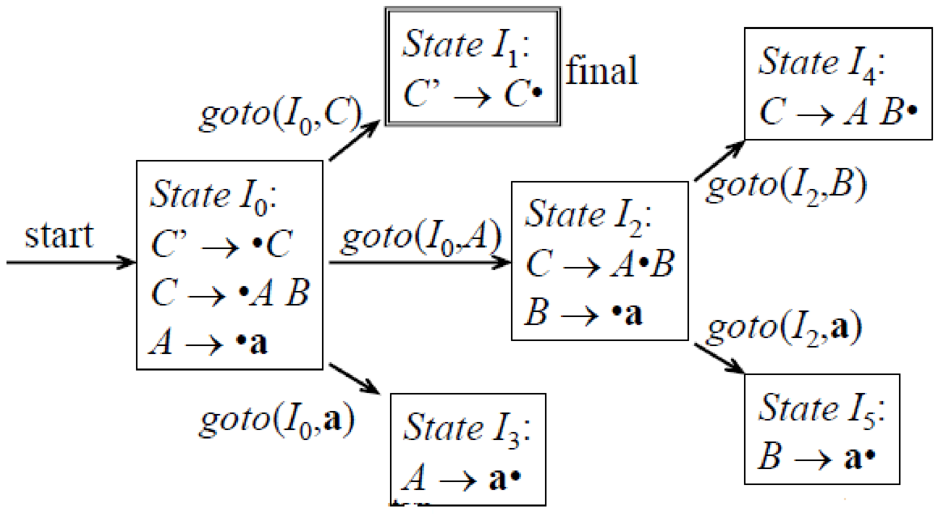
Algorithm:

```
Augment the grammar by adding production S' → S
    C = { closure({S'→.S}) }
repeat the followings until no more set of LR(0) items can be added to C.
    for each I in C and each grammar symbol X
        if goto(I,X) is not empty and not in C
            add goto(I,X) to C
```

Example: The augmented grammar is:

C'→C
C→AB
A→a
B→a

I_0 = *closure* (*C'* → •*C*)
I_1 = goto(I_0,*C*) = *closure*(*C'* → *C*•)
and so on


start
State I_0: $C' \to \bullet C$, $C \to \bullet A\ B$, $A \to \bullet \mathbf{a}$
goto(I_0,C)
State I_1: $C' \to C\bullet$
final
goto(I_0,A)
State I_2: $C \to A \bullet B$, $B \to \bullet \mathbf{a}$
goto(I_0,a)
State I_3: $A \to \mathbf{a}\bullet$
goto(I_2,B)
State I_4: $C \to A\ B\bullet$
goto(I_2,a)
State I_5: $B \to \mathbf{a}\bullet$

Example 2: Find canonical LR (0) collection for the following grammar:

E' →E
E → *E* + *T* | *T*
T → *T* * *F* | *F*
F → **(** *E* **)** | **id**

Solution ---------do itself------------

<u>Constructing SLR Parsing Tables</u>

Algorithm

1. Construct the canonical collection of sets of LR(0) items for G'.
   C← {I_0... In}
2. Create the parsing action table as follows
   - If A→α.aβ is in I_i and goto(I_i,a) = Ij then set action[i , a] = ***shift j.***
   - If A→α. is in Ii , then set action[i,a] to" ***reduce A***→α" for all 'a' in FOLLOW(A) where A≠S'.
   - If S'→S. is in Ii , then action[i,$] = ***accept***.
   - If any conflicting actions generated by these rules, the grammar is not SLR(1).
3. Create the parsing goto table
   - for all non-terminals A, if goto(Ii,A)=Ij then goto[i,A]=j
4. All entries not defined by (2) and (3) are errors.
5. Initial state of the parser contains S'→**.**S

Example 1: Construct the SLR parsing table for the grammar:

S → AA
A → aA | b

Add Augment Production and insert '•' symbol at the first position for every production in G

S` → •S
S → •AA
A → •aA
A → •b

I0 State:

Add Augment production to the I0 State and Compute the Closure

I0 = Closure (S` → •S)

Add all productions starting with S in to I0 State because "•" is followed by the non-terminal. So, the I0 State becomes

I0 = S` → •S
S → •AA

Add all productions starting with "A" in modified I0 State because "•" is followed by the non-terminal. So, the I0 State becomes.

I0= S` → •S
S → •AA
A → •aA
A → •b

State I1= Go to (I0, S) = closure (S` → S•) = S` → S•
Here, the Production is reduced so close the State.

I1= S` → S•

State I2= Go to (I0, A) = closure (S → A•A)
Add all productions starting with A in to I2 State because "•" is followed by the non-terminal. So, the I2 State becomes

I2 =S→A•A
A → •aA
A → •b

Go to (I2, a) = Closure (A → a•A) = (same as I3)

Go to (I2, b) = Closure (A → b•) = (same as I4)

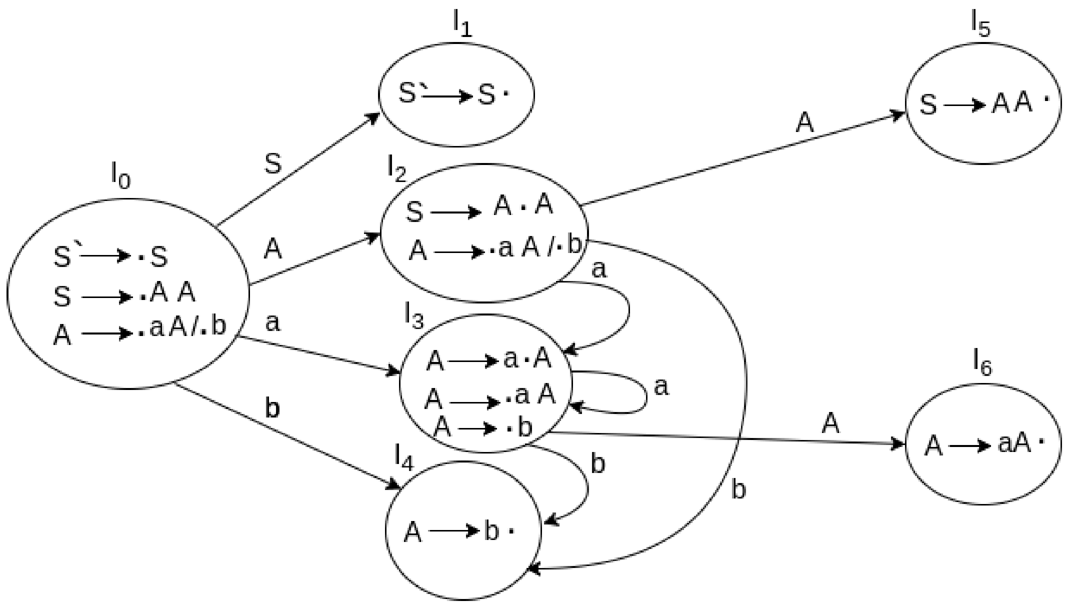
I3= Go to (I0, a) = Closure (A → a•A)

Add productions starting with A in I3.
A → a•A
A → •aA
A → •b

Go to (I3, a) = Closure (A → a•A) = (same as I3)
Go to (I3, b) = Closure (A → b•) = (same as I4)

I4= Go to (I0, b) = closure (A → b•) = A → b•
I5= Go to (I2, A) = Closure (S → AA•) = SA → A•
I6= Go to (I3, A) = Closure (A → aA•) = A → aA•

Drawing DFA:
The DFA contains the 7 states I0 to I6.



LR(0) Table

If a state is going to some other state on a terminal then it correspond to a shift move.

If a state is going to some other state on a variable then it correspond to go to move.

If a state contain the final item in the particular row then write the reduce node completely.

States	Action			Go to	
	a	b	$	A	S
I_0	S3	S4		2	1
I_1			accept		
I_2	S3	S4		5	
I_3	S3	S4		6	
I_4	r3	r3	r3		
I_5	r1	r1	r1		
I_6	r2	r2	r2		

Example 2: Construct the SLR parsing table for the grammar:

C→AB
A→a
B→a

Soln: The augmented grammar of given grammar is:

1). C'→C
2). C→AB
3). A→a
4). B→a

Step 1:- construct the canonical LR(0) collection for the grammar as,

State I_0:	State I_1 :	State I_2 :
closure(C'→.C**)**	closure (goto(I_0, C))	closure (goto(I_0, A))
C'→**.**C	closure(C'→C**.)**	closure(C→A**.**B)
C→**.**AB	C'→C**.**	C→A**.**B
A→**.**a		B→**.**a

State I_3:	State I_4 :	State I_5 :
closure (goto($I_{0,}$ a))	closure (goto($I_{2,}$ B))	closure (goto($I_{2,}$ a))
closure(A→a.**)**	closure(C→AB.**)**	closure(B→**.**a)
A→a**.**	C→AB**.**	B→a**.**

Step 2 : Construct SLR parsing table that contains both action and goto table as follows:

State I_0:
$C' \to \bullet C$
$C \to \bullet A\ B$
$A \to \bullet\mathbf{a}$

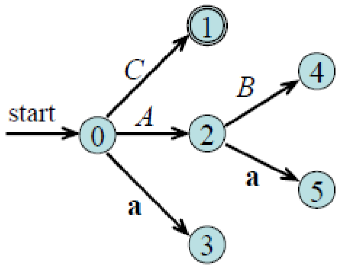
State I_1:
$C' \to C\bullet$

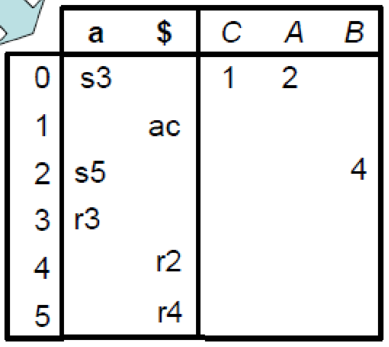
State I_2:
$C \to A\bullet B$
$B \to \bullet\mathbf{a}$

State I_3:
$A \to \mathbf{a}\bullet$

State I_4:
$C \to A\ B\bullet$

State I_5:
$B \to \mathbf{a}\bullet$





	a	$	C	A	B
0	s3		1	2	
1		ac			
2	s5				4
3	r3				
4		r2			
5		r4			

Example 2: Construct the SLR parsing table for the grammar:

1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → id

Solution:

State	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
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			

Homework:

[1]. Construct the SLR parsing table for the following grammar

X → S S + | S S * | a

[2]. Construct the SLR parsing table for the following grammar

S’ → S
S → aABe
A → Abc
A → b
B → d

LR(1) Grammars

SLR is so simple and can only represent the small group of grammar
LR(1) parsing uses look-ahead to avoid unnecessary conflicts in parsing table
LR(1) item = LR(0) item + look-ahead

LR(0) item:	LR(1) item:
$[A\rightarrow\alpha\bullet\beta]$	$[A\rightarrow\alpha\bullet\beta, a]$

Constructing LR(1) Parsing Tables

Computation of Closure for LR(1)Items:

1. Start with *closure*(*I*) = *I* (where I is a set of LR(1) items)
2. If $[A\rightarrow\alpha\bullet B\beta, a] \in$ *closure*(*I*) then
   add the item $[B\rightarrow\bullet\gamma, b]$ to *I* if not already in *I*, *where* $b \in$ FIRST(βa).
3. Repeat 2 until no new items can be added.

Computation of Goto Operation for LR(1) Items:

If I is a set of LR(1) items and X is a grammar symbol (terminal or non-terminal), then goto(I,X) is computed as follows:

1. For each item $[A\rightarrow\alpha\bullet X\beta, a] \in I$, add the set of items
   closure($\{[A\rightarrow\alpha X\bullet\beta, a]\}$) to *goto*($I$,$X$) if not already there
2. Repeat step 1 until no more items can be added to *goto*(I,X)

Construction of The Canonical LR(1) Collection:

Algorithm:

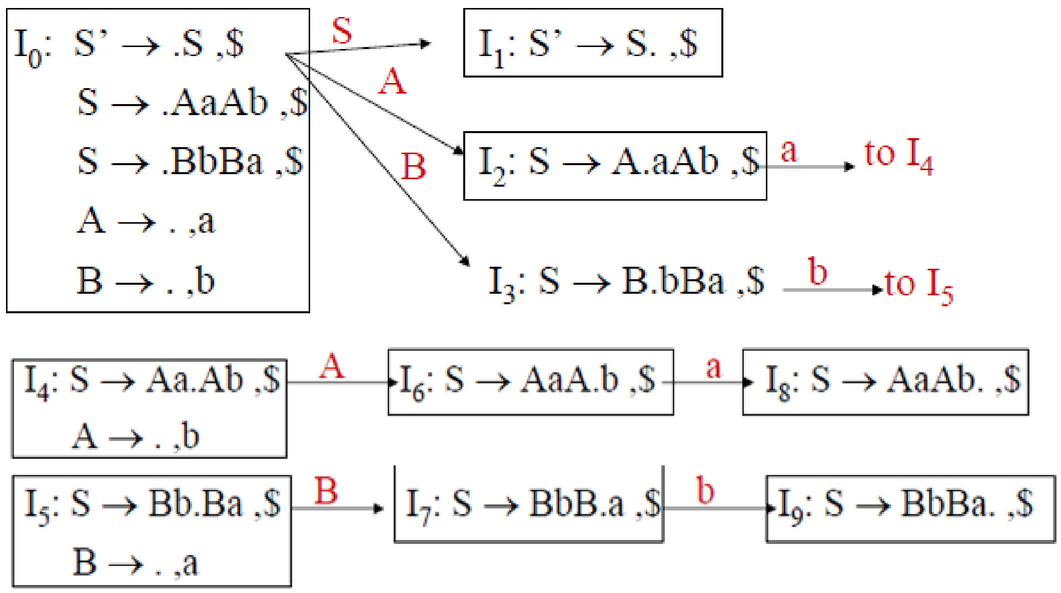
```
Augment the grammar with production S'→S
C = { closure({S'→.S,$}) } (the start stat of DFA)
repeat the followings until no more set of LR(1) items can be added to C.
    for each I ∈ C and each grammar symbol X ∈ (N∪T)
        goto(I,X) ≠ φ and goto(I,X) not ∈ C then
            add goto(I,X) to C
```

Example: Construct canonical LR(1) collection of the grammar:

S→AaAb
S→BbBa
A→∈
B→∈

Its augmented grammar is:

S'→S
S→AaAb
S→BbBa
A→∈
B→∈



Constructing LR(1) Parsing Tables

SLR used the LR(0) items, that is the items used were productions with an embedded dot, but contained no other (lookahead) information. The LR(1) items contain the same productions with embedded dots, but add a second component, which is a terminal (or $). This second component becomes important only when the dot is at the extreme right. For LR(1) we do that reduction only if the input symbol is exactly the second component of the item.

Algorithm:

1. Construct the canonical collection of sets of LR(1) items for G'.
   C = {I_0... In}
2. Create the parsing action table as follows
   - If [A→α.aβ,b] in I_i and goto(Ii,a) = Ij then action[i,a] = ***shift j.***
   - If A→α., a is in Ii, then action[i,a] = ***reduce A***→α where A≠S'.
   - If S'→S.,$ is in Ii , then action[i,$] = ***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 goto(Ii,A) = Ij then goto[i,A] = j
4. All entries not defined by (2) and (3) are errors.
5. Initial state of the parser contains S'→.S, $

LR(1) Parsing Tables: Example1

Construct LR(1) parsing table for given grammar:

S'→S
S→CC
C→cC
C→d

I_0: S'→.S, $
S→**.**CC, $
C→**.**cC, c / d
C→**.**d, c / d

I_1: S'→S**.** , $

I_2: S→C**.C** , $
C→**.**cC, $
C→**.**d, $

…………………………………….up to I9

Example 2:
Construct LR(1) parsing table for the augmented grammar,

1. S' → *S*
2. *S* → *L* = *R*
3. *S* → *R*
4. *L* → ***** *R*
5. *L* → **id**
6. *R* → *L*

Step 1: At first find the canonical collection of LR(1) items of the given augmented grammar as,

State I_0:
closure(S'→**.**S, $**)**
S'→**.**S, $
S → **.***L* = *R*, $
S → **.***R*, $

State I_1 :
closure (goto(I_0, S))
closure(S'→S**., $)**
S'→S**. , $**

State I_2 :
closure (goto(I_0, L))
closure((*S* → *L*. = *R*, $),(*R* →L**.** ,$))
S → *L*. = *R*, $
R →L**.** ,$

L → **.*** *R, $*
L →**. Id, =**
R →**.***L,$*

State I_3 :
closure (goto(I_0, R))
closure(*S* → R**.** , *$*)
S → R**.** , *$*

State I_4 :
closure(goto(I_0, *))
closure(*L* → *** .***R, =)*
{(*L* → *** .***R,* =),(*R* →**.***L*,=),(*L* → **.*** *R, =)*,(*L* →**. Id, =**)}

State I_5 :
closure (goto(I_0, id))
closure(*L* →**Id. , =)**
L →**Id. , =**

State I_6 :
closure(goto((I_2, =))
closure(*S* → *L* **=.** *R, $*)
S → *L* **=.** *R, $*
R →**.***L, $*
L → **.*** *R, $*
L →**. Id, $**

State I_7 :
closure(goto((I_4, R))
closure(*L* →***** *R***.** *,*=)
L →***** *R***.** , =

State I_8 :
closure (goto(I_4, L))
closure(R→L. , =)
R→L. , =

State I_9 :
closure(goto((I_6, R))
closure(S→L=R. , $)
S→L=R. , $

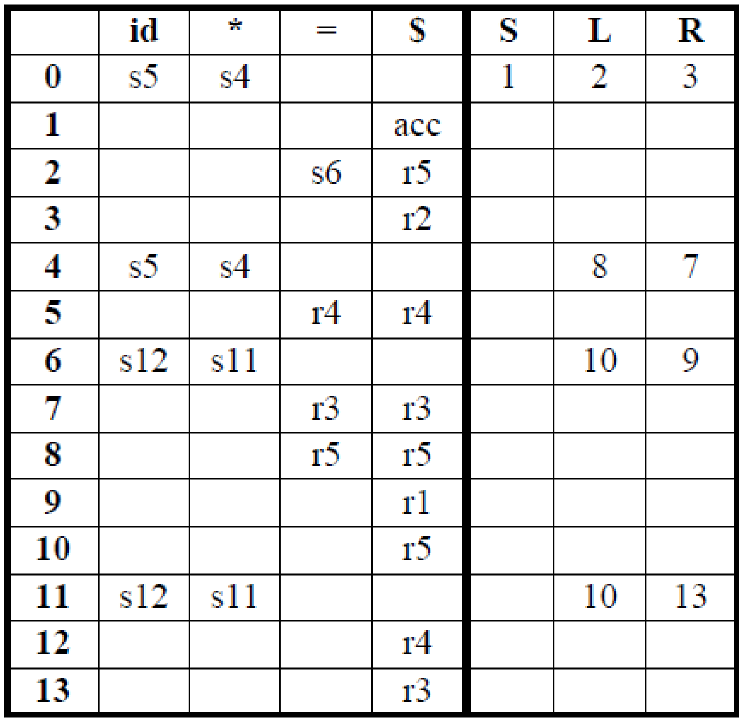
State I_{10} :
closure(goto((I_6, L))
R→L. , $

State I_{11} :
Closure(goto(I_6, *))
Closure(L→*.R, $)
L→***.**R , $
R→**.**L , $
L→**.***R , $
L→**.id** , $

State I_{12} :
closure(goto((I_6, id))
closure(L→id. , $)
L→id**.** , $

State I_{13} :
closure(goto((I_{11}, R))
L→*R. , $

Step 2: Now construct LR(1) parsing table



	id	*	=	$	S	L	R
0	s5	s4			1	2	3
1				acc			
2			s6	r5			
3				r2			
4	s5	s4				8	7
5			r4	r4			
6	s12	s11				10	9
7			r3	r3			
8			r5	r5			
9				r1			
10				r5			
11	s12	s11				10	13
12				r4			
13				r3			

LALR(1) Grammars

It is an intermediate grammar between the SLR and LR(1) grammar.
A typical programming language generates thousands of states for canonical LR parsers while they generate only hundreds of states for LALR parser.
LALR(1) parser combines two or more LR(1) sets(whose core parts are same) into a single state to reduce the table size.

Example:

I_1: L → id. , = $\qquad\qquad\qquad\qquad$ I_{12}: $\quad$ L → id. , =

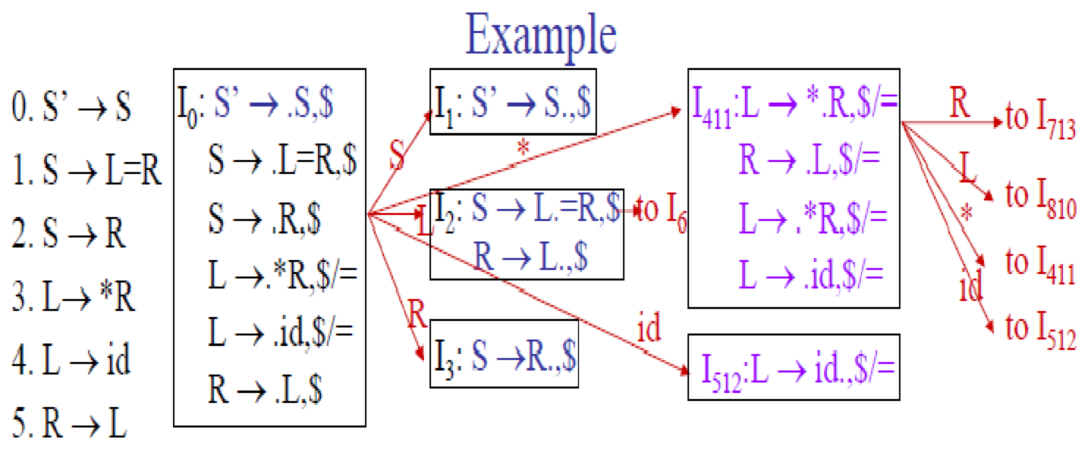
$\qquad\qquad\qquad\Longrightarrow$

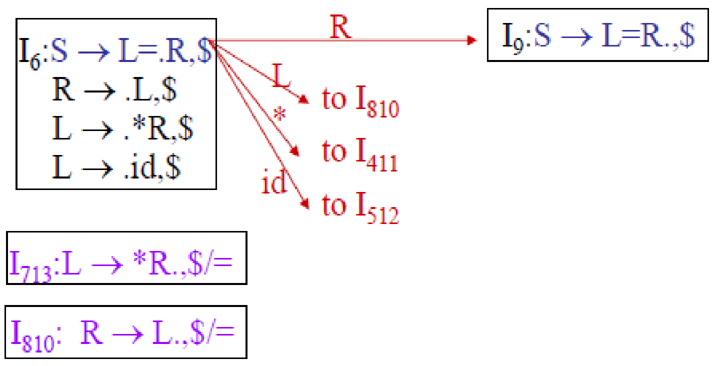
I_2: L → id. , $ $\qquad\qquad\qquad\qquad\qquad\quad$ L → id. , $

Constructing LALR Parsing Tables

1. Create the canonical LR(1) collection of the sets of LR(1) items for the given grammar C={I_0,...,In} .
2. Find each core; find all sets having that same core; replace those sets having same cores with a single set which is their union.
   C={I0,...,In} then C'={J1,...,Jm}where m ≤ n
3. Create the parsing tables (action and goto tables) same as the construction of the parsing tables of LR(1) parser.
   – Note that: If J=I1 ∪ ... ∪ Ik since I1,...,Ik have same cores then cores of goto(I1,X),...,goto(I2,X) must be same.
   – So, goto(J,X)=K where K is the union of all sets of items having same cores as goto(I1,X).
4. If no conflict is introduced, the grammar is LALR(1) grammar.

(We may only introduce reduce/reduce conflicts; we cannot introduce a hift/reduce conflict)





Same Cores
I4 and I11

I5 and I12

I7 and I13

I8 and I10

Grammar:
1. $S' \to S$
2. $S \to L = R$
3. $S \to R$
4. $L \to * R$
5. $L \to \textbf{id}$
6. $R \to L$

	id	*****	**=**	**$**	**S**	**L**	**R**
0	s5	s4			1	2	3
1				acc			
2			s6	r5			
3				r2			
4	s5	s4				8	7
5			r4	r4			
6	s12	s11				10	9
7			r3	r3			
8			r5	r5			
9				r1			

no shift/reduce or
no reduce/reduce
conflict
⇓
so, it is a LALR
grammar

Kernel item:

This includes the initial items, S'→S and all items whose dot are not at the left end.

Non-Kernel item:
The productions of a grammar which have their dots at the left end are non-kernel items.

Example: Do itself……………….
Take any one grammar then find canonical collection of LR(0) items and finally list kernel and non-kernel items.

Limitations of Syntax Analyzers
Syntax analyzers receive their inputs, in the form of tokens, from lexical analyzers. Lexical analyzers are responsible for the validity of a token supplied by the syntax analyzer. Syntax analyzers have the following drawbacks -

- it cannot determine if a token is valid,
- it cannot determine if a token is declared before it is being used,
- it cannot determine if a token is initialized before it is being used,
- it cannot determine if an operation performed on a token type is valid or not.

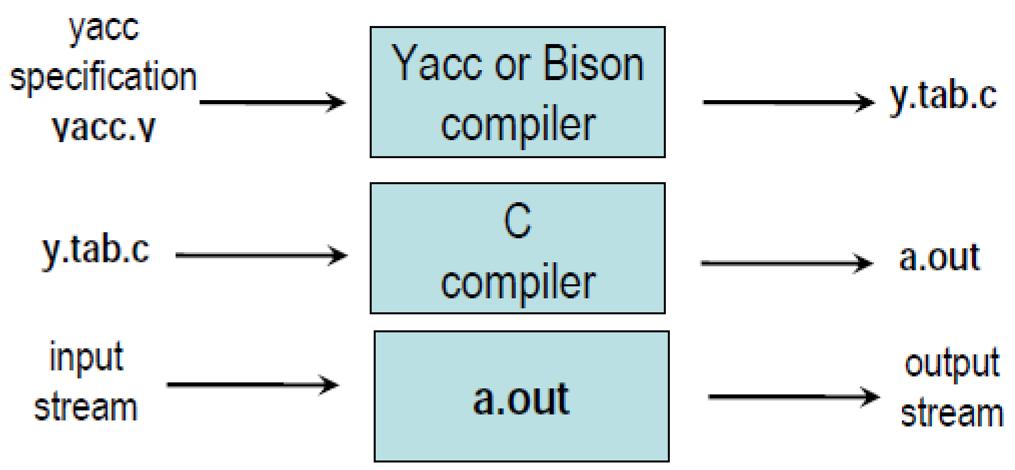
These tasks are accomplished by the semantic analyzer, which we shall study in Semantic Analysis.

Parser Generators

Introduction to Bison

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

- The job of the Bison parser is to group tokens into groupings according to the grammar rules—for example, to build identifiers and operators into expressions.
- The tokens come from a function called the lexical analyzer that must supply in some fashion (such as by writing it in C).
- The Bison parser calls the lexical analyzer each time it wants a new token. It doesn't know what is "inside" the tokens.
- Typically the lexical analyzer makes the tokens by parsing characters of text, but Bison does not depend on this.
- The Bison parser file is C code which defines a function named *yyparse* which implements that grammar. This function does not make a complete C program: you must supply some additional functions.



Stages in Writing Bison program

1. Formally specify the grammar in a form recognized by Bison
2. Write a lexical analyzer to process input and pass tokens to the parser.
3. Write a controlling function that calls the Bison produced parser.
4. Write error-reporting routines.

Bison Specification

• A *bison specification* consists of four parts:

```
%{
        C declarations
%}
Bison declarations
%%
        Grammar rules
%%
Additional C codes
```

Productions in Bison are of the form

```
Non-terminal:  tokens/non-terminals {action}
               | Tokens/non | terminals {action}
               ……………………………………..
               ;
```

Bison Declaration

Tokens that are single characters can be used directly within productions, e.g. '+', '-', '*'
Named tokens must be declared first in the declaration part using

%token *Token Name (Upper Case Letter)*

e.g   %token INTEGER IDENTIFIER
       %token NUM 100

– *%left*, *%right* or *%nonassoc* can be used instead for *%token* to specify the precedence & associativity (precedence declaration). All the tokens declared in a single precedence declaration have equal precedence and nest together according to their associativity.
– *%union* declares the collection data types
– *%type <non-terminal>* declares the type of semantic values of non-terminal
– *%start <non-terminal>* specifies the grammar start symbol (by default the start symbol of grammar)

Grammar Rules

- ✓ In order for Bison to parse a grammar, it must be described by a *Context-Free Grammar* that is LALR (1).
- ✓ A non-terminal in the formal grammar is represented in Bison input as an identifier, like an identifier in C. By convention, it is in *lower case*, such as *expr*, *declaration.*
- ✓ A Bison grammar rule has the following general form:
  - *RESULT: COMPONENTS...;*

  where, RESULT is the non-terminal symbol that this rule describes and COMPONENTS are various terminal and non-terminal symbols that are put together by this rule.
  For example, *exp: exp '+' exp;* says that two groupings of type 'exp', with a '+' token in between, can be combined into a larger grouping of type 'exp'.
- ✓ Multiple rules for the same RESULT can be written separately or can be joined with the vertical-bar character '|' as follows:

```
RESULT:     RULE1-COMPONENTS...
            | RULE2-COMPONENTS...
            ..........................................;
```

- ✓ If COMPONENTS in a rule is empty, it means that RESULT can match the empty string. For example, here is how to define a comma-separated sequence of zero or more 'exp' groupings:

```
expseq:     /* empty */
            | expseq1
            ;
expseq1:    exp
            | expseq1 ',' exp
            ;
```

It is customary to write a comment '/* empty */' in each rule with no components.

Semantic Actions:

To make program useful, it must do more than simply parsing the input, i.e., must produce some output based on the input.

Most of the time the action is to compute semantics value of whole constructs from the semantic values associated with various tokens and groupings.

For Example, here is a rule that says an expression can be the sum of two sub-expression,

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

The action says how to produce the semantic value of the sum expression from the value of two sub expressions.
In bison, the default data type for all semantics is int. i.e. the parser stack is implemented as an integer array. It can be overridden by redefining the macro YYSTYPE. A line of the form #defines YYSTYPE double in the C declarations section of the bison grammar file.

To use multiple types in the parser stack, a "union" has to be declared that enumerates all possible types used in the grammar.
Example:

```
%union{
        double val;
        char *str;
    }
```
This says that there are two alternative types: double and char*.
Tokens are given specific types by a declaration of the form:

%token <*val*> exp

Interfacing with Flex

Bison provides a function called yyparse() and *expects* a function called yylex() that performs lexical analysis. Usually this is written in lex. If not, then yylex() whould be written in the C code area of bison itself.

If yylex() is written in flex, then bison should first be called with the -d option: bison -d grammar.y
This creates a file grammar.tab.h that containes #defines of all the %token declarations in the grammar. The sequence of invocation is hence:

```
bison -d grammar.y
flex grammar.flex
gcc -o grammar grammar.tab.c lex.yy.c –lfl
```

Practice

• Get familiar with Bison: Write a desk calculator which performs '+' and '*' on unsigned integers

1. Create a Directory: "mkdir calc"
2. Save the five files (calc.lex, calc.y, Makefile, main.cc, and heading.h) to directory "calc"
3. Command Sequence: "make"; "./calc"
4. Use input programs (or stdin) which contain expressions with integer constants and operators + and *, then press Ctrl-D to see the result

Programming Example

```
/* Mini Calculator */
/* calc.lex */
%{
     #include "heading.h"
     #include "tok.h"
     int yyerror(char *s);
     int yylineno = 1;
%}
digit       [0-9]
int_const   {digit}+
%%
{int_const}{ yylval.int_val = atoi(yytext); return INTEGER_LITERAL; }
"+"         { yylval.op_val = new std::string(yytext); return PLUS; }
"*"         { yylval.op_val = new std::string(yytext); return MULT; }
[\t]*       {}
[\n]        { yylineno++;    }

.           { std::cerr << "SCANNER "; yyerror(""); exit(1); }
```

```
%%
------------------------------------------------------------------------------------------------------------
/* Mini Calculator */
/* calc.y */
%{
     #include "heading.h"
     int yyerror(char *s);
     int yylex(void);
%}
     %union{
                int       int_val;
                string*   op_val;
          }
%start     input
%token     <int_val>  INTEGER_LITERAL
%type      <int_val>  exp
%left      PLUS
%left      MULT
%%
input:          /* empty */
          | exp { cout << "Result: " << $1 << endl; }
          ;
exp:      INTEGER_LITERAL  { $$ = $1; }
          | exp PLUS exp   { $$ = $1 + $3; }
          | exp MULT exp   { $$ = $1 * $3; }
          ;
%%
int yyerror(string s)
{
     extern int yylineno;  // defined and maintained in lex.c
     extern char *yytext;  // defined and maintained in lex.c
     cerr << "ERROR: " << s << " at symbol \"" << yytext;
     cerr << "\" on line " << yylineno << endl;
     exit(1);
}
int yyerror(char *s)
{
     return yyerror(string(s));
}
```

unit: 3

Syntax Directed Translation

Introduction

We have learnt how a parser constructs parse trees in the syntax analysis phase. The plain parse-tree constructed in that phase is generally of no use for a compiler, as it does not carry any information of how to evaluate the tree. The productions of context-free grammar, which makes the rules of the language, do not accommodate how to interpret them. For example

$E \rightarrow E + T$

The above CFG production has no semantic rule associated with it, and it cannot help in making any sense of the production.

In syntax directed translation, along with the grammar we associate some informal notations and these notations are called as semantic rules. So we can say that

Grammar + semantic rule = SDT (syntax directed translation)

- In syntax directed translation, every non-terminal can get one or more than one attribute or sometimes 0 attribute depending on the type of the attribute. The value of these attributes is evaluated by the semantic rules associated with the production rule.
- In the semantic rule, attribute is VAL and an attribute may hold anything like a string, a number, a memory location and a complex record
- In Syntax directed translation, whenever a construct encounters in the programming language then it is translated according to the semantic rules define in that particular programming language.

The following tasks should be performed in semantic analysis:

- Scope resolution
- Type checking
- Array-bound checking

There are two notations for associating semantic rules with productions, which are:

- Syntax- directed definitions and
- Translation schema.

Syntax-directed definition

Syntax-Directed Definitions are high level specifications for translations. They hide many implementation details and free the user from having to explicitly specify the order in which translation takes place. A syntax-directed definition is a generalization of a context-free grammar in which each grammar symbol is associated with a set of attributes. This set of attributes for a grammar symbol is partitioned into two subsets **synthesized** and **inherited** attributes of that grammar.

In brief,

A *syntax-directed definition* is a grammar together with *semantic rules* associated with the productions. These rules are used to compute attribute values.

Mathematically,

Given a production

A → α
then each semantic rule is of the form

b = *f*(*c*1,*c*2,…,*ck*)
where *f* is a function and *ci* are attributes of *A* and α, and either

– *b* is a *synthesized* attribute of *A*
– *b* is an *inherited* attribute of one of the grammar symbols in α.

Example: ***The syntax directed definition for a simple desk calculator***

Production	**Semantic Rules**
L → E **return**	print(E.val)
E → E1 + T	E.val = E1.val + T.val
E → T	E.val = T.val
T → T1 * F	T.val = T1.val * F.val
T → F	T.val = F.val
F → (E)	F.val = E.val
F → **digit**	F.val = **digit**.lexval

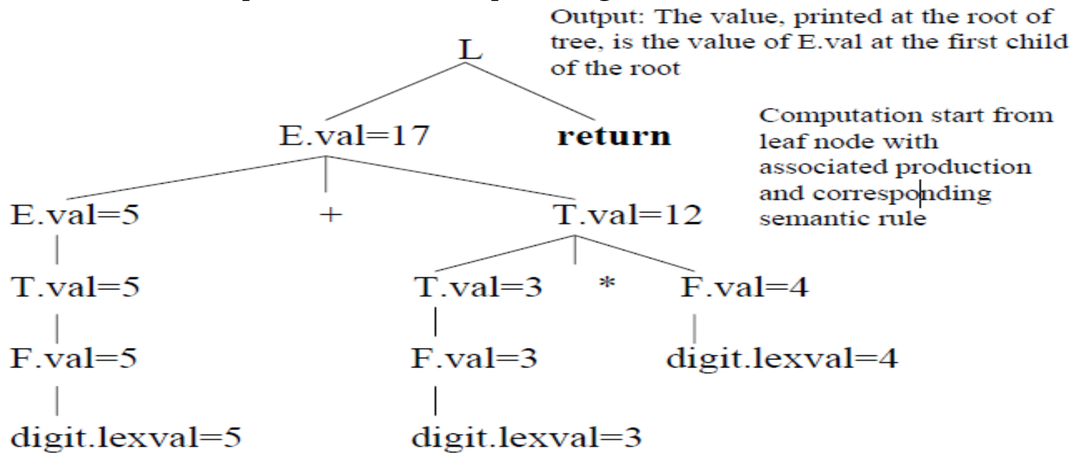
Note: all attributes in this example are of the synthesized type.

Annotated Parse Tree

A parse tree constructing for a given input string in which each node showing the values of attributes is called an annotated parse tree.
Example:

Let's take a grammar,
L → E **return**
E → E1 + T
E → T
T → T1 * F
T → F
F → (E)
F → **digit**

Now the annotated parse tree for the input string 5+3*4 is,


Output: The value, printed at the root of tree, is the value of E.val at the first child of the root
L
E.val=17
return
Computation start from leaf node with associated production and corresponding semantic rule
E.val=5
+
T.val=12
T.val=5
T.val=3
*
F.val=4
F.val=5
F.val=3
digit.lexval=4
digit.lexval=5
digit.lexval=3

Inherited and Synthesized Attributes:

A node in which attributes are derived from the parent or siblings of the node is called inherited attribute of that node. As in the following production,

S → ABC

'A' can get values from S, B and C. B can take values from S, A, and C. Likewise, C can take values from S, A, and B.

The attributes of a node that are derived from its children nodes are called synthesized attributes. Terminals do not have inherited attributes. A non-terminal 'A' can have both inherited and synthesized attributes. The difference is how they are computed by rules associated with a production at a node N of the parse tree. To illustrate, assume the following production:

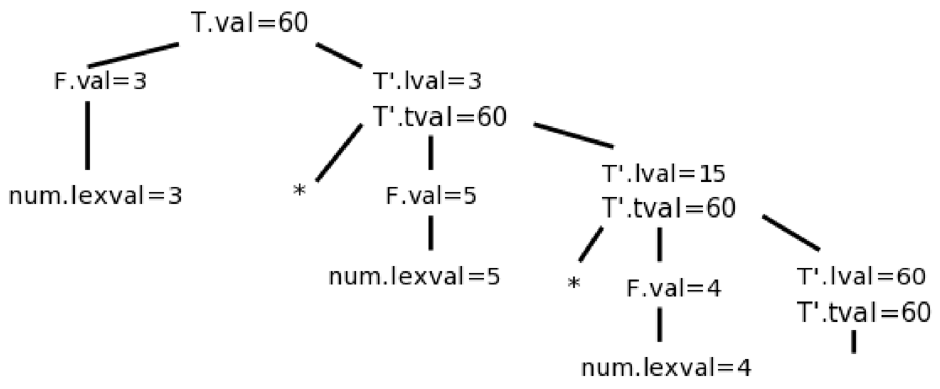
S → ABC

If S is taking values from its child nodes (A, B, C), then it is said to be a synthesized attribute, as the values of ABC are synthesized to S.

Example:

T → F T'
T → T * F
T → F
F → num

Production	Semantic Rules	Type
T → F T'	T'.lval = F.val	Inherited
	T.val = T'.tval	Synthesized
T' → * F T_1'	T'$_1$.lval = T'.lval * F.val	Inherited
	T'.tval = T'$_1$.tval	Synthesized
T' → ε	T'.tval = T'.lval	Synthesized
F → num	F.val = num.lexval	Synthesized



Dependency Graph

If interdependencies among the inherited and synthesized attributes in an annotated parse tree are specified by arrows then such a tree is called dependency graph.

In order to correctly evaluate attributes of syntax tree nodes, a *dependency graph* is useful. A dependency graph is a directed graph that contains attributes as nodes and dependencies across attributes as edges.

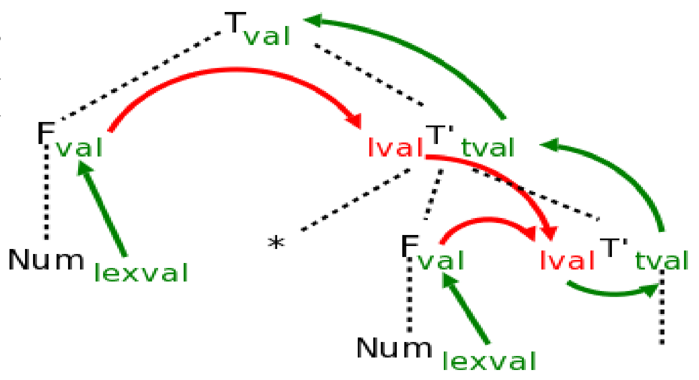
Example: let’s take a grammar,

$T \rightarrow F\ T$

$T \rightarrow T * F$

$T \rightarrow F$

$F \rightarrow \text{num}$



Example 2:

$D \rightarrow T\ L$

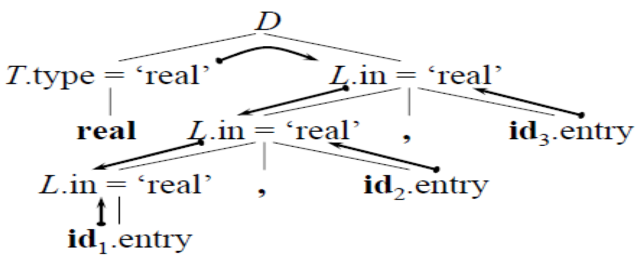
$T \rightarrow \textbf{int}$

$T \rightarrow \textbf{real}$

$L \rightarrow L_1\ \textbf{id}$

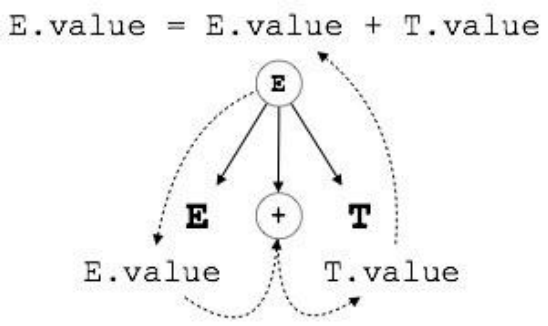
$L \rightarrow \textbf{id}$

Input : real id1 , id2 , id3



S-Attributed Definitions

If an SDT uses only synthesized attributes, it is called as S-attributed SDT. These attributes are evaluated using S-attributed SDTs that have their semantic actions written after the production (right hand side).



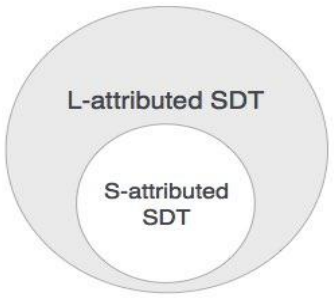
As depicted above, attributes in S-attributed SDTs are evaluated in bottom-up parsing, as the values of the parent nodes depend upon the values of the child nodes.

L-Attributed Definitions

This form of SDT uses both synthesized and inherited attributes with restriction of not taking values from right siblings. In L-attributed SDTs, a non-terminal can get values from its parent, child, and sibling nodes. As in the following production

$S \rightarrow ABC$

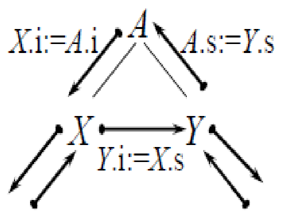
S can take values from A, B, and C (synthesized). 'A' can take values from S only. B can take values from S and A. C can get values from S, A, and B. No non-terminal can get values from the sibling to its right. Attributes in L-attributed SDTs are evaluated by depth-first and left-to-right parsing manner.



We may conclude that if a definition is S-attributed, then it is also L-attributed as L-attributed definition encloses S-attributed definitions.

Example:

$A \rightarrow X\,Y$

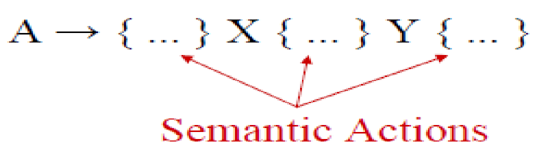


$$X.i := A.i$$
$$Y.i := X.s$$
$$A.s := Y.s$$

Translation Schema

- The Syntax directed translation scheme is a context -free grammar.
- The syntax directed translation scheme is used to evaluate the order of semantic rules.
- In translation scheme, the semantic rules are embedded within the right side of the productions.
- The position at which an action is to be executed is shown by enclosed between braces. It is written within the right side of the production.

Example:

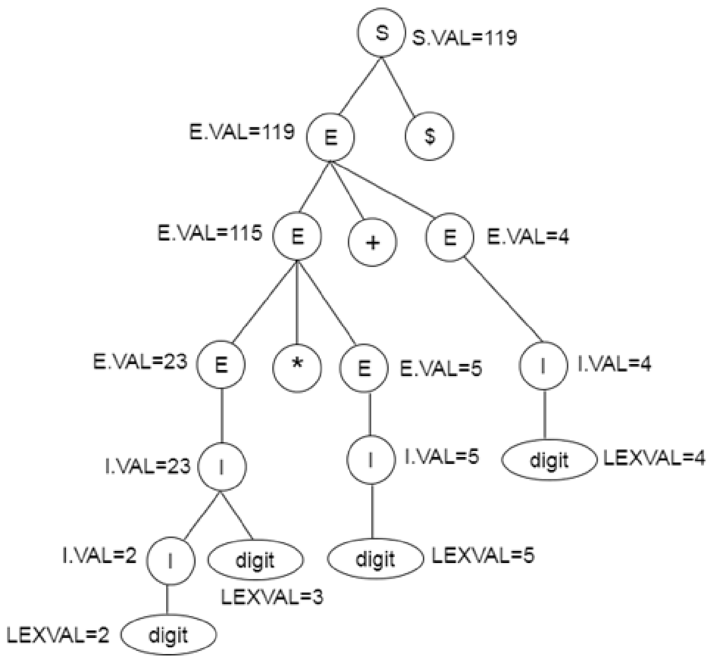


Example:

Production	Semantic Rules
S → E $	{ printE.VAL }
E → E + E	{E.VAL := E.VAL + E.VAL }
E → E * E	{E.VAL := E.VAL * E.VAL }
E → (E)	{E.VAL := E.VAL }
E → I	{E.VAL := I.VAL }
I → I digit	{I.VAL := 10 * I.VAL + LEXVAL }
I → digit	{ I.VAL:= LEXVAL}

- **E.val** is one of the attributes of E.
- **num.lexval** is the attribute returned by the lexical analyzer.

<u>Parse tree for SDT:</u>



Q:- Define the syntax directed definitions with an example. How definitions are different from translation schemas?

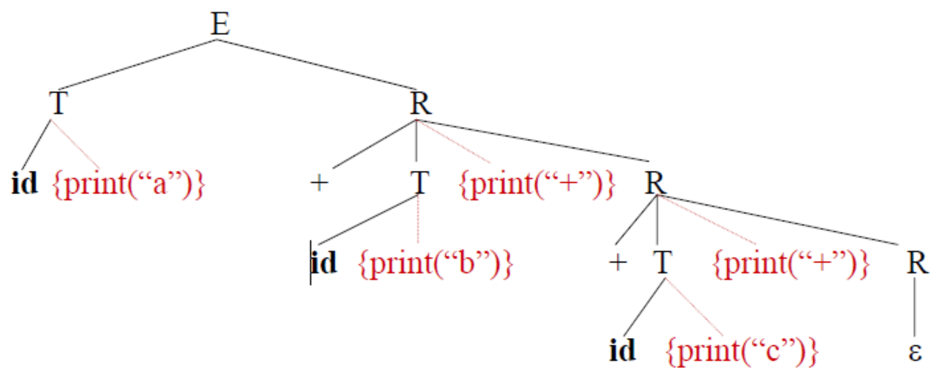
In syntax directed definition each grammar symbols associated with the semantic rules while in translation schema we use semantic actions instead of semantic rules.

Example: A simple translation schema that converts infix expression into corresponding postfix expressions.

$$
\begin{aligned}
&E \rightarrow T\ R \\
&R \rightarrow +\ T\ \{\ \text{print}(\text{“+”})\ \}\ R_1 \\
&R \rightarrow \varepsilon \\
&T \rightarrow \textbf{id}\ \{\ \text{print}(\textbf{id}.\text{name})\ \}
\end{aligned}
$$

a+b+c ➔ ab+c+

infix expression postfix expression



Eliminating Left Recursion from a Translation Scheme

Let us take a left recursive translation schema:

$A \rightarrow A1\ Y\ \{\ A.a = g(A1.a, Y.y)\ \}$

$A \rightarrow X\ \{\ A.a = f(X.x)\ \}$

In this grammar each grammar symbol has a synthesized attribute written using their corresponding lower case letters.

Now eliminating left recursion as,

$A \rightarrow XR$

$R \rightarrow YR1$

$R \rightarrow \varepsilon$

Hint: $A \rightarrow A\ \alpha \mid \beta$

$\Downarrow$

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' \mid \varepsilon$

Now taking the new semantic actions for each symbols as follows,

inherited attribute of the new non-terminal

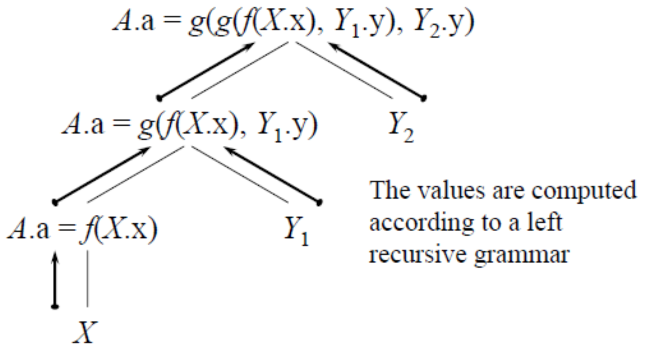
synthesized attribute of the new non-terminal

$$
\begin{aligned}
&A \rightarrow X\ \{\ R.in = f(X.x)\ \}\ R\ \{\ A.a = R.syn\ \} \\
&R \rightarrow Y\ \{\ R_1.in = g(R.in, Y.y)\ \}\ R_1\ \{\ R.syn = R_1.syn\} \\
&R \rightarrow \varepsilon\ \ \{\ R.syn = R.in\ \}
\end{aligned}
$$

When we eliminate the left recursion from the grammar (to get a suitable grammar for the top-down parsing) we also have to change semantic actions

Evaluating attributes

Evaluation of string XYY


A.a = g(g(f(X.x), Y1.y), Y2.y)
A.a = g(f(X.x), Y1.y)
Y2
A.a = f(X.x)
Y1
X
The values are computed according to a left recursive grammar

----

Q. For the following grammar:

E → E + E | E * E | (E) | id

Annotate the grammar with syntax directed definitions using synthesized attributes. Remove left recursion from the grammar and rewrite the attributes correspondingly.

Soln: First part:

E → E + E {E.val = E1.val + E2.val}
E → E * E {E.val = E1.val * E2.val}
E → (E) {E.val = E1.val}
E →id {E.val=id.lexval}

Second part:

Removing left recursion as,
E → (E)R | id R
R → +ER1 | *ER1 | ε

Now add the attributes within this non-left recursive grammar as,

E → (E) {R.in=E1.val}R {E.val=R.syn}
E →id {R.in=id.lexval} R1 { E.val=R.syn }
R → +E {R1.in=E.val+R.in}R1 {R.syn=R1.syn}
R → *E { R1.in=E.val*R.in }R1 { R.syn=R1.syn }
R → ε { R.syn=R.in}

Q. For the following grammar:

E → E + E | E - E | T
T → (E) | num

Annotate the grammar with syntax directed definitions using synthesized attributes. Remove left recursion from the grammar and rewrite the attributes correspondingly.

Soln: do itself

Q. For the following grammar:

E → E + E | E * E | (E) | id

At first remove the left recursion then construct an annotated parse tree for the expression 2*(3+5) using the modified grammar.

Trace the path in which semantic attributes are evaluated.

E → E + E {E.val = E1.val + E2.val}
E → E * E {E.val = E1.val * E2.val}
E → (E) {E.val = E1.val}
E →id {E.val=id.lexval}

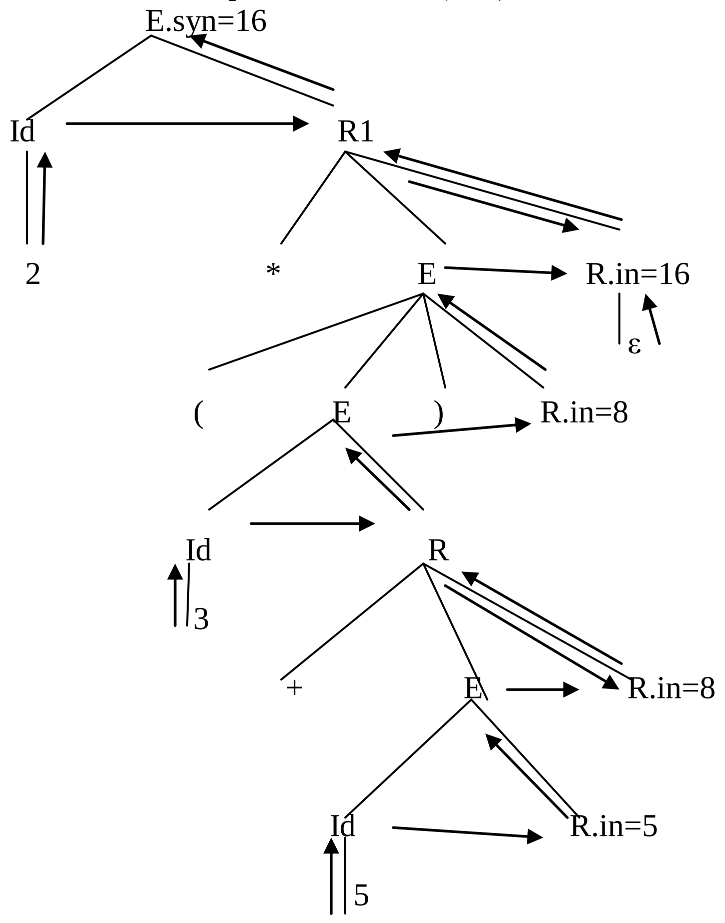
Removing left recursion as,

E → (E)R | id R
R → +ER1 | *ER1 | ε

Now add the attributes within this non-left recursive grammar as,

E → (E) {R.in=E1.val}R {E.val=R.syn}
E →id {R.in=id.lexval} R1 { E.val=R.syn }
R → +E {R1.in=E.val+R.in}R1 {R.syn=R1.syn}
R → *E { R1.in=E.val*R.in }R1 { R.syn=R1.syn }
R → ε { R.syn=R.in}

Second part: An annotated parse tree for 2*(3+5) is,



Type Checking

Compiler must check that the source program follows both the syntactic and semantic conventions of the source language. Type checking is the process of checking the data type of different variables.
The design of a type checker for a language is based on information about the syntactic construct in the language, the notation of type, and the rules for assigning types to the language constructs.

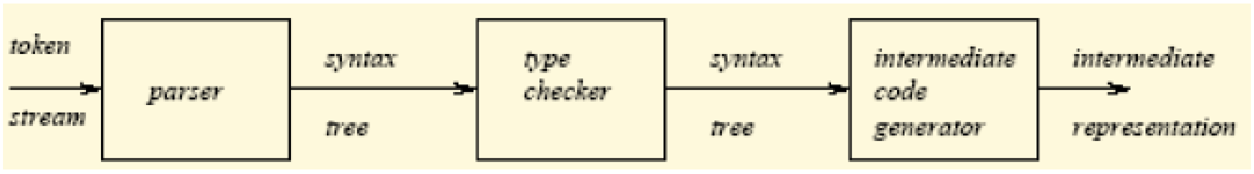


Fig: Position of Type Checker

Type expressions

The type of a language construct is denoted by a *type expression*.
A *type expression* can be:

- **A basic type**
  - a primitive data type such as *integer, real, char, boolean*, …
  - *type-error* signal an error during type checking
  - *void* : no type
- **A type name**
  - a name can be used to denote a type expression.
- **A type constructor applies to other type expressions.**
  - **arrays**: If T is a type expression, then *array(I,T)* is a type expression where I denotes index range. Ex: array(0..99, int)
  - **products**: If T1 and T2 are type expressions, then their Cartesian product *T1 X T2* is a type expression. Ex: int x int
  - **pointers**: If T is a type expression, then *pointer(T)* is a type expression. Ex: pointer(int)
  - **functions**: We may treat functions in a programming language as mapping from a domain type D to a range type R. So, the type of a function can be denoted by the type expression *D→R* where D is R type expressions. Ex: int → int represents the type of a function which takes an int value as parameter, and its return type is also **int.**

Type systems

The collection of different data types and their associated rules to assign types to programming language constructs is known as type systems.

- Informal type system rules, for example "*if both operands of addition are of type integer, then the result is of type integer*"
- A type checker implements type system

Example Type Checking of Expressions

$E \rightarrow$ id	{ E.type = l*ookup(id.entry)* }
$E \rightarrow$ charliteral	{ E.type = *char* }
$E \rightarrow$ intliteral	{ E.type = *int* }
$E \rightarrow E1 + E2$	{ E.type = ($E1$.type == $E2$.type) ? $E1$.type : *type_error* }
$E \rightarrow E1\ [E2]$	{ E.type = ($E2$.type == *int* and $E1$.type == *array(s,t)*) ? t : *type_error* }
$E \rightarrow E1 \uparrow$	{ E.type = ($E1$.type == *pointer(t)*) ? t : *type_error* }
$S \rightarrow$ id = E	{S.type = (id.type == E.type) ? *void* : *type_error*}

Note: the type of id is determined by : id.type = lookup(id.entry)

$S \rightarrow$ if E then $S1$	{S.type = (E.type == *boolean*) ? $S1$.type : *type_error*}
$S \rightarrow$ while E do $S1$	{S.type = (E.type == *boolean) ?* $S1$.type : *type_error*}
$S \rightarrow S1$; $S2$	{S.type = ($S1$.type == *void* and $S2$.type == *void) ? void : type_error*}

Static versus Dynamic type Checking

Static checking: The type checking at the compilation time is known as static checking. Typically syntactical errors and misplacement of data type take place at this stage.

– Program properties that can be checked at compile time known as static checking.

– Typical examples of static checking are:

- Type checks
- Flow-of-control checks
- Uniqueness checks
- Name-related checks

***<u>Dynamic type checking</u>*: ** The type checking at the run time is known as static checking. Compiler generates verification code to enforce programming language's dynamic semantics.

- A programming language is *strongly-typed*, if every program its compiler accepts will execute without type errors.

* In practice, some of types checking operations are done at run-time (so, most of the programming languages are not strongly-typed).
* Example: **int** x[100]; … x[i] →most of the compilers cannot guarantee that i will be between 0 and 99

Type Conversion and Coercion

Type conversion

The process of converting data from one type to another type is known as type conversion. Often if different parts of an expression are of different types then type conversion is required.

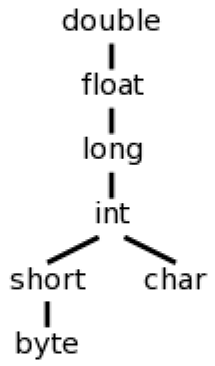
For example, in the expression: *z = x + y what is the type of z if x is integer and y is real? Compiler have to convert one of them to ensure that both operand of same type!*

In many language *Type conversion* is explicit, for example using type casts i.e. must be specify as *inttoreal(x)*

Coercion

The process of converting one type to another by compiler itself is known as coercion. Type conversion which happens implicitly is called coercion. Implicit type conversions are carried out by the compiler recognizing a type incompatibility and running a type conversion routine (for example, something like inttoreal (int)) that takes a value of the original type and returns a value of the required type.

Mathematically the hierarchy on the right is a partially order set in which each pair of elements has a least upper bound. For many binary operators (all the arithmetic ones we are considering, but not exponentiation) the two operands are converted to the LUB. So adding a short to a char, requires both to be converted to an int. adding a byte to a float, requires the byte to be converted to a float (the float remains a float and is not converted).

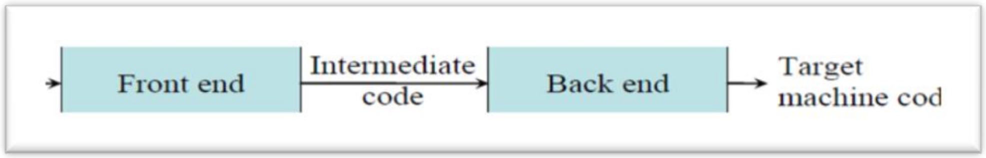


……………………………………………………………………………………………………..

Unit - 4

Intermediate Code Generation

The front end translates the source program into an intermediate representation from which the backend generates target code. Intermediate codes are machine independent codes, but they are close to machine instructions.



Advantages of using Intermediate code representation:

- If a compiler translates the source language to its target machine language without having the option for generating intermediate code, then for each new machine, a full native compiler is required.
- Intermediate code eliminates the need of a new full compiler for every unique machine by keeping the analysis portion same for all the compilers.
- The second part of compiler, synthesis, is changed according to the target machine.
- It becomes easier to apply the source code modifications to improve code performance by applying code optimization techniques on the intermediate code.

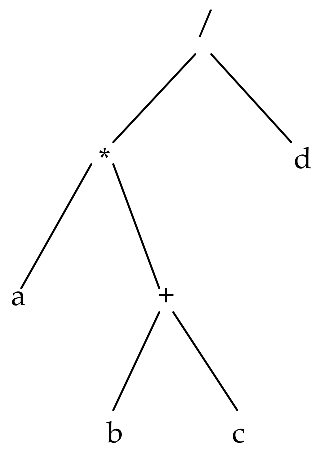
Intermediate Representations

There are three kinds of intermediate representations:

1. *Graphical representations* (e.g. Syntax tree or Dag)
2. *Postfix notation*: operations on values stored on operand stack (similar to JVM byte code)
3. *Three-address code*: (e.g. triples and quads) Sequence of statement of the form $x = y$ op z

Syntax tree:

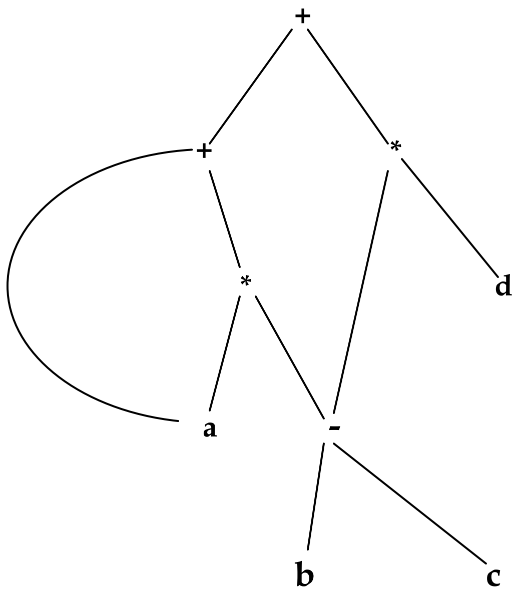
Syntax tree is a graphic representation of given source program and it is also called variant of parse tree. A tree in which each leaf represents an operand and each interior node represents an operator is called syntax tree.
Example: Syntax tree for the expression a*(b + c)/d



Directed acyclic graph (DAG)

A DAG for an expression identifies the common sub expressions in the expression. It is similar to syntax tree, only difference is that a node in a DAG representing a common sub expression has more than one parent, but in syntax tree the common sub expression would be represented as a duplicate sub tree.

Example: DAG for the expression a + a * (b - c) + (b - c) * d



Postfix notation

The representation of an expression in operators followed by operands is called postfix notation of that expression. In general if x and y be any two postfix expressions and OP is a binary operator then the result of applying OP to the x and y in postfix notation by "x y OP".

Examples:

1. (a+ b) * c in postfix notation is: a b + c *

2. a * (b + c) in postfix notation is: a b c + *

- Postfix notation is the useful form of intermediate code if the given language is expressions.
- Postfix notation is also called as 'suffix notation' and 'reverse polish'.
- Postfix notation is a linear representation of a syntax tree.
- In the postfix notation, any expression can be written unambiguously without parentheses.
- The ordinary (infix) way of writing the sum of x and y is with operator in the middle: x * y. But in the postfix notation, we place the operator at the right end as xy *.
- In postfix notation, the operator follows the operand.

Three Address Code

The address code that uses three addresses, two for operands and one for result is called three code. Each instruction in three address code can be described as a 4-tuple: (operator, operand1, operand2, result).

A quadruple (Three address code) is of the form:

x **=** *y* ***op*** *z w*here *x*, *y* and *z* are names, constants or compiler-generated temporaries and ***op*** is any operator.

We use the term "three-address code" because each statement usually contains three addresses (two for operands, one for the result). Thus the source language like *x + y * z* might be translated into a sequence

*t1 = y * z*

t2 = x + t1 where *t1* and *t2* are the compiler generated temporary name.

- Assignment statements: *x* **=** *y op z, op is binary*
- Assignment statements: *x* **=** *op y, op is unary*
- Indexed assignments: *x* **=** *y*[*i*], *x*[*i*] **=** *y*
- Pointer assignments: *x* **= &***y*, *x* **=** **y*, **x* **=** *y*
- Copy statements: *x* **=** *y*
- Unconditional jumps: **goto** *label*
- Conditional jumps: **if** *x relop y* **goto** *label*
- Function calls: **param** *x…* **call** *p, n* **return** *y*

Example: Three address code for expression: (B+A)*(Y-(B+A))

```
t1 = B + A
t2 = Y - t1
```

t3 = t1 * t2

Example 2: Three address code for expression:

i* = 2 * *n* + *k

While i do

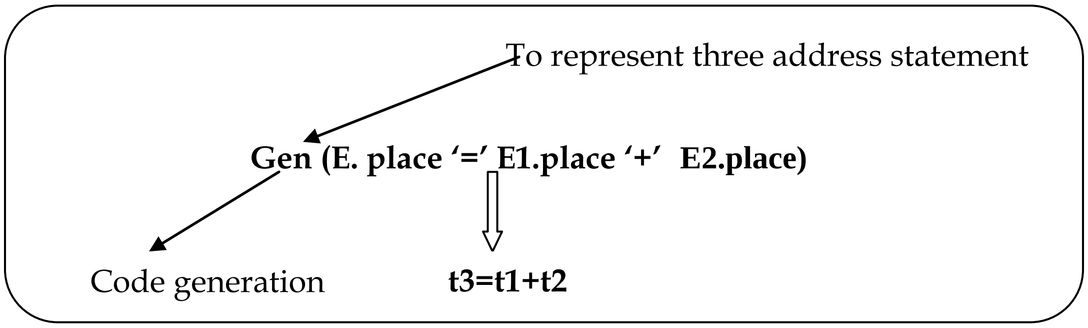
i = i - k

```
Solution:     t1 = 2

              t2 = t1 * n
              t3 = t2 + k
              i = t3
      L1: if i = 0 goto L2
              t4 = i - k
              i = t4
              goto L1
      L2: ………………..
```

<u>Naming conventions for three address code</u>

- *S*.code →three-address code for evaluating *S*
- *S*.begin → label to start of *S* or nil
- *S*.after →label to end of *S* or nil
- *E*.code → three-address code for evaluating *E*
- *E*.place → a name that holds the value of *E*


To represent three address statement
Gen (E. place '=' E1.place '+' E2.place)
Code generation
t3=t1+t2

Syntax-Directed Translation into Three-Address Code

1. <u>Assignment statements</u>

Productions	**Semantic rules**
S → **id** = *E*	*S*.code = *E*.code || *gen* (**id**.place '=' *E*.place); *S*.begin = *S*.after = nil
E → *E*1 **+** *E*2	*E*.place = *newtemp*();
	E.code = *E*1.code || *E*2.code || *gen* (*E*.place '=' *E*1.place '+' *E*2.place)
E → *E*1 * *E*2	*E*.place = *newtemp*();
	E.code = *E*1.code || *E*2.code || *gen* (*E*.place '=' *E*1.place '*' *E*2.place)

$E \rightarrow -\ E1$	E.place = *newtemp*(); E.code = $E1$.code || *gen* (E.place '=' 'minus' $E1$.place)
$E \rightarrow$ **(** $E1$ **)**	E.place = $E1$.place E.code = $E1$.code
$E \rightarrow$ **id**	E .place = **id**.name E.code = null
$E \rightarrow$ **num**	E.place = *newtemp*(); E.code = *gen*(E.place '=' **num**.value)

2. Boolean Expressions

Boolean expressions are used to compute logical values. They are logically used as conditional expressions in statements that alter the flow of control, such as if—then, if—the—else or while---do statements.

Control-Flow Translation of Boolean Expressions

Production	Semantic Rules
$B \rightarrow B_1 \parallel B_2$	B_1.true = B.true B_1.false = newlabel() B_2.true = B.true B_2.false = B.false B.code = B_1.code || label(B1.false) || B_2.code
$B \rightarrow B_1$ && B_2	B_1.true = newlabel() B_1.false = B.false B_2.true = B.true B_2.false = B.false B.code = B_1.code || label(B1.true) || B_2.code
$B \rightarrow ! B_1$	B_1.true = B.false B_1.false = B.true B.code = B_1.code
$B \rightarrow E_1$ relop E_2	B.code = E_1.code || E_2.code || gen(if E_1.addr relop.lexeme E_2.addr goto B.true) || gen(goto B.false)
$B \rightarrow$ true	B.code = gen(goto B.true)
$B \rightarrow$ false	B.code = gen(goto B.false)

3. Flow of control statements

Control statements are 'if—then', 'if—then—else', and 'while---do'. Control statements are generated by the following grammars:

$S \rightarrow$ If exp then S1

$S \rightarrow$ If exp then S1 else S2

$S \rightarrow$ while exp do S1

Production	**Semantic Rules**

$S \rightarrow$ if (B) S_1	B.true = newlabel()
	B.false = S.next
	S_1.next = S.next
	S.code = B.code || label(B.true) || S_1.code

$S \rightarrow$ if (B) S_1 else S_2	B.true = newlabel()
	B.false = newlabel()
	S_1.next = S.next
	S_2.next = S.next
	S.code = B.code || label(B.true) || S_1.code || gen(goto S.next) || label(B.false) || S_2.code

$S \rightarrow$ while (B) S_1	begin = newlabel()
	B.true = newlabel()
	B.false = S.next
	S_1.next = begin
	S.code = label(begin) || B.code || label(B.true) || S_1.code || gen(goto begin)

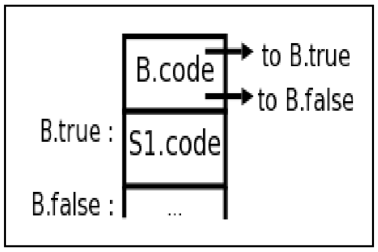
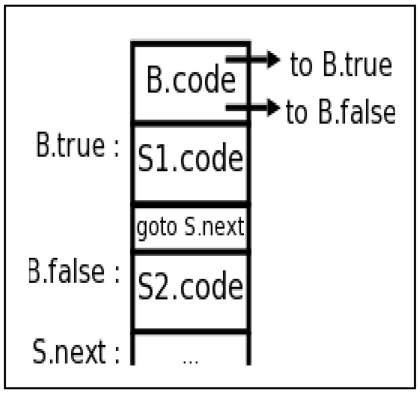


Fig: If--then



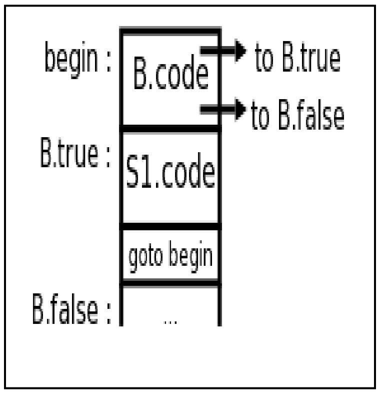


Fig: If—then—else

Fig: while---do

Example: Generate three address code for the expression

if (x < 5 | | (x > 10 && x == y)) x = 3 ;

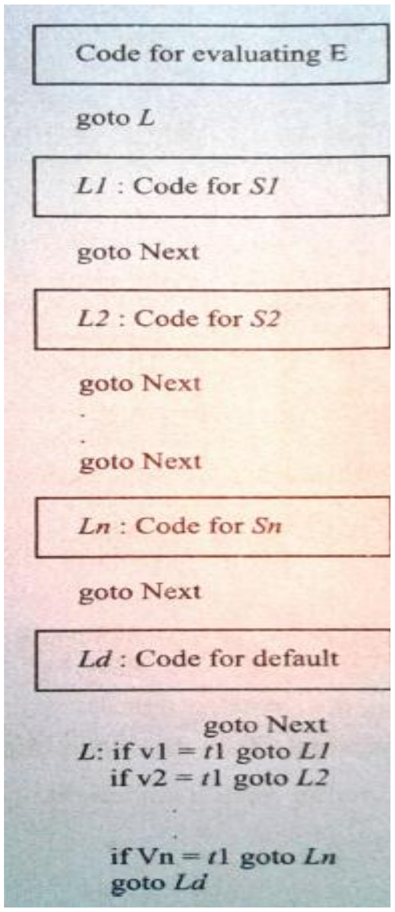
Solution:

```
L1: if x < 5 goto L2
      goto L3
L3: if x > 10 goto L4
        goto L1
  L4: if x == y goto L2
        goto L1
  L2: x = 3
```

Switch/ case statements

A switch statement is composed of two components: an expression E, which is used to select a particular case from the list of cases; and a case list, which is a list of n number of cases, each of which corresponds to one of the possible values of the expression E, perhaps including a default value.

```
Switch (E)
{
      Case V1: S1
      Case V2: S2
      ………………
      ……………….
      Case Vn : Sn
}
```



```
Code for evaluating E
goto L
L1 : Code for S1
goto Next
L2 : Code for S2
goto Next
.
.
goto Next
Ln : Code for Sn
goto Next
Ld : Code for default
            goto Next
L: if v1 = t1 goto L1
   if v2 = t1 goto L2
   .
   if Vn = t1 goto Ln
   goto Ld
```

Fig: A switch / case three address translation

Example: Convert the following switch statement into three address code:

```
Switch (i + j)
{
        Case 1: x=y + z
        Case 2: u=v + w
        Case 3: p=q * w
        Default: s=u / v
    }
```

Solution:

```
    L1: t1=i + j
    L2: goto(L15)
    L3: t2= y+z
    L4: x=t2
    L5: goto(L19)
    L6: t3=v+w
    L7: u=t3
    L8: goto (L19)
    L9: t4=q*w
    L10: p=t4
    L11: goto (L19)
    L12: t5= u/v
    L13: s=t5
    L14: goto(L19)
    L15: if t1=1 goto(L3)
    L16: if t1=2 goto(L6)
    L17: if t1=3 goto(L9)
    L18: goto(L12)
    L19: Next
```

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,

$$\boxed{\textbf{base + (i – low)* w}}$$

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].

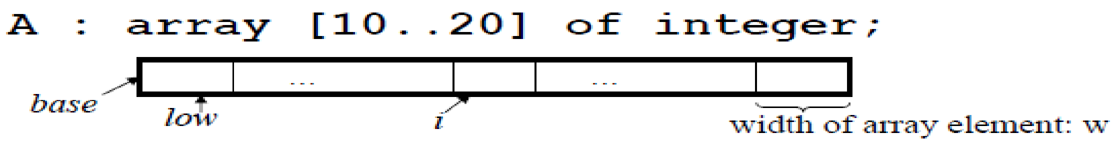
The given expression can be partially evaluated at compile time if it is rewritten as,

$$\mathbf{i * w + (base - low* w)}$$
$$\mathbf{= i * w + C}$$

Where 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.

i.e A[i]= i* w + C



$A[i] = \text{base}_A + (i - \text{low}) * w$

$= i * w + c$ **where** $c = \text{base}_A - \text{low} * w$ **with** low = 10; w = 4

Example: address of 15th element of array is calculated as below,

Suppose base address of array is 100 and type of array is integer of size 4 bytes and lower bound of array is 10 then,

A[15]=15 * 4 + (100 – 10 * 4)

= 60 + 60

= 120

Similarly for two dimensional array, we assume that array implements by using row major form, the relative address of $A[i_1, i_2]$ can be calculated by the formula,

$$A[i_1,i_2]=\text{base}_A + ((i_1 - \text{low}_1) * n_2 + i_2 - \text{low}_2) * w$$

Where low_1, low_2 are the lower bounds on the values i_1 and i_2, n_2 is the number of values that i_2 can take. Also given expression can be rewrite as,

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

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

Example: Let A be a 10 X 20 array, there are 4 bytes per word, assume low1=low2=1.

Solution: Let X=A[Y, Z]

Now using formula for two dimensional array as,

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

$= ((Y * 20) + Z) * 4 + \text{base}_A - ((1 * 20) + 1) * 4$

$= ((Y * 20) + Z) * 4 + \text{base}_A - ((1 * 20) + 1) * 4$

$= ((Y * 20) + Z) * 4 + \text{base}_A – 84$

We can convert the above expression in three address codes as below:

T1= Y * 20

T1= T1+Z

T2=T1*4

T3=base_A -84

T4=T2+ T3
X= T4

4. Declarative statements

We can explain the declarative statement by using the following example,

$S \rightarrow D$	{offset=0}
$D \rightarrow$ id : T	{enter-to-symbol-table(id.name, T.type, offset); (offset= offset+T.width)}
$T \rightarrow$ integer	{T.type=Integer; T.width=4}
$T \rightarrow$ real	{T.type=real; T.width=8}
$T \rightarrow$ array [num] of T_1	{T.type=array (num.val, T_1.type) T.width=num.val * T_1.width}
$T \rightarrow \uparrow$T1	{ T.type=pointer(T1.type); T.width=4}

- Initially offset is set to zero. As each new name is seen, that name is entered in the symbols 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-to-symbol-table (name, type, offset) creates a symbol table entry for name, gives it type and relative address offset in its data area.
- Integers have width 4 and reals have width 8. 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.

5. Procedure Calls

The procedure is such an important and frequently used programming construct that is imperative for a compiler to generate good code for procedure calls and returns. Consider a grammar for a simple procedure call statement:

S → **call id (**Elist **)**
Elist → Elist
Elist → E

When procedure call occurs, space must be allocated for the activation record of the called procedure. The argument of the called procedure must be evaluated and made available to the called procedure in a known place. The return address is usually location of the instruction that follows the call in the calling procedure. Finally a jump to the beginning of the code for the called procedure must be generated.

6. Back patching

If we decide to generate the three address code for given syntax directed definition using single pass only, then the main problem that occurs is the decision of addresses of the labels. ‘goto’ statements refer these label statements and in one pass it becomes difficult to know the location of these label statements. The idea to back-patching is to leave the label unspecified and fill it later, when we know what it will be.

If we use two passes instead of one pass then in one pass we can leave these addresses unspecified and in second pass this incomplete information can be filled up.

Exercise: Generate three address codes for the following statements:

```
While (a<c and b<d)
{
   If(a==1 ) then
          c=c+1
   else
        while (a<=d)
                a=a+3
}
```

……..

Unit -4 [Chapter-2]

Code Generation and optimization

The process of transform intermediate code + tables into final machine (or assembly) code is known as code generation. The process of eliminating unnecessary and inefficient code such as dead code, code duplication etc from the intermediate representation of source code is known as code optimization. Code generation + Optimization are the back end of the compiler.

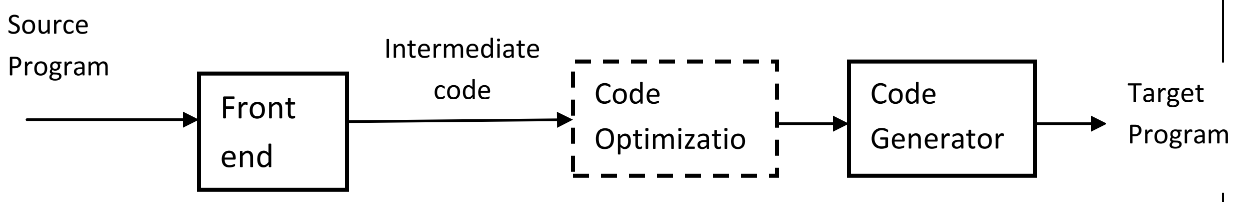


Fig: Place of Code generator in compilation process

Code generator design Issues

The code generator mainly concern with:

- ♠ Input to the code generator
- ♠ Target program
- ♠ Target machine
- ♠ Instruction selection
- ♠ Register allocation (Storage allocation)
- ♠ Choice of evaluation order

1. Input to the Code Generator

The input to the code generator is intermediate representation together with the information in the symbol table.

2. The Target Program

The output of the code generator is target code. Typically, the target code comes in three forms such as: absolute machine language, relocatable machine language and assembly language.

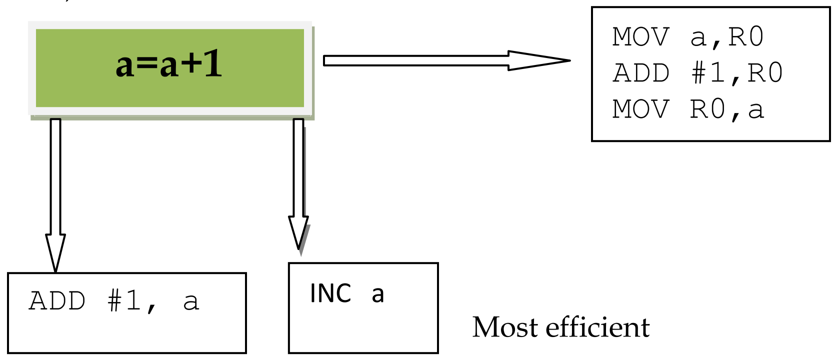
The advantage of producing target code in absolute machine form is that it can be placed directly at the fixed memory location and then can be executed immediately. The benefit of such target code is that small programs can be quickly compiled.

3. The Target Machine

Implementing code generation requires thorough understanding of the target machine architecture and its instruction set.

4. Instruction Selection

Instruction selection is important to obtain efficient code. Suppose we translate three-address code,



```
MOV a,R0
ADD #1,R0
MOV R0,a
```

```
ADD #1, a
```

```
INC a
```

5. Register Allocation

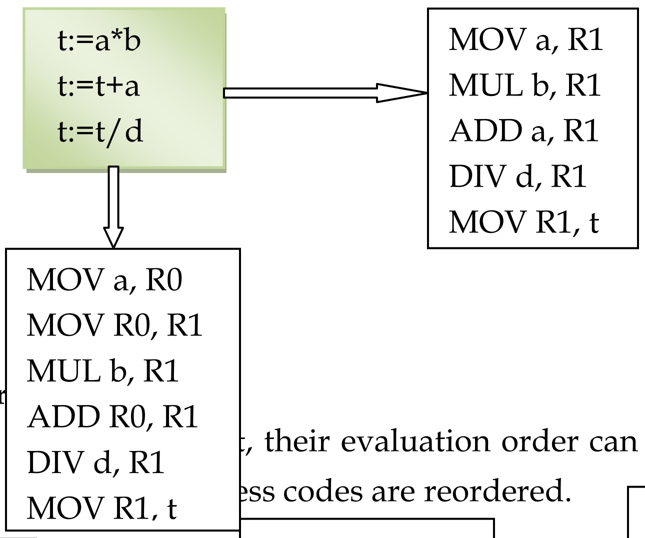
Since registers are the fastest memory in the computer, the ideal solution is to store all values in registers. However, there are normally not nearly enough registers for this to be possible. So we must choose which values are in the registers at any given time.

Actually this problem has two parts.

1. Which values should be stored in registers?
2. Which register should each selected value be stored in

The reason for the second problem is that often there are register requirements, e.g., floating-point values in floating-point registers and certain requirements for even-odd register pairs for multiplication/division.

Example



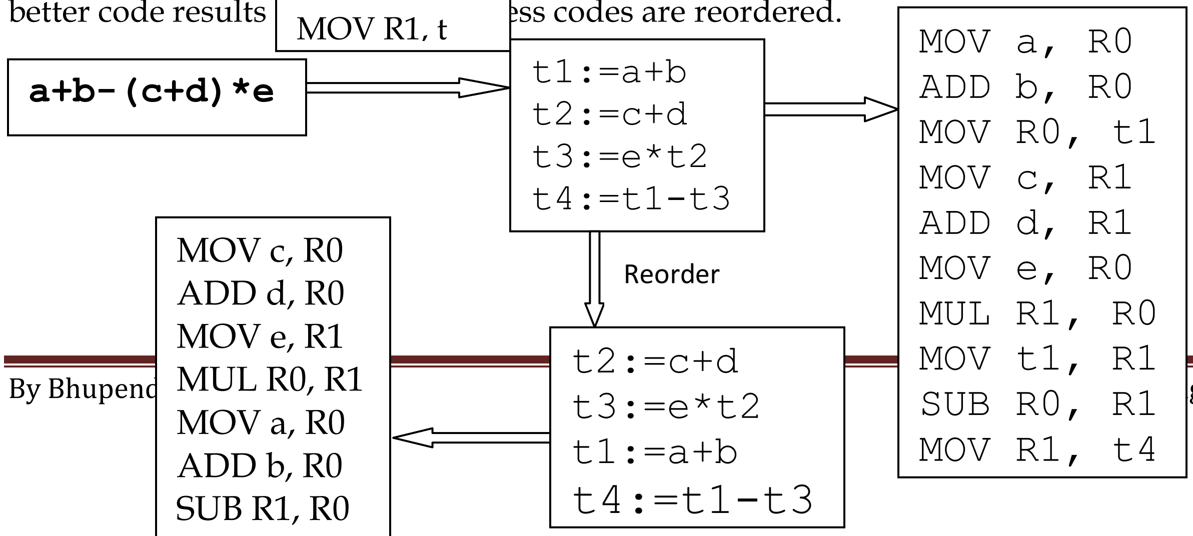
```
t:=a*b
t:=t+a
t:=t/d
```

```
MOV a, R1
MUL b, R1
ADD a, R1
DIV d, R1
MOV R1, t
```

```
MOV a, R0
MOV R0, R1
MUL b, R1
ADD R0, R1
DIV d, R1
MOV R1, t
```

6. Evaluation Order

When instructions , their evaluation order can be changed. Sometimes better code results ess codes are reordered.



```
a+b-(c+d)*e
```

```
t1:=a+b
t2:=c+d
t3:=e*t2
t4:=t1-t3
```

```
MOV a, R0
ADD b, R0
MOV R0, t1
MOV c, R1
ADD d, R1
MOV e, R0
MUL R1, R0
MOV t1, R1
SUB R0, R1
MOV R1, t4
```

```
t2:=c+d
t3:=e*t2
t1:=a+b
t4:=t1-t3
```

```
MOV c, R0
ADD d, R0
MOV e, R1
MUL R0, R1
MOV a, R0
ADD b, R0
SUB R1, R0
```

Basic Blocks

A *basic block* is a sequence of consecutive instructions in which flow of control enters by one entry point and exit to another point without halt or branching except at the end.

Example:

```
        MOV 1, R0
        ADD n, R0
        MOV 2, R1
        MUL R0, R1
        JMP L2
L1:     MUL 2, R0
        SUB 1, R1
L2:     MUL 3, R1
        JMPNZ R1, L1
```

```
MOV 1, R0
ADD n, R0
MOV 2, R1
MUL R0, R1
JMP L2
```

```
L1: MUL 2, R0
    SUB 1, R1
```

```
L2: MUL 3, R1
    JMPNZ R1, L1
```

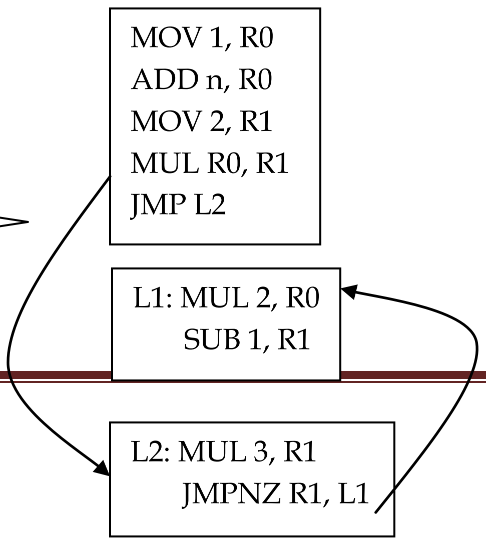
Flow Graphs

A flow graph is a graphical depiction of a sequence of instructions with control flow edges. A flow graph can be defined at the intermediate code level or target code level. The nodes of flow graphs are the basic blocks and flow-of-control to immediately follow node connected by directed arrow.

Simply, if flow of control occurs in basic blocks of given sequence of instructions then such group of blocks is known as flow graphs.

Example:

```
        MOV 1, R0
        ADD n, R0
        MOV 2, R1
        MUL R0, R1
        JMP L2
L1:     MUL 2, R0
        SUB 1, R1
```



```
MOV 1, R0
ADD n, R0
MOV 2, R1
MUL R0, R1
JMP L2
```

```
L1: MUL 2, R0
    SUB 1, R1
```

```
L2: MUL 3, R1
    JMPNZ R1, L1
```

```
L2:    MUL 3, R1
       JMPNZ R1, L1
```

Equivalence of basic blocks

Two basic blocks are (semantically) *equivalent* if they compute the same set of expressions.

```
b = 0
t1 = a + b
t2 = c * t1
a = t2
```

$\Downarrow$

```
a= c*a
b=0
```

```
a= c*a

b=0
```

$\Downarrow$

```
a= c*a
b=0
```

Transformations on Basic Blocks

The process of code optimization to improve speed or reduce code size of given sequence of codes and convert to basic blocks by shafting and preserving the meaning of the code is known as transformation on basic blocks. There are mainly two types of code transformations:

- ♠ Global transformations and
- ♠ Local transformations

Global transformations are performed across basic blocks whereas *Local transformations* are only performed on single basic blocks. A local transformation is safe if the transformed basic block is guaranteed to be equivalent to its original form. There are two classes of local transformations which are:

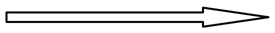
1. Structure preserving transformations and
    - ♠ Common sub-expression elimination
    - ♠ Dead code elimination
    - ♠ Renaming of temporary variables
    - ♠ Interchange of two independent statements
2. Algebraic transformations

Common sub expression elimination

Here we eliminate sub expressions that do not impact on our resultant basic block.

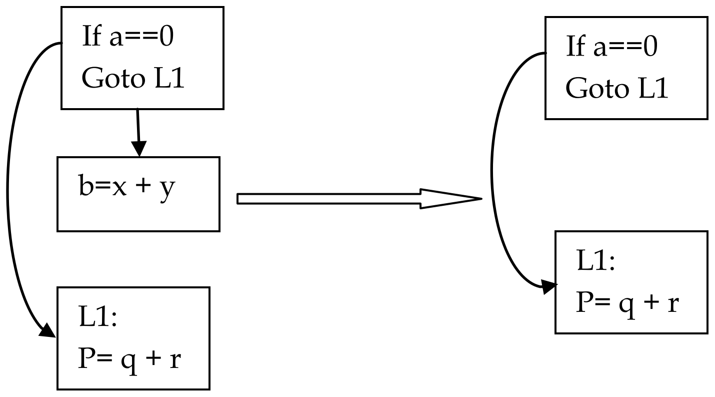
```
t1 = b * c
t2 = a - t1
t3 = b * c
t4 = t2 + t3
```

```
t1 = b * c
t2 = a - t1
t3 = t2 + t1
```



Dead code elimination

Here we remove unused expressions.



```
If a==0
Goto L1

b=x + y

L1:
P= q + r
```

```
If a==0
Goto L1

L1:
P= q + r
```

Renaming Temporary variables

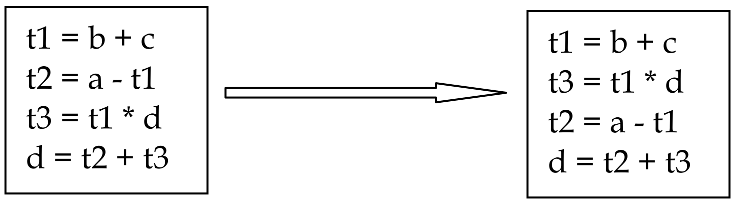
Temporary variables that are dead at the end of a block can be safely renamed.

Let we have a statement t1=a + b where t1 is a temporary variable.

If we change this statement to t2= a + b where t2 is a new temporary variable. Then the value of basic block is not changed and such new equivalent statement of their original statement is called ***normal-form block***.

Interchange of statements

Independent statements can be reordered without affecting the value of block to make its optimal use.

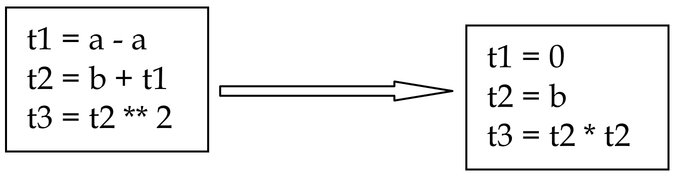


```
t1 = b + c
t2 = a - t1
t3 = t1 * d
d = t2 + t3
```

```
t1 = b + c
t3 = t1 * d
t2 = a - t1
d = t2 + t3
```

Algebraic Transformations

Change arithmetic operations to transform blocks to algebraic equivalent forms. Here we replace expansive expressions by cheaper expressions.



```
t1 = a - a
t2 = b + t1
t3 = t2 ** 2
```

```
t1 = 0
t2 = b
t3 = t2 * t2
```

Next use information

Next-use information is needed for dead-code elimination and register assignment (if the name in a register is no longer needed, then the register can be assigned to some other name).

If *i: x* = … and *j: y* = *x* + *z* are two statements *i* & *j*, then *next-use* of *x* at *i* is *j*.

Next-use is computed by a backward scan of a basic block and performing the following actions on statement

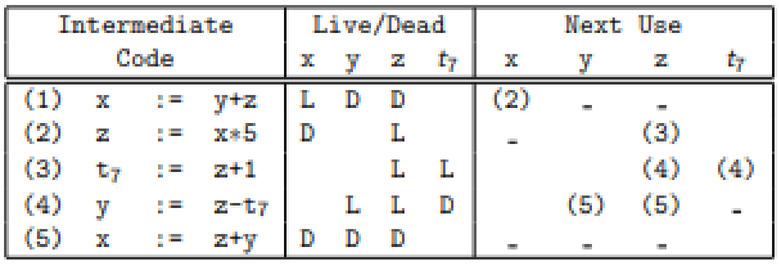
i*: *x* = *y* op *z

Add liveness /next-use info on *x*, *y*, and *z* to statement *i* (whatever in the symbol table)

Before going up to the previous statement (scan up):

- ♠ Set *x* info to "not live" and "no next use"
- ♠ Set *y* and *z* info to "live" and the next uses of *y* and *z* to *i*

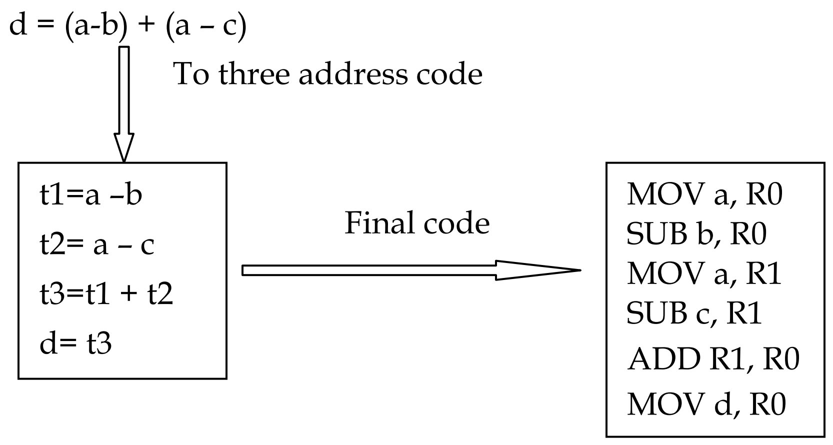
Example:



Intermediate Code				Live/Dead				Next Use			
				x	y	z	t_7	x	y	z	t_7
(1)	x	:=	y+z	L	D	D		(2)	-	-	
(2)	z	:=	x*5	D		L		-		(3)	
(3)	t_7	:=	z+1			L	L			(4)	(4)
(4)	y	:=	z-t_7		L	L	D		(5)	(5)	-
(5)	x	:=	z+y	D	D	D		-	-	-	

Code generator

The code generator converts the optimized intermediate representation of a code to final code which is normally machined dependent.



d = (a-b) + (a – c)

To three address code

```
t1=a –b
t2= a – c
t3=t1 + t2
d= t3
```

Final code

```
MOV a, R0
SUB b, R0
MOV a, R1
SUB c, R1
ADD R1, R0
MOV d, R0
```

Register Descriptors

A *register descriptor* keeps track of what is currently stored in a register at a particular point in the code, e.g. a local variable, argument, global variable, etc.

MOV a, R0 "**R0** contains **a**"

Address Descriptors

An *address descriptor* keeps track of the location where the current value of the name can be found at run time, e.g. a register; stack location, memory address, etc.

MOV a, R0
MOV R0, R1 "**a** in **R0** and **R1**"

Example 1: At first convert the following expression into three address code sequence then show the register as well as address descriptor contents.

Statement: d = (a - b) + (a – c) + (a – c)

Solution: The three address code sequence of above statement is:

t1=a –b
t2= a – c
t3=t1 + t2
d= t3+t2

Statements	Code generated	Resister descriptor	Address Descriptor
t1=a –b	MOV a, R0 SUB b, R0	R0 contains t1	t1 in R0
t2= a – c	MOV a, R1 SUB c, R1	R0 contains t1 R1 contains t2	t1 in R0 t2 in R1
t3=t1 + t2	ADD R1, R0	R0 contains t3 R1 contains t2	t3 in R0 t2 in R1
d= t3+t2	ADD R1, R0 MOV d, R0	R0 contains d	d in R0 and memory

Example 2: At first convert the following expression into three address code sequence then show the register as well as address descriptor contents.

Statement: X = (a / b + c) – d * e

Solution: The three address code sequence of above statement is:

t1=a /b
t2= t1 + c
t3=d * e
X= t2 – t3

Statements	Code generated	Resister descriptor	Address Descriptor
t1=a /b	MOV a, R0 DIV b, R0	R0 contains t1	t1 in R0
t2= t1 + c	ADD c, R0	R0 contains t2	t2 in R0
t3=d * e	MOV d, R1 MUL e, R1	R0 contains t2 R1 contains t3	t2 in R0 t3 in R1
X= t2 – t3	SUB R1, R0 MOV X, R0	R0 contains X R1 contains t3	X is in R0 and in memory

Code optimization

[Q]. Explain various optimization approaches (Peephole optimizations) with suitable example for each of the approach.

Ans: The various optimization approaches are listed below:

* Redundant instruction elimination
* Flow of control optimization
* Dead code elimination
* Common sub expression elimination
* Algebraic simplification
* Use of machine idioms
* Reduction in strength
* Loop optimization
* Loop invariant optimization

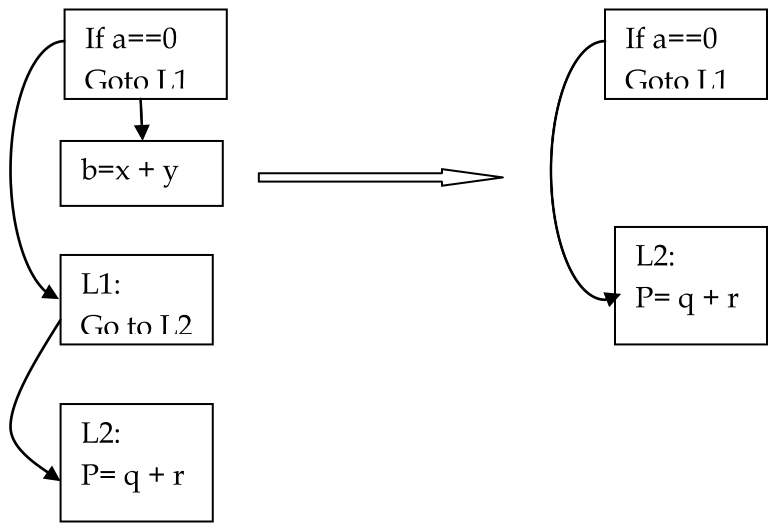
Redundant instruction elimination

Consider an instruction sequence

I. MOV R0, a
II. MOV a, Ro

Here, we can delete instruction (II) only when both of them are in the same block. The first instruction can also be deleted if live (a) = false.

Flow of control optimization



Use of machine idioms

If the addressing mode *INC* is defined then we can replace given expression into their equivalent efficient form as below,

```
………………                ………………
   a= a+1        ⟹          INC a
Total cost=3            Total cost=2
```

Reduction in strength

Replace expensive arithmetic operations with corresponding cheaper expressions.

```
………………                ………………
   a= x^2                   a= x * x
   b= y/8        ⟹         b= y>>3
…………..                  ………………
```

Loop optimization

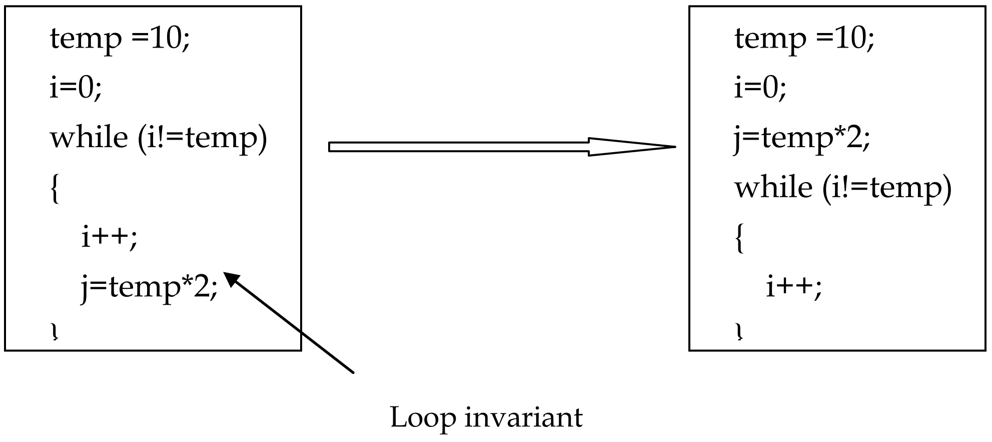
If a complex expression is used as condition to the loop then such a expression can be reduced into smaller sub expressions outside the loop and only their final result used to the looping as a condition check. Such an approach is called loop optimization.

```
                                t1=limit*2
While (i<=limit* 2-10)   ⟹     t2=t1-10
                                While (i<=t2)
```

Loop invariant optimization

A code fragment inside the loop is said to be loop invariant if its computation does not depends on the loop. Such fragment can be removed from the loop and can be computed before the loop execution as below,

Eg:



```
temp =10;
i=0;
while (i!=temp)
{
   i++;
   j=temp*2;
}
```

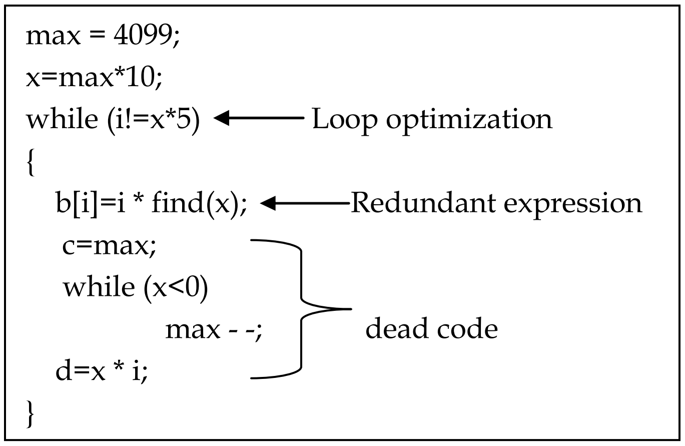
```
temp =10;
i=0;
j=temp*2;
while (i!=temp)
{
   i++;
}
```

Example 1: On the following piece of code,

```
max = 4099;
x=max*10;
while (i!=x*5)
{
   b[i]=i * find(x);
    c=max;
    while (x<0)
              max - -;
   d=x * I;
}
```

Identify different kinds of optimizations possible and describe them. Rewrite the code after making optimizations.

Solution:



```
max = 4099;
x=max*10;
while (i!=x*5)
{
   b[i]=i * find(x);
    c=max;
    while (x<0)
              max - -;
   d=x * i;
}
```