

Compilers

A **compiler** is a program that can read a program in one language - the **source language** - and translate it into an equivalent program in another language - the **target language**;

Note - An important role of the compiler is to report any errors in the source program that it detects during the translation process.

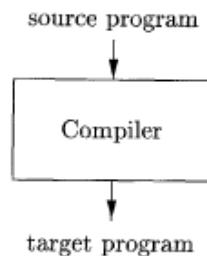


Figure 1.1: A compiler

If the target program is an executable machine-language program, it can then be called by the user to process inputs and produce outputs.

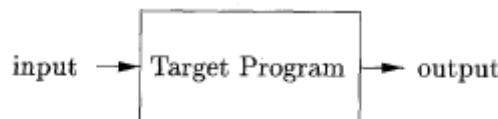


Figure 1.2: Running the target program

Interpreter

An interpreter is another common kind of **language processor**. Instead of producing a target program as a translation, an interpreter appears to directly execute the operations specified in the source program on inputs supplied by the user.

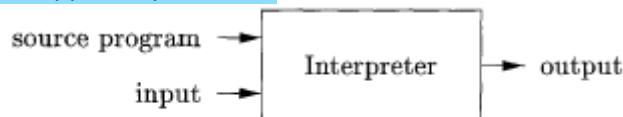


Figure 1.3: An interpreter

Note –

1. The machine-language target program produced by a **compiler** is usually much faster than an interpreter at mapping inputs to outputs.
2. An **interpreter**, however, can usually give better error diagnostics than a compiler, because it executes the source program statement by statement.

Hybrid Compiler

Java language processors combine compilation and interpretation, as shown below

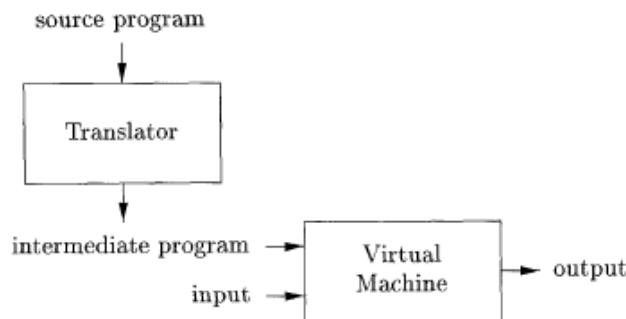


Figure 1.4: A hybrid compiler

A Java source program may first be compiled into an **intermediate form** called **bytecodes**. The bytecodes are then interpreted by a virtual machine. A benefit of this arrangement is that **bytecodes compiled on one machine can be interpreted on another machine**.

Language Processing System

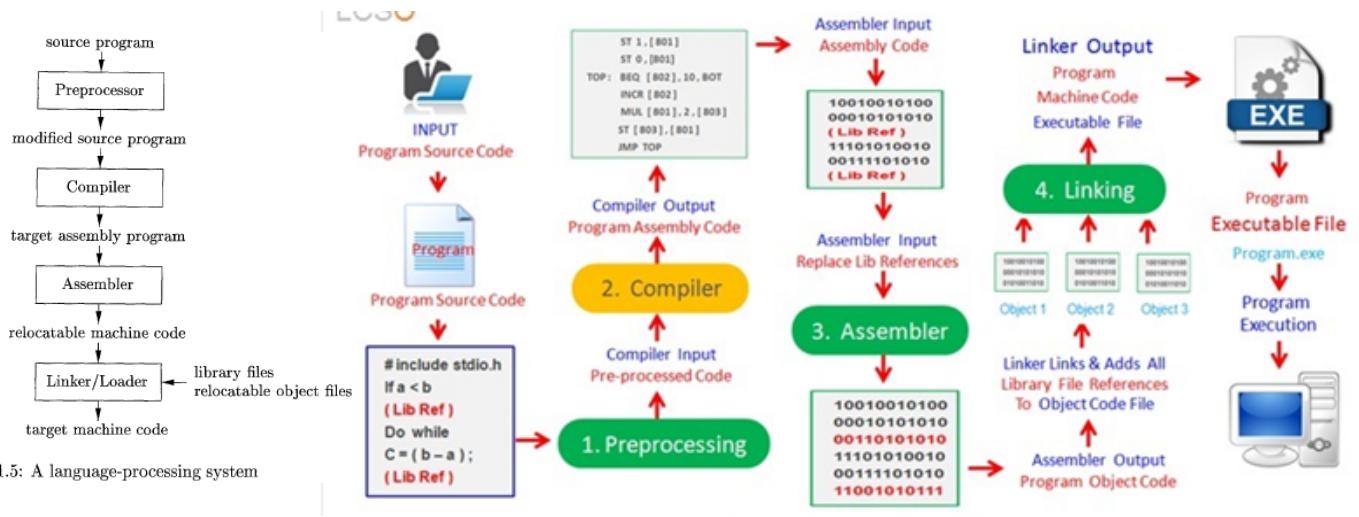


Figure 1.5: A language-processing system

1. A **source program** may be divided into modules stored in separate files. The task of collecting the source program is sometimes entrusted to a separate program, called a **preprocessor**.
2. The **preprocessor** may also expand shorthands, called macros, into source language statements. Preprocessor allows user to use header file but it does not link the object modules.
3. The **modified source program** is then fed to a **compiler**.
4. The **compiler** may produce an **assembly-language program** as its output because **assembly language is easier to produce as output and is easier to debug**.
5. The assembly language is then processed by a program called an **assembler** that produces **relocatable machine code** as its output.
6. Large programs are often compiled in pieces, so the **relocatable machine code may have to be linked together with other relocatable object files and library files into the code** that actually runs on the machine.
7. The **linker** resolves external memory addresses, where the code in one file may refer to a location in another file.
8. The **loader** then puts together all of the executable object files into memory for execution. Loader can perform linking.

The Structure of a Compiler

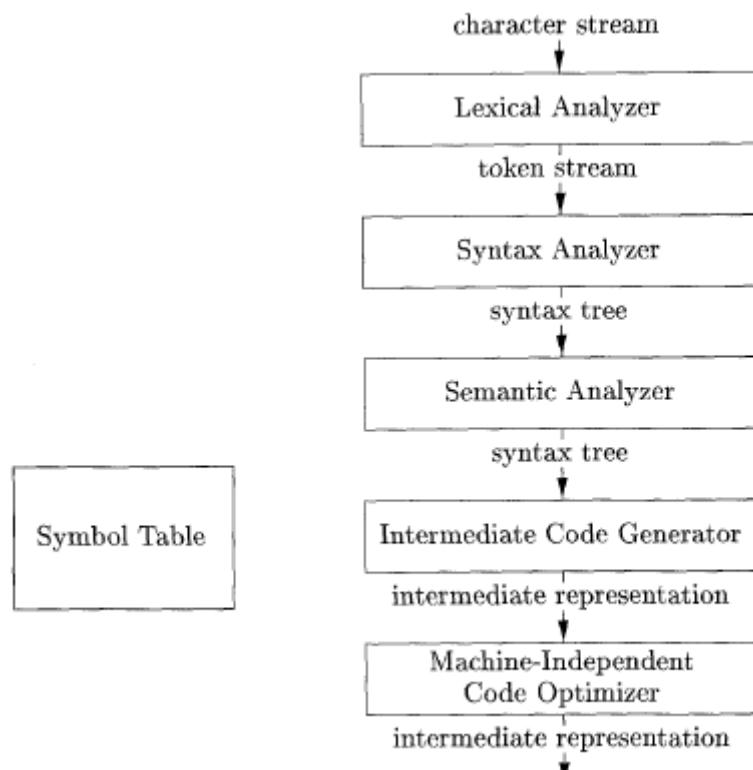
Up to this point we have treated a compiler as a single box that maps a source program into a semantically equivalent target program. If we open up this box a little, we see that there are two parts to this mapping: **analysis** and **synthesis**.

The analysis part

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

The synthesis part

1. It constructs the desired target program from the intermediate representation and the information in the symbol table.
2. The synthesis part is the **back end**.
3. **Symbol table** is a data structure created and maintained by compiler in order to store information about the occurrence of various entities such as variable names, function names, objects, classes, interfaces, address etc.



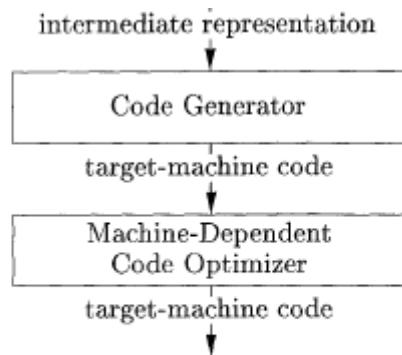


Figure 1.6: Phases of a compiler

Note –

1. **The symbol table, which stores information about the entire source program, is used by all phases of the compiler.**
2. Some compilers have a **machine-independent optimization** phase between the front end and the back end.

Lexical Analysis

The first phase of a compiler is called **lexical analysis** or **scanning**. The **lexical analyzer** reads the **stream of characters** making up the source program and **groups the characters into meaningful sequences called lexemes**.

For each lexeme, the lexical analyzer produces as output a token of the form

(token-name, attribute-value)

That it passes on to the subsequent phase, **syntax analysis**. In the token, the **first component token-name** is an abstract symbol that is used during **syntax analysis**, and the **second component attribute-value** points to an entry in the symbol table for this token.

For example, suppose a source program contains the assignment statement

position = initial + rate * 60

The characters in this assignment could be grouped into the following **lexemes** and mapped into the following **tokens** passed on to the **syntax analyzer**:

1. Position is a **lexeme** that would be mapped into a **token (id, 1)**, where **id** is an **abstract symbol** standing for **identifier** and 1 points to the symbol **table entry** for position.
2. The assignment symbol = is a lexeme that is mapped into the token (=). Since this token needs no attribute-value, we have omitted the second component. We could have used any abstract symbol such as **assign** for the token-name, but for notational convenience we have chosen to use the **lexeme itself as the name of the abstract symbol**.
3. **initial** is a lexeme that is mapped into the token **(id, 2)**, where 2 points to the symbol-table entry for initial .

4. `+` is a **lexeme** that is mapped into the token `(+)`.
5. `rate` is a **lexeme** that is mapped into the token `(id, 3)`, where 3 points to the symbol-table entry for `rate`.
6. `*` is a lexeme that is mapped into the token `(*)`.
7. `60` is a lexeme that is mapped into the token `(60)`

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

Lexical analysis as the sequence of tokens

`(id, 1) => (id, 2) (+) (id, 3) (*) (60)`

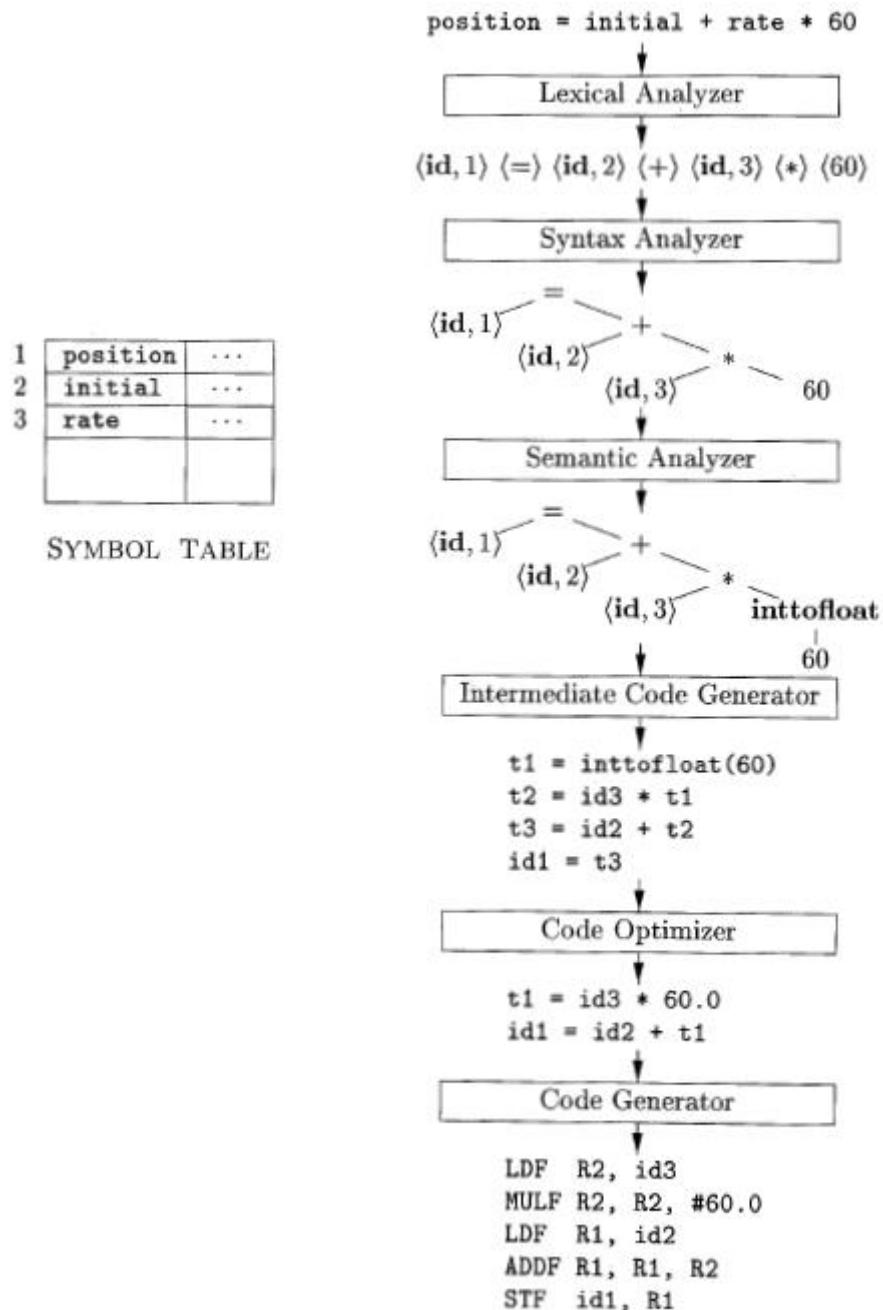


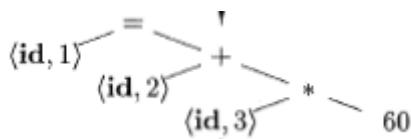
Figure 1.7: Translation of an assignment statement

Syntax Analysis

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

This tree shows the **order** in which the operations in the assignment are to be performed.

`position = initial + rate * 60`



The tree has an interior node labeled ***** with **(id, 3)** as its left child and the integer **60** as its right child. The node **(id, 3)** represents the identifier **rate**. The node labeled ***** makes it explicit that we must first multiply the value of **rate** by **60**.

Semantic Analysis

The **semantic analyzer** uses the **syntax tree** and the **information in the symbol table** to check the source program for semantic consistency with the **language definition**.

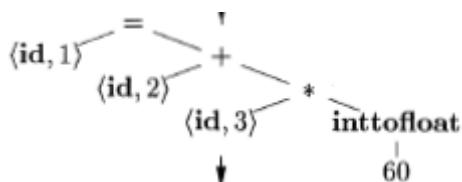
It also gathers type information and saves it in either the syntax tree or the symbol table, for subsequent use during intermediate-code generation.

An important part of semantic analysis is **type checking**, where the compiler checks that each operator has matching operands.

For example, many programming language definitions require an array index to be an **integer**; the compiler must report an error if a **floating-point number** is used to index an array.

Coercions

The language specification may permit some **type conversions** called **coercions**. For example a **binary arithmetic operator** may be applied to either a pair of integers or to a pair of floating-point numbers. **If the operator is applied to a floating-point number and an integer**, the compiler may **convert or coerce the integer into a floating-point number**.



Suppose that **position**, **initial**, and **rate** have been declared to be **floating-point numbers**, and that the lexeme **60** by itself forms an **integer**.

The type checker in the semantic analyzer discovers that the operator ***** is applied to a **floating-point number rate and an integer 60**.

In this case, the **integer may be converted into a floating-point number**.

Notice that the output of the semantic analyzer has an **extra node for the operator inttofloat**, which explicitly converts its integer argument into a floating-point number.

Intermediate Code Generation

In the process of translating a source program into target code, a compiler may construct one or more intermediate representations, which can have a variety of forms.

Syntax trees are a form of intermediate representation; they are commonly used during syntax and semantic analysis.

After syntax and semantic analysis of the source program, many compilers generate an explicit low-level or machine-like intermediate representation.

We consider an intermediate form called **three-address code**, which consists of a sequence of assembly-like instructions with three operands per instruction. Each operand can act like a register.

```

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

```

Each three-address assignment instruction has at most one operator on the right side. Thus, these instructions fix the order in which operations are to be done;

Code Optimization

The machine-independent code-optimization phase attempts to improve the intermediate code so that better target code will result. Code optimization can be done at source code, Intermediate code, target code.

A simple intermediate code generation algorithm followed by code optimization is a reasonable way to generate good target code.

```

t1 = id3 * 60.0
id1 = id2 + t1

```

Note – We have machine dependent optimization also. "Pass" in compiler refers to grouping of one or more phase as one module or phase. That's why we have one pass, two pass compiler.

Code Generation

The code generator takes as input an intermediate representation of the source program and maps it into the target language.

If the target language is machine code, registers or memory locations are selected for each of the variables used by the program. Then, the intermediate instructions are translated into sequences of machine instructions that perform the same task.

```

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

```

Symbol-Table Management

The symbol table is a data structure containing a record for each variable name, with fields for the attributes of the name. The data structure should be designed to allow the compiler to find the record for each name quickly and to store or retrieve data from that record quickly.

Identifiers/ Names/Variables

1. An **identifier** is a string of **characters**, typically **letters** or **digits** that refers to (**identifies**) an **entity**, such as a data object, a procedure, a class, or a type.
All identifiers are names, but not all names are identifiers.
2. Names can also be **expressions**. For example, the name **x.y** might denote the field **y** of a structure denoted by **x**. Here, **x** and **y** are identifiers, while **x.y** is a name, but not an identifier.
3. **A variable refers to a particular location of the store. It is common for the same identifier to be declared more than once; each such declaration introduces a new variable. Even if each identifier is declared just once, an identifier local to a recursive procedure will refer to different locations of the store at different times.**

Declarations and Definitions

1. **Declarations** tell us about the **types of things**.
2. While **definitions** tell us about **their values**.

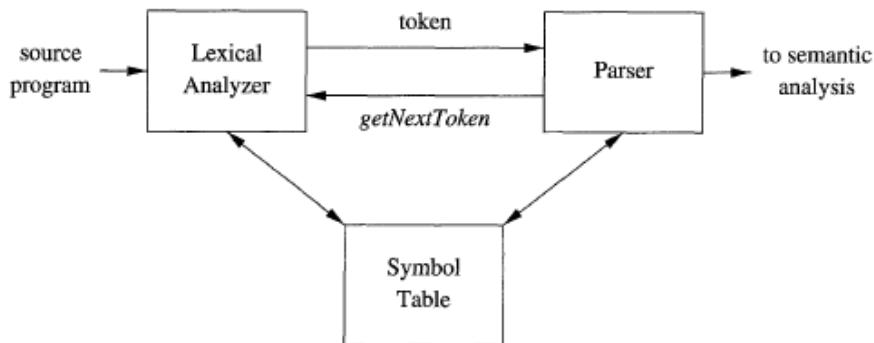
For example, **int i;** is a **declaration of i**, while **i = 1;** is a **definition of i**.

Lexical Analysis

The Role of the Lexical Analyzer

The main task of the lexical analyzer is to **read the input characters of the source program, group them into lexemes, and produce as output a sequence of tokens for each lexeme in the source program. The stream of tokens is sent to the parser** for syntax analysis.

1. It is common for the lexical analyzer to interact with the symbol table as well.
2. When the lexical analyzer **discovers a lexeme constituting an identifier**, it needs to **enter that lexeme into the symbol table**.



Interactions between the lexical analyzer and the parser

Other tasks can be performed by lexical analyser is:

1. **Removing out comments and whitespace** (blank, newline, tab, and perhaps other characters that are used to **separate tokens in the input**).
2. Another task is correlating error messages generated by the compiler with the source program.
3. The lexical analyzer may keep track of the number of newline characters seen, so it can associate **a line number with each error message**.
4. In some compilers, the lexical analyzer makes a copy of the source program with the error messages inserted at the appropriate positions.
5. **In some compilers, the lexical analyzer makes a copy of the source program with the error messages inserted** at the appropriate positions.
6. If the source program uses a **macro-preprocessor**, the **expansion of macros** may also be performed by the lexical analyzer.

Lexical Analysis versus Parsing

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

1. The separation of lexical and syntactic analysis often allows us to **simplify** at least one of these tasks.
2. **Compiler efficiency is improved**. A separate lexical analyzer allows us to apply specialized techniques that serve only the lexical task, not the job of parsing.
3. Compiler portability is enhanced. **Input-device-specific typicality** can be **restricted to the lexical analyzer**.

Tokens, Patterns, and Lexemes

- A **token** is a pair consisting of a token name and an optional attribute value. The token name is an abstract symbol representing a kind of lexical unit.
- A **pattern** is a **description of the form** that the lexemes of a token may take. In the case of a keyword as a token, the pattern is just the sequence of characters that form the keyword.
- A **lexeme** is a sequence of characters in the source program that matches the pattern for a token and is identified by the lexical analyzer as an instance of that token.

Tricky Problems When Recognizing Tokens

The following example is taken from **Fortran**, in the **fixed-format** still allowed in Fortran 90. In the statement

DO 5 I = 1.25

It is not clearly understood that the first lexeme is D051, an instance of the identifier token, until we see the dot following the 1.

Note that blanks in fixed-format Fortran are ignored.

Had we seen a comma instead of the dot, we would have had a **do-statement**

DO 5 I = 1,25

in which the first lexeme is the keyword DO.

Example from GateOverflow**Example**

result = x > y ? "x is greater than y" : "x is less than or equal to y";

Solution: there are 10 tokens.

Example

The number of tokens in following program?

```
# define M 100
int main ( )
{
// declare variables
int n = 2020;
return n % M;
```

Solution

Number of tokens = 16

Macros and comments are not considered as tokens.

Example:

```
printf("string %d ",++i++&&&i***a);
return(x?y:z)
```

Solution there are 24 tokens, *** are three tokens as ** is not a operator. &&, & two tokens.

Example

Consider the following program segment :

```
int foo(unsigned int n)
{
    int c, x = 0;
    While (n != 0)
    {
        if (n & 01) x++;
        n >>= 1;
    }
    return c;
}
```

The number of tokens generated by the Lexical Analyzer in the above program are _____.

Solution

>>=, !=, ++ will be considered as 1 – 1 token.

Total number of tokens = 40

Example

No of tokens in the following C statement are _____

```
/* abc */Printf("what's up %d", ++&&* *a);
```

Solution: 12

Examples Which one is lexical error?

1. ; a = b + c **No**
2. c = a + "Hello" **No**
3. int a = 1, b=2; **No**
4. String S ="Hello **Yes**
5. char a = 'ab **Yes**
6. char b = ab' **Yes**
7. String J = compiler" **Yes**
8. int a = 908; **No**
9. float c = 9.7; **No**
10. float d = 9...8; **Yes**
11. int1 a, b; **No**
12. int 1a, b; **Yes**
13. 1int a, b; **Yes**
14. int a, b; **No**

Syntax Analysis

The Role of the Parser

The parser obtains a string of tokens from the lexical analyzer, and verifies that the string of token names can be generated by the grammar for the source language.

The parser **constructs a parse tree and passes it to the rest of the compiler for further processing.**

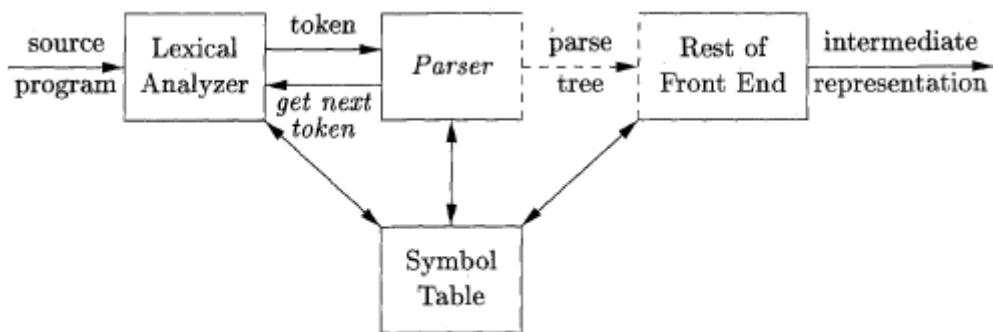


Figure 4.1: Position of parser in compiler model

There are three general types of parsers for grammars:

1. Universal
2. Top-down
3. Bottom-up.

1. **Universal parsing** methods such as the **CYK algorithm** and **Earley's algorithm** can parse any grammar. These general methods are, however, too "inefficient" to use in production compilers. Usually we have linear time algo but sometimes it takes polynomial time.
2. **Top-down** methods build parse trees from the **top (root) to the bottom (leaves)**.
3. **Bottom-up methods** start from the **leaves** and work their way up to the root.

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

1. Lack of tokens to frame a statement.
2. More no. of tokens than actually required for a statement.

Syntax Error Handling

A compiler is expected to assist the programmer in locating and **tracking down** errors. Common programming errors can occur at many different levels.

- **Lexical errors** include misspellings of identifiers, keywords, or operators - e.g., the use of an identifier elipsesize instead of ellipsesize – and missing quotes around text intended as a string.
- **Syntactic errors** include **misplaced semicolons or extra or missing braces**; that is, {" or "}" As another example, in C or Java, the appearance of a case statement without an enclosing switch is a syntactic error.

- **Semantic errors** include **type mismatches** between operators and operands. An example is a return statement in a Java method with result type void.
- **Logical errors** can be anything from **incorrect reasoning** on the part of the programmer to the use in a C program of the assignment operator `=` instead of the comparison operator `==`. The program containing `=` may be well formed; however, it may not reflect the programmer's intent. But logical error is not considered as compiler error.

Derivations

Beginning with the **start symbol**, **each rewriting step replaces a nonterminal by the body of one of its productions**. This derivational view corresponds to the **top-down construction of a parse tree**.

For example, consider the following grammar, with a single nonterminal E

$$E \rightarrow E + E \mid E * E \mid -E \mid (E) \mid \text{id}$$

We can take a single E and repeatedly apply productions in any order to get a sequence of replacements. For example

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(\text{id})$$

We call such a sequence of replacements a **derivation of $-(\text{id})$ from E** .

1. The symbol \Rightarrow means “**derives in one step**”.
2. The symbol $\stackrel{*}{\Rightarrow}$ means “**derives in zero or more steps**”.
3. The symbol $\stackrel{+}{\Rightarrow}$ means “**derives in one or more steps**”.

Sentential form of G

If $S \stackrel{*}{\Rightarrow} \alpha$ where S is the start symbol of a grammar G , we say that α is a **sentential form** of G . Note that a sentential form may contain both terminals and nonterminals, and may be empty.

Sentence of G

A **sentence** of G is a sentential form with no nonterminals. The language generated by a grammar is its set of sentences. Thus, a string of terminals w is in $L(G)$, the language generated by G , if and only if w is a sentence of G ($G \stackrel{*}{\Rightarrow} w$).

The string $-(\text{id} + \text{id})$ is a **sentence** of the grammar because there is a derivation

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E + E) \Rightarrow -(\text{id} + E) \Rightarrow -(\text{id} + \text{id})$$

The strings $E, -E, -(E), \dots, -(\text{id} + \text{id})$ are all sentential forms of this grammar. We write $E \stackrel{*}{\Rightarrow} -(\text{id} + \text{id})$ to indicate that $-(\text{id} + \text{id})$ can be derived from E .

Leftmost Derivations

In **leftmost derivations**, the leftmost nonterminal in each sentential is always chosen. If $\alpha \Rightarrow \beta$ is a step in which the leftmost nonterminal in α is replaced, we write $\stackrel{\alpha \Rightarrow \beta}{lm}$

$$E \stackrel{lm}{\Rightarrow} -E \stackrel{lm}{\Rightarrow} -(E) \stackrel{lm}{\Rightarrow} -(E + E) \stackrel{lm}{\Rightarrow} -(\text{id} + E) \stackrel{lm}{\Rightarrow} -(\text{id} + \text{id})$$

Example of leftmost derivations

Rightmost Derivations

In **rightmost derivations**, the **rightmost nonterminal** is always chosen; we write $\alpha \xrightarrow[\text{rm}]{\beta}$ in this case.

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E + E) \Rightarrow -(E + \text{id}) \Rightarrow -(\text{id} + \text{id})$$

Example of rightmost derivations

Note - Rightmost derivations are sometimes called canonical derivations.

Parse Trees and Derivations

A **parse tree** is a graphical representation of a derivation that filters out the order in which productions are applied to replace nonterminals.

The interior node is labeled with the **nonterminal A** in the head of the production; **the children of the node are labeled, from left to right, by the symbols in the body of the production by which this A was replaced during the derivation.**

The **leaves** of a parse tree are labeled by **nonterminals or terminals** and, read from **left to right**, constitute a **sentential form**, called the **yield** or **frontier** of the tree.

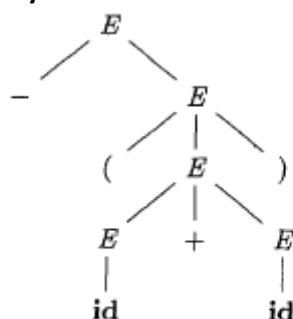


Figure 4.3: Parse tree for $-(\mathbf{id} + \mathbf{id})$

Note - Every parse tree has associated with it a unique leftmost and a unique rightmost derivation.

$$E \Rightarrow -E \Rightarrow -(E + E) \Rightarrow -(\text{id} + E) \Rightarrow -(\text{id} + \text{id}) \quad (4.8) \quad (\text{LMP})$$

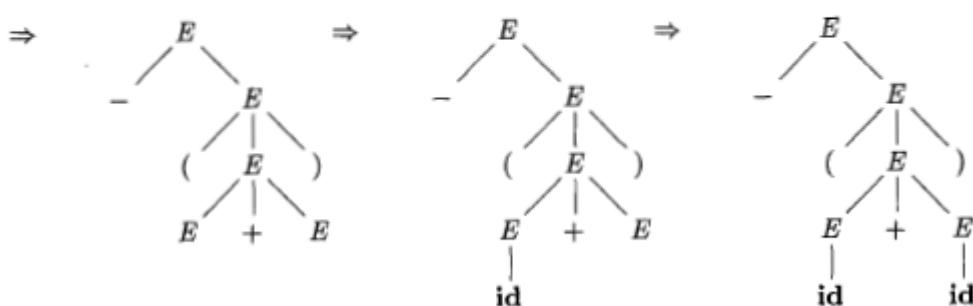
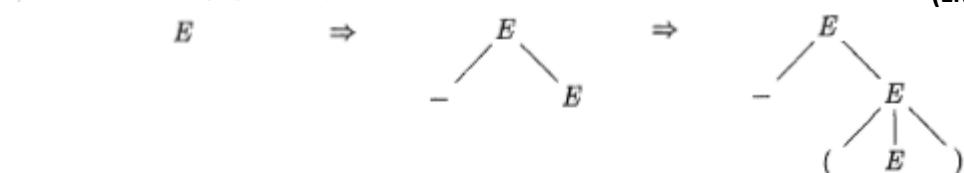


Figure 4.4: Sequence of parse trees for derivation (4.8)

Ambiguity

1. A grammar that produces **more than one parse tree** for **some sentence** is said to be **ambiguous**.
2. An ambiguous grammar is one that produces **more than one leftmost derivation** or **more than one rightmost derivation** for the same sentence.

The following arithmetic expression grammar permits **two distinct leftmost derivations** for the sentence **id + id * id**.

$$\begin{array}{ll}
 E \Rightarrow E + E & E \Rightarrow E * E \\
 \Rightarrow id + E & \Rightarrow E + E * E \\
 \Rightarrow id + E * E & \Rightarrow id + E * E \\
 \Rightarrow id + id * E & \Rightarrow id + id * E \\
 \Rightarrow id + id * id & \Rightarrow id + id * id
 \end{array}$$

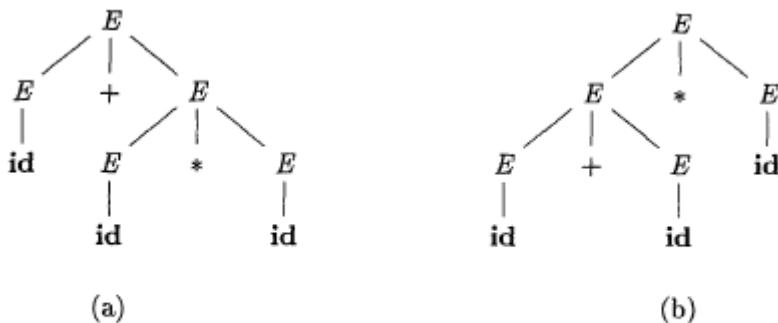


Figure 4.5: Two parse trees for **id+id*id**

Note that the parse tree of Fig. (a) reflects the **commonly assumed precedence of + and ***, while the tree of Fig. (b) **does not**.

Lexical Versus Syntactic Analysis

"**Why use regular expressions to define the lexical syntax of a language?**"

There are several reasons.

1. The **lexical rules of a language are frequently quite simple**, and to describe them we do not need a notation as powerful as grammars.
2. Regular expressions generally provide a more **concise** and **easier-to-understand** notation for tokens than grammars.
3. More efficient lexical analyzers can be **constructed automatically from regular expressions** than from arbitrary grammars.

Grammars, on the other hand, are most useful for describing **nested structures** such as **balanced parentheses**, matching **begin-end's**, corresponding if-then-else's, and so on. **These nested structures cannot be described by regular expressions.**

Eliminating Ambiguity

"Sometimes" an ambiguous grammar can be rewritten to eliminate the ambiguity

Dangling Else Problem

We shall eliminate the ambiguity from the following "dangling else" grammar:

$$\begin{array}{lcl} stmt & \rightarrow & \text{if } expr \text{ then } stmt \\ & | & \text{if } expr \text{ then } stmt \text{ else } stmt \\ & | & \text{other} \end{array}$$

Here "other" stands for any other statement. According to this grammar, the compound conditional statement

if E_1 then S_1 else if E_2 then S_2 else S_3

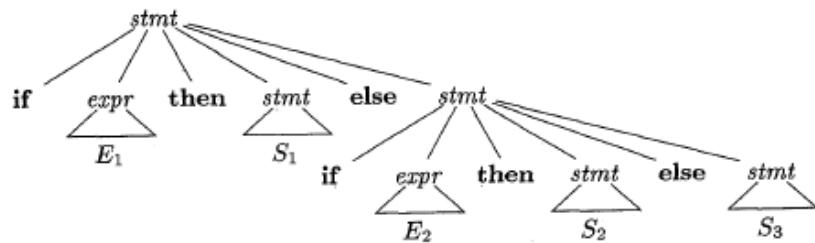


Figure 4.8: Parse tree for a conditional statement

has the above parse tree. But this grammar is ambiguous since the string

if E_1 then if E_2 then S_1 else S_2

has the two parse trees as shown

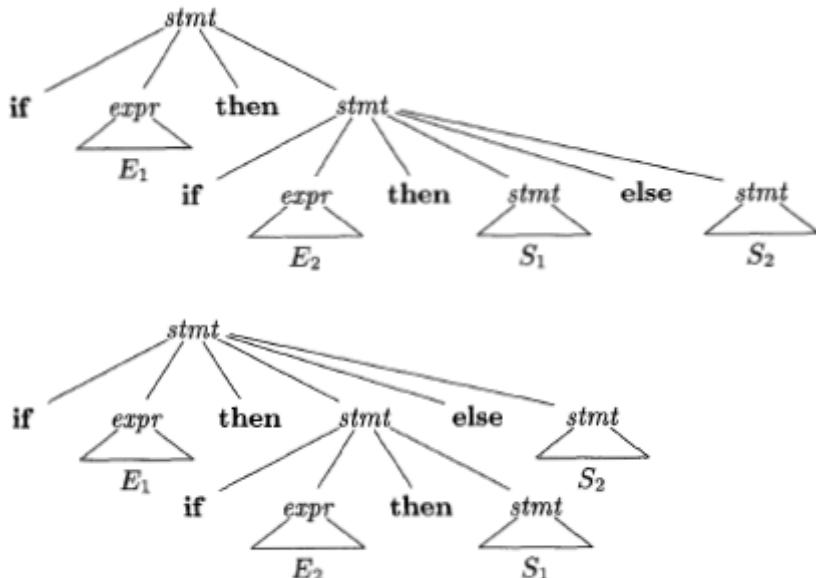


Figure 4.9: Two parse trees for an ambiguous sentence

```

stmt   →  matched_stmt
      |  open_stmt
matched_stmt → if expr then matched_stmt else matched_stmt
      |  other
open_stmt   → if expr then stmt
      |  if expr then matched_stmt else open_stmt
  
```

Figure 4.10: Unambiguous grammar for if-then-else statements

Elimination of Left Recursion

Where **the leftmost symbol of the body** is the same as the **nonterminal at the head** of the production.

Head of production → body of production

Suppose the procedure for expr decides to apply this production.

1. The body begins with expr so the procedure for expr is called recursively.
2. Since the look-ahead symbol changes only when a terminal in the body is matched.
3. No change to the input took place between recursive calls of expr.
4. As a result, the second call to expr does exactly what the first call did, which means a third call to expr, and so on, **forever**.

A left-recursive production can be eliminated by rewriting the offending production.

$A \rightarrow A\alpha \mid \beta$ (Immediate left recursion)

It is possible for a recursive-descent parser to loop forever. A problem arises with "left-recursive" productions like

$expr \rightarrow expr + term$

A grammar is **left recursive** if it has a **nonterminal A** such that there is a derivation $A \stackrel{+}{\Rightarrow} A\alpha$ for some string α . (Indirect left Recursion)

Removal A left recursive grammar is given as

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

We can rewrite it as

$$\begin{aligned} A &\rightarrow \beta A' \\ A' &\rightarrow \alpha A' \mid \epsilon \end{aligned}$$

Example remove the left recursion from the following grammar

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

Solution

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

Example remove the left recursion from the following grammar

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

Solution

$$\begin{array}{l} E \rightarrow T E' \\ E' \rightarrow +T E' \mid \text{eps} \\ T \rightarrow F T' \\ T' \rightarrow *F T' \mid \text{eps} \\ F \rightarrow (E) \mid \text{id} \end{array}$$

is obtained by **eliminating immediate left recursion** from the expression grammar.

Removal of Immediate left Recursion

Immediate left recursion can be **eliminated** by the following technique, which works for any number of A-productions. First, **group the productions as**

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \dots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$$

where no β_i begins with an A . Then, replace the A-productions by

$$\begin{array}{l} A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \dots \mid \beta_n A' \\ A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \dots \mid \alpha_m A' \mid \epsilon \end{array}$$

The nonterminal A generates the same strings as before but is **no longer left recursive**.

Note - It does not eliminate left recursion involving derivations of “two or more steps”.

For example, consider the grammar

$$\begin{array}{l} S \rightarrow A a \mid b \\ A \rightarrow A c \mid S d \mid \epsilon \end{array}$$

The **nonterminal S is left recursive** because $S \Rightarrow Aa \Rightarrow Sda$ but it is **not immediately left recursive**.

We have an algorithm that systematically eliminates left recursion from a grammar. **It is guaranteed to work** if the **grammar has no cycles** (derivations of the form $A \stackrel{*}{\Rightarrow} A$) or ϵ -**productions** (productions of the form $A \rightarrow \epsilon$).

Note – Cycles can be eliminated systematically as can ϵ -productions. (I assume here that they are talking about to convert a CFG into CNF or GNF).

Example Remove the left recursion from the following grammar

$$S \rightarrow Sa \mid S \mid \text{eps}$$

Solution

$$\begin{array}{l} S \rightarrow S' \\ S' \rightarrow aS' \mid S' \mid \text{eps} \end{array}$$

But we can remove $S' \rightarrow S'$, it's a trivial production, hence equivalent grammar will be

$$S \rightarrow S'$$

$$S' \rightarrow aS' \mid \text{eps}$$

Algorithm 4.19: Eliminating left recursion.

INPUT: Grammar G with no cycles or ϵ -productions.

OUTPUT: An equivalent grammar with no left recursion.

METHOD: Apply the algorithm in Fig. 4.11 to G . Note that the resulting non-left-recursive grammar may have ϵ -productions. \square

- 1) arrange the nonterminals in some order A_1, A_2, \dots, A_n .
- 2) **for** (each i from 1 to n) {
- 3) **for** (each j from 1 to $i - 1$) {
- 4) replace each production of the form $A_i \rightarrow A_j \gamma$ by the productions $A_i \rightarrow \delta_1 \gamma \mid \delta_2 \gamma \mid \dots \mid \delta_k \gamma$, where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \dots \mid \delta_k$ are all current A_j -productions
- 5) }
- 6) eliminate the immediate left recursion among the A_i -productions
- 7) }

Figure 4.11: Algorithm to eliminate left recursion from a grammar

Example Remove the indirect left recursion from the following grammar

$$\begin{aligned} S &\rightarrow A \ a \mid b \\ A &\rightarrow A \ c \mid S \ d \mid \epsilon \end{aligned}$$

Solution

We order the non-terminals as S, A .

We substitute for S in A $\rightarrow S \ d$ to obtain the following A-productions

$$A \rightarrow A \ c \mid A \ a \ d \mid b \ d \mid \epsilon$$

Now we can remove direct left recursion

$$\begin{aligned} S &\rightarrow A \ a \mid b \\ A &\rightarrow b \ d \ A' \mid A' \\ A' &\rightarrow c \ A' \mid a \ d \ A' \mid \epsilon \end{aligned}$$

Left Factoring

Left factoring is a grammar transformation that is useful for producing a grammar suitable for **predictive**, or **top-down**, **parsing**.

When the **choice between two alternative A-productions is not clear**, we may be able to rewrite the productions to defer the decision until enough of the input has been seen that we can make the right choice.

For example, if we have the two productions

$$\begin{array}{lcl} \text{stmt} & \rightarrow & \text{if } \text{expr} \text{ then stmt else stmt} \\ & | & \text{if } \text{expr} \text{ then stmt} \end{array}$$

Algorithm: Left factoring a grammar

INPUT: Grammar G.

OUTPUT: An equivalent **left-factored grammar**.

METHOD: For each nonterminal A , find the longest prefix α common to two or more of its alternatives. If $\alpha \neq \epsilon$ — i.e., there is a nontrivial common prefix — replace all of the A -productions $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \dots \mid \alpha\beta_n \mid \gamma$, where γ represents all alternatives that do not begin with α , by

$$\begin{array}{l} A \rightarrow \alpha A' \mid \gamma \\ A' \rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_n \end{array}$$

Here A' is a new nonterminal. **Repeatedly apply this transformation** until no two alternatives for a nonterminal have a common prefix.

Example The following grammar abstracts the "dangling-else" problem:

$$\begin{array}{l} S \rightarrow i \text{ } E \text{ } t \text{ } S \mid i \text{ } E \text{ } t \text{ } S \text{ } e \text{ } S \mid a \\ E \rightarrow b \end{array}$$

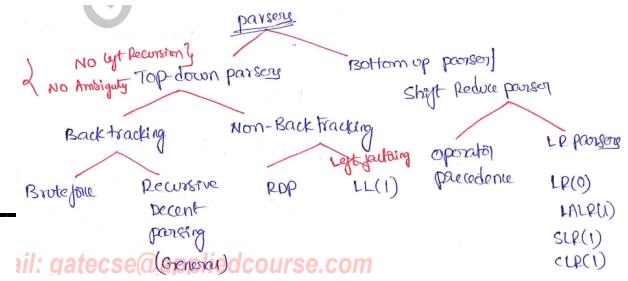
Solution

Left factored grammar

$$\begin{array}{l} S \rightarrow i \text{ } E \text{ } t \text{ } S \text{ } S' \mid a \\ S' \rightarrow e \text{ } S \mid \epsilon \\ E \rightarrow b \end{array}$$

which of the following sentences are verified by Semantic Analyzer ?

- 1) Has the variable been declared ?
- 2) What is the type and size of variable ?
- 3) Instead of semi-colon if comma are used in for loop structure. (false syntax analyzer)
- 4) If a function is not declared, but it is defined. (false linker)
- 5) If x is a function, how many arguments does it take?
- 6) Does the name "x" correspond to a variable or a function.



Top-Down Parsing

mail: gatecse@gnudcourse.com

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

Top-down parsing can be viewed as finding a “leftmost derivation” for an input string

The sequence of parse trees in given below for the input **id+id*id** is a top-down parse according to the following grammar:

$$\begin{aligned}
 E &\rightarrow T E' \\
 E' &\rightarrow + T E' \mid \epsilon \\
 T &\rightarrow F T' \\
 T' &\rightarrow * F T' \mid \epsilon \\
 F &\rightarrow (E) \mid \text{id}
 \end{aligned}$$

Solution

This sequence of trees corresponds to a leftmost derivation of the input.

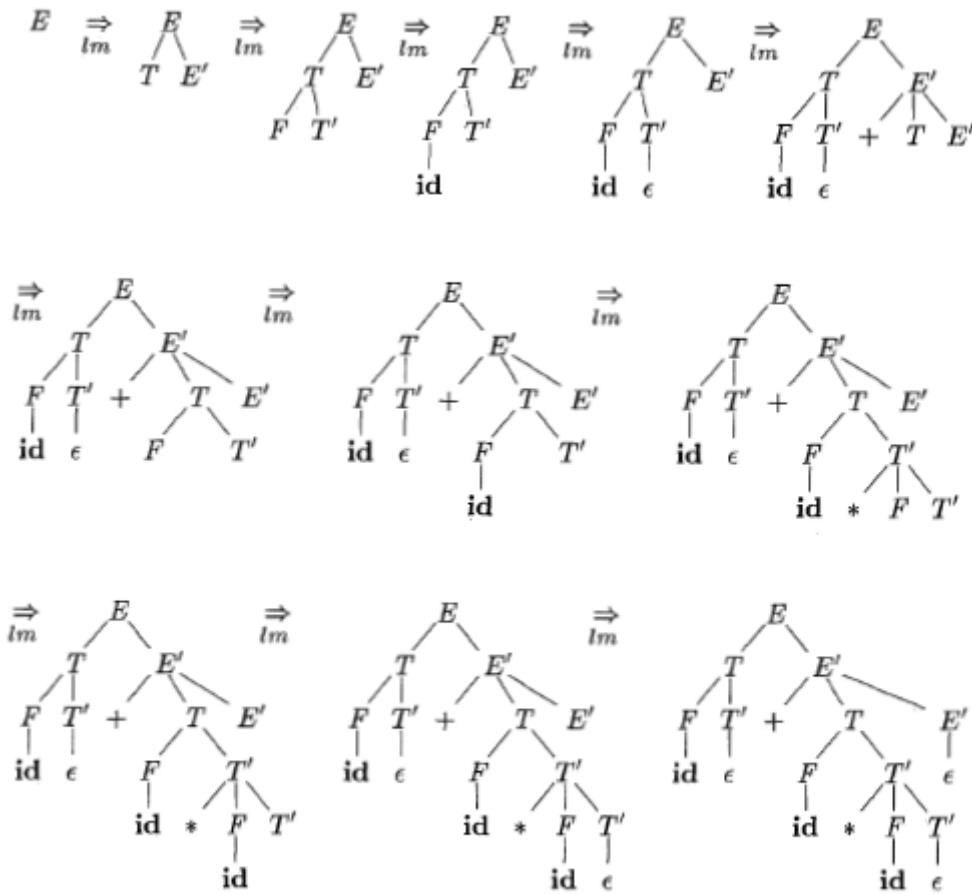


Figure 4.12: Top-down parse for **id + id * id**

At each step of a top-down parse, **the key problem is that of determining the production to be applied for a nonterminal**, say A. Once an A-production is chosen, **the rest of the parsing process consists of “matching” the terminal symbols in the production body with the input string**.

The **Top-Down** construction of a parse tree is done by starting with the root, labeled with the starting nonterminal, and repeatedly performing the following two steps.

1. At node N, labeled with nonterminal A, select one of the productions for A and construct children at N for the symbols in the production body.
2. Find the **next node** at which a subtree is to be constructed, typically the **leftmost unexpanded** nonterminal of the tree.

In general, the selection of a production for a nonterminal may involve **trial-and-error**; that is, we may have to try a production and backtrack to try another production if the first is found to be unsuitable.

A production is **unsuitable** if, after using the production, we cannot complete the tree to match the input string.

Note – “Backtracking is not needed, however, in an important special case called predictive parsing”.

Predictive Parsing

Recursive-descent parsing is a top-down method of syntax analysis in which a set of recursive procedures is used to process the input.

One procedure is associated with each nonterminal of a grammar.

Predictive parsing is a simple form of recursive-descent parsing

In which the **lookahead symbol unambiguously determines** the flow of control through the procedure body for **each nonterminal**.

The class of grammars for which we can construct predictive parsers looking k symbols ahead in the input is sometimes called the **LL(k) class**.

Recursive-Descent Parsing = O(n) time complexity where n is length of string

```

void A() {
1)      Choose an A-production,  $A \rightarrow X_1 X_2 \cdots X_k$ ;
2)      for (  $i = 1$  to  $k$  ) {
3)          if (  $X_i$  is a nonterminal )
4)              call procedure  $X_i()$ ;
5)          else if (  $X_i$  equals the current input symbol  $a$  )
6)              advance the input to the next symbol;
7)          else /* an error has occurred */;
}
}

```

Figure 4.13: A typical procedure for a nonterminal in a top-down parser

A recursive-descent parsing program consists of a **set of procedures, one for each nonterminal**. Execution begins with the procedure for the start symbol, which halts and announces success if its procedure body **scans the entire input string**. **Recursive descent parsing is not backtracking parser**.

Example Consider the grammar

$$\begin{array}{l} S \rightarrow c A d \\ A \rightarrow a b \mid a \end{array}$$

To construct a parse tree top-down for the input string $w = \text{cad}$.

Solution

1. S has only one production, so we use it to expand S and obtain the tree.
2. The leftmost leaf, labeled c , matches the first symbol of input w , so we advance the input pointer to a , the second symbol of w , and consider the next leaf, labeled A .
3. Now, we expand A using the first alternative $A \rightarrow ab$ to obtain the tree, we have a match for second input symbol a , so we advance the input pointer to d .
4. The third input symbol, and compare d against the next leaf, labeled b . Since b does not match d , we **report failure** and go **back to A** to see whether there is another alternative for A that has not been tried, but that might produce a match.
5. In going back to A , we must reset the input pointer to position 2, the position it had when we first came to A .
6. The second alternative for A produces the tree, and the procedure halts.

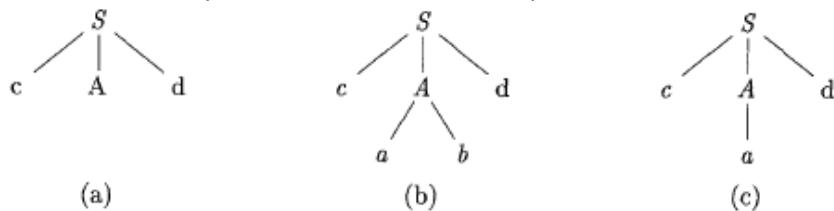


Figure 4.14: Steps in a top-down parse

Note - A **left-recursive grammar** can cause a **recursive-descent parser**, even one with backtracking, to go into an **infinite loop**. Here *even one with backtracking* sentence talks about special case of recursive descent parser but generally **it does not allow backtracking**.

FIRST and FOLLOW

The construction of both **top-down and bottom-up parsers** is aided by two functions, **FIRST** and **FOLLOW**, associated with a grammar G .

1. During **top-down parsing**, **FIRST** and **FOLLOW** allow us to choose which production to apply, based on the next input symbol.
2. During **panic-mode error recovery**, sets of tokens produced by **FOLLOW** can be used as synchronizing tokens.

First(X)

To compute $\text{FIRST}(X)$ for all grammar symbols X , apply the following rules until no more terminals or ϵ can be added to any **FIRST** set.

1. $A \rightarrow a\alpha \quad \text{First}(A) = \{a\}$ where a is terminal.
2. $A \rightarrow B\alpha \quad \text{First}(A) = \text{First}(B)$, where A , and B are non-terminals, and B doesn't ϵ .
3. $A \rightarrow B\alpha \quad \{\text{First}(B) - \epsilon\} \cup \{\text{First}(\alpha)\}$ where B, α are non-terminals where B contains ϵ .

Example

i. $S \rightarrow aAB$

$A \rightarrow b$

$B \rightarrow c$

First(S) = {a}

First(A) = {b}

First(B) = {c}

ii. $S \rightarrow AB$

$A \rightarrow a \mid \epsilon$

$B \rightarrow b$

First(S) = {a} \cup {b} = {a, b}

First(A) = {a, ϵ }

First(B) = {b}

iii. $S \rightarrow AB$

$A \rightarrow a$

$B \rightarrow b \mid \epsilon$

First(S) = {a}

First(A) = {a}

First(B) = {b, ϵ }

Follow Set

To compute FOLLOW(A) for all nonterminals A, apply the following rules until nothing can be added to any FOLLOW set.

1. Place $\$$ in FOLLOW(S) where S is the start symbol, and $\$$ is the input right end marker.
2. If there is a production $A \rightarrow \alpha B \beta$ then everything in $\text{FIRST}(\beta)$ except ϵ is in FOLLOW(B).
3. If there is a production $A \rightarrow \alpha B$ or a production $A \rightarrow \alpha B \beta$, where $\text{FIRST}(\beta)$ contains ϵ then everything in FOLLOW(A) is in FOLLOW(B).

Note : To find follow of any variable first we look at whether it appear on righthand side if it is then we find follow. If in case we cannot find follow of variable we look same variable at some other production. In 2nd example given below $\$$ is not included in T because we have tried to look into second production but we are not able to find follow so we proceed to find same variable on some other production which is $S \rightarrow T M E \mid T \text{-id}$

Example Find the First() and Follow() set

$$\begin{aligned}
 G &= \{S \rightarrow A0 \mid S1 \mid \epsilon, A \rightarrow S2 \mid B1 \mid \epsilon, B \rightarrow 0B \mid A1 \mid BA\} \\
 \text{First}(S) &= \{0, 1, 2, \epsilon\} \\
 \text{First}(A) &= \{0, 1, 2, \epsilon\} \\
 \text{First}(B) &= \{0, 1, 2\} \\
 \text{Follow}(S) &= \{1, 2, \$\} \\
 \text{Follow}(A) &= \{0, 1, 2\} \\
 \text{Follow}(B) &= \{0, 1, 2\}
 \end{aligned}$$

Example Find the Follow() set

$$\begin{aligned}
 S &\rightarrow M \text{id} \mid T M E \mid -E \mid T - \text{id} & \text{Follow}(S) &= \{\$\} \\
 T &\rightarrow E \times T \mid E & \text{Follow}(T) &= \{+, -\} \\
 E &\rightarrow (\text{id}) & \text{Follow}(E) &= \{\$, \times, +, -\} \\
 M &\rightarrow + \mid - & \text{Follow}(M) &= \{\text{id}, ()\}
 \end{aligned}$$

LL(1) Grammars

Predictive parsers, that is, recursive-descent parsers needing no backtracking, can be constructed for a class of grammars called LL(1). The first "L" in LL(1) stands for scanning the input from left to right, the second "L" for producing a leftmost derivation, and the "1" for using one input symbol of lookahead at each step to make parsing action decisions. **LL(0) parsers** do look at the tokens, but they don't decide which productions to apply upon them. They just determine if the sequence belongs to the language or not. This means that every non-terminal symbol must have a single right-hand side and that there may be no recursion.

1. The class of LL(1) grammars is rich enough to cover most programming constructs.
2. No left-recursive or ambiguous grammar can be LL(1).

A grammar G is LL(1) if and only if whenever $A \rightarrow \alpha \mid \beta$ are two distinct productions of G, the following conditions hold:

1. $\text{First}(\alpha) \cap \text{First}(\beta) = \emptyset$, Where, For **no terminal 'a'**, **do both α and β derive strings** beginning with a .
2. **At most α and β** can derive the empty string.
3. If $\beta \xrightarrow{*} \epsilon$ then $\text{First}(\alpha) \cap \text{Follow}(\beta) = \emptyset$, and if $\alpha \xrightarrow{*} \epsilon$ then $\text{First}(\beta) \cap \text{Follow}(\alpha) = \emptyset$.

Construction of a predictive parsing table

INPUT: Grammar G

OUTPUT: Parsing table M

METHOD: For each production $A \rightarrow \alpha$ of the grammar, do the following:

1. For each terminal a in $\text{FIRST}(A)$, add $A \rightarrow \alpha$ to $M[A, a]$.
2. If ϵ is in $\text{FIRST}(\alpha)$, then for each terminal b in $\text{FOLLOW}(A)$, add $A \rightarrow \alpha$ to $M[A, b]$. If ϵ is in $\text{FIRST}(\alpha)$ and $\$$ is in $\text{FOLLOW}(A)$, add $A \rightarrow \alpha$ to $M[A, \$]$ as well.

If, after performing the above, there is no production at all in $M[A, a]$, then set $M[A, a]$ to **error** (which we normally represent by an empty entry in the table). \square

Example construct LL(1) parsing table for the following grammar

$$\begin{array}{l}
 E \rightarrow T E' \\
 E' \rightarrow + T E' \mid \epsilon \\
 T \rightarrow F T' \\
 T' \rightarrow * F T' \mid \epsilon \\
 F \rightarrow (E) \mid \text{id}
 \end{array}$$

- First(E) = First(T) = First(F) = { (, id}
- First(E') = { +, ϵ }
- First(T') = { *, ϵ }

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

	id	+	*	()	\$
E	E->TE'			E->TE'		
E'		E'->+TE'			E'->eps	E'->eps
T	T->FT'			T->FT'		
T'		T'->eps	T'->*FT'		T'->eps	T'->eps
F	F->id			F->epsilon		

1. For **every LL(1) grammar**, each parsing-table entry **uniquely identifies a production** or **signals an error**.
2. If G is **left-recursive** or **ambiguous**, then M will have at least one **multiple** entries. Although **left recursion elimination** and **left factoring** are easy to do, but

There are some grammars for which “no amount of alteration” will produce an LL(1) grammar.

If grammar is left recursive and left factored then it is not LL(1)

The language in the following example has **no LL(1) grammar** at all.

Example 4.33: The following grammar, which abstracts the dangling-else problem, is repeated here from Example 4.22:

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

Solution

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

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

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

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

First(S') \cap Follow(S') $\neq \emptyset$, hence this grammar is **not LL(1)**.

NON - TERMINAL	INPUT SYMBOL					
	a	b	e	i	t	\$
S	$S \rightarrow a$			$S \rightarrow iEtSS'$		
S'			$S' \rightarrow \epsilon$ $S' \rightarrow eS$			$S' \rightarrow \epsilon$
E		$E \rightarrow b$				

Parsing Table M

This grammar is ambiguous!!

NOTE : For top down parsers, the grammar should be unambiguous, deterministic and should not be left recursive. All the 3 conditions must be satisfied for LL(1) parsers.

Nonrecursive Predictive Parsing

A **nonrecursive predictive parser** can be built by maintaining a **stack explicitly, rather than implicitly via recursive calls**. The parser mimics a **leftmost derivation**. If w is the input that has been matched so far,

then the stack holds a sequence of grammar symbols α such that

$$S \xrightarrow[lm]{*} wa$$

1. The table-driven parser in has an **input buffer**, a **stack** containing a **sequence of grammar symbols**, a **LL(1) parsing table** and an **output stream**.
2. The input buffer contains the string to be parsed, followed by the end marker $\$$.
3. We reuse the symbol $\$$ to mark the **bottom of the stack**, which **initially contains the start symbol of the grammar on top of $\$$** .

Suppose, **X is the symbol on top of the stack**, and **a, the current input symbol**.

1. If X is a **nonterminal**, the parser chooses an X-production by consulting entry **M[X, a]** of the parsing table M.

Algorithm Table-driven predictive parsing.

INPUT: A string w and a parsing table M for grammar G .

OUTPUT: If w is in $L(G)$, a **leftmost derivation of w** ; otherwise, an error indication

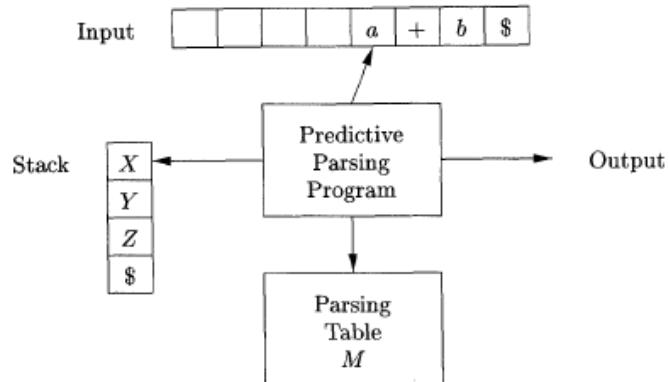


Figure 4.19: Model of a table-driven predictive parser

Method:

Initially, the parser is in a configuration with **w\$ in the input buffer** and the start symbol **S of G on top of the stack, above \$**.

- Set i/p to point to the first symbol of w .
 - Set X to the top stack symbol.
1. If $X = a = \$$ input string is valid and accepted.
 2. If ($X=a!=\$$) X is a terminal on the top of the stack and match with input symbol, POP X from stack and advance the i/p.
 3. If (X is a non-terminal) parser takes decision from parsing table
If $M[X, a] = uvw$ then PUSH into the stack such a way that **u is on the top**.

EXAMPLE

	id	+	*	()	\$
E	E->TE'			E->TE'		
E'		E'->+TE'			E'->eps	E'->eps
T	T->FT'			T->FT'		
T'		T'->eps	T'->*FT'		T'->eps	T'->eps
F	F->id			F->ε		

input:	id	+	id	*	id	\$
--------	----	---	----	---	----	----

MATCHED	STACK	INPUT	ACTION
E\$	id + id * id\$		
TE'\$	id + id * id\$		output $E \rightarrow TE'$
FT'E'\$	id + id * id\$		output $T \rightarrow FT'$
id T'E'\$	id + id * id\$		output $F \rightarrow id$
id	T'E'\$	+ id * id\$	match id
id	E'\$	+ id * id\$	output $T' \rightarrow \epsilon$
id	+ TE'\$	+ id * id\$	output $E' \rightarrow + TE'$
id +	TE'\$	id * id\$	match +
id +	FT'E'\$	id * id\$	output $T \rightarrow FT'$
id +	id T'E'\$	id * id\$	output $F \rightarrow id$
id + id	T'E'\$	* id\$	match id
id + id	* FT'E'\$	* id\$	output $T' \rightarrow * FT'$
id + id *	FT'E'\$	id\$	match *
id + id *	id T'E'\$	id\$	output $F \rightarrow id$
id + id * id	T'E'\$	\$	match id
id + id * id	E'\$	\$	output $T' \rightarrow \epsilon$
id + id * id	\$	\$	output $E' \rightarrow \epsilon$

Figure 4.21: Moves made by a predictive parser on input id + id * id

Moves made by a predictive parser on input id + id * id

The nonrecursive predictive parser makes the sequence of moves, these moves correspond to a leftmost derivation

$$E \xrightarrow{lm} TE' \xrightarrow{lm} FT'E' \xrightarrow{lm} id T'E' \xrightarrow{lm} id E' \xrightarrow{lm} id + TE' \xrightarrow{lm} \dots$$

(Notice the content of stack from above to down, this is same as LMD)

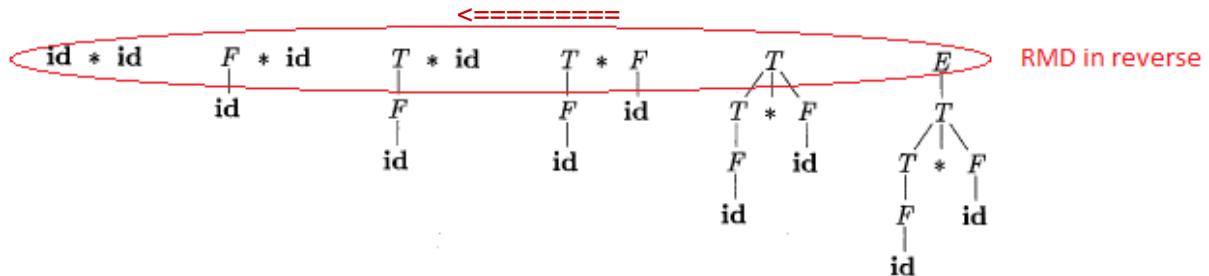
Note that the sentential forms in this derivation correspond to the input that has already been matched (in column MATCHED) followed by the stack contents.

Note : Size of parsing table of LL(k) = $V_n \times (1+V_t+V_t^2+\dots+V_t^k)$

Where V_n = non-terminal symbol, V_t = terminal symbol

Bottom up Parsing

A bottom-up parser corresponds to the construction of a parse tree for an input string **beginning at the leaves (the bottom) and working up towards the root (the top)**.

A bottom-up parse for id * id

A bottom-up parse of the token stream id * id, with respect to the following expression grammar

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

The goal of bottom-up parsing is therefore to construct a derivation in reverse.

RMD

$$E \Rightarrow T \Rightarrow T * F \Rightarrow T * id \Rightarrow F * id \Rightarrow id * id$$

Reductions

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

The sequence starts with the input string **id*id**. The **first reduction** produces **F * id** by reducing the leftmost **id to F**, using the production $F \rightarrow id$. Second reduction produces **T * id** by reducing **F to T**.

Now, **we have a choice between reducing the string T**, which is the body of $E \rightarrow T$, and the string consisting of the second id, which is the body of $F \rightarrow id$. Rather than **reduce T to E**, the second id is reduced to T, resulting in the string **T * F**. This string then reduces to T. The parse completes with the reduction of T to the start symbol E.

Note- A reduction is the reverse of a step in a derivation.

Handle

Bottom-up parsing during a **left-to-right scan** of the input constructs a **rightmost derivation in reverse**. Informally, a "handle" is a substring that matches the body of a production, and whose reduction represents one step along the reverse of a rightmost derivation.

RIGHT SENTENTIAL FORM	HANDLE	REDUCING PRODUCTION
$id_1 * id_2$	id_1	$F \rightarrow id$
$F * id_2$	F	$T \rightarrow F$
$T * id_2$	id_2	$F \rightarrow id$
$T * F$	$T * F$	$E \rightarrow T * F$

Figure 4.26: Handles during a parse of $id_1 * id_2$

1. Notice that the **string w to the right of the handle must contain only terminal symbols**.
2. If the grammar is **unambiguous** then **every right-sentential form of the grammar has exactly one handle**.
3. A **rightmost derivation in reverse** can be obtained by "handle" reductions.

Shift-Reduce Parsing

Shift-reduce parsing is a form of **bottom-up** parsing in which

1. A **stack** holds **grammar symbols**, and
2. An **input buffer** holds the **rest of the string to be parsed**.
3. We use **\$** to mark the **bottom of the stack** and also the **right end** of the input.

Note - The handle always appears at the top of the stack just before it is identified as the handle, never inside.

1. During a **left-to-right scan** of the input string, the **parser shifts zero or more input symbols** onto the **stack**, until it is **ready to reduce** a string β of **grammar symbols** on top of the stack.
2. It then reduces β to the head of the appropriate production.
3. The parser **repeats this cycle** until it has detected an error or until the stack contains the **start symbol** and the **input is empty**.

Following are the steps through the actions a shift-reduce parser might take in parsing the input string **id*id**.

STACK	INPUT	ACTION
\$	id₁ * id₂ \$	shift
\$ id ₁	* id ₂ \$	reduce by $F \rightarrow id$
\$ F	* id ₂ \$	reduce by $T \rightarrow F$
\$ T	* id ₂ \$	shift
\$ T *	id ₂ \$	shift
\$ T * id ₂	\$	reduce by $F \rightarrow id$
\$ T * F	\$	reduce by $T \rightarrow T * F$
\$ T	\$	reduce by $E \rightarrow T$
\$ E	\$	accept

3: Configurations of a shift-reduce parser on input id₁*id₂

While the primary operations are shift and reduce, there are actually **four possible actions** a shift-reduce parser can make:

1. **Shift**, Shift the next input symbol onto the top of the stack.
2. **Reduce**, the **right end of the string** to be **reduced must be at the top of the stack**. Locate the left end of the string within the stack and decide with what nonterminal to replace the string.
3. **Accept**, announce successful completion of parsing.
4. **Error**, discover a syntax error and call an error recovery routine.

Conflicts during Shift-Reduce Parsing

There are context-free grammars for which shift-reduce parsing cannot be used. Parser cannot decide whether to shift or to reduce (a **shift/reduce** conflict), or cannot decide which of several reductions to make (a **reduce/reduce** conflict).

These grammars are not in the LR(k) class of grammars.

Introduction to LR Parsing: Simple LR

LR(K) Parsing:

The "L" is for left-to-right scanning of the input, the "R" for constructing a rightmost derivation in reverse, and the k for the number of input symbols of lookahead that are used in making parsing decisions.

Note- LR(K), when K is not given we assume it to be 1.

Why LR Parsers?

1. Why LR Parsers?
2. LR parsers can be constructed to recognize virtually all programming language constructs for which context-free grammars can be written.
3. Non- LR context-free grammars exist.
4. An LR parser can detect a syntactic error as soon as it is possible to do so on a left-to-right scan of the input.
5. The class of grammars that can be parsed using LR methods is a proper superset of the class of grammars that can be parsed with predictive or LL methods.
6. For a grammar to be **LR(k)**, we must be able to recognize the occurrence of the right side of a production in a right-sentential form, with k input symbols of lookahead.
7. We have many LR parser generators available such a generator takes a context-free grammar and automatically produces a parser for that grammar.
8. If G is an $LR(k)$ grammar, then $L(G)$ is a DCFL.

LR(0) Parser

How does a shift-reduce parser know when to shift and when to reduce? An LR parser makes **shift-reduce decisions** **by maintaining states** to keep track of where we are in a parse.

1. States represent sets of "LR(0) items."
2. An **LR(0) item** (item for short) of a grammar G is a production of G with a dot at some position of the body.

Thus, production $A \rightarrow XYZ$ yields the four LR(0) items or items:

$$\begin{aligned} A &\rightarrow \cdot XYZ \\ A &\rightarrow X \cdot YZ \\ A &\rightarrow XY \cdot Z \\ A &\rightarrow XYZ \cdot \end{aligned}$$

Note The production $A \rightarrow \epsilon$ generates only one item, $A \rightarrow \cdot$

Intuitively, an item indicates how much of a production we have seen at a given point in the parsing process.

1. The item $A \rightarrow \cdot XYZ$ indicates that we hope to see a string derivable from XYZ next on the input.
2. $A \rightarrow X \cdot YZ$ indicates that we have just seen on the input a string derivable from X and that we hope next to see a string derivable from YZ .
3. Item $A \rightarrow XYZ \cdot$ indicates that we have seen the body XYZ and that it may be time to reduce XYZ to A .

1. Collection of sets of LR(0) items, called the **canonical LR(0) collection**.
2. A **deterministic finite automaton** is used to make parsing decisions.
3. Such an automaton is called an **LR(0) automaton**. Each state of the LR(0) automaton represents a **set of items** in the canonical LR(0) collection.

To construct the canonical **LR(0)** collection for a grammar,

1. We define an **augmented grammar** and two functions, **CLOSURE** and **GOTO**.
2. If **G** is a grammar with start symbol **S**, then **G'**, the **augmented grammar** for **G**, is **G** with a new start symbol **S'** and production **S' → S**.
3. Input acceptance occurs when and only when the parser is about to reduce by **St → S**.

Closure of Item Sets

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

1. Initially, add every item in **I** to **CLOSURE(I)**.
2. If **A → α·Bβ** is in **CLOSURE(I)** and **B → γ** is a production, then add the item **B → ·γ** to **CLOSURE(I)** if, it is not already there. Apply this rule until no more new items can be added to **CLOSURE(I)**.

Example Construct **LR(0)** automata for the following augmented grammar

$$\begin{array}{lcl} E' & \rightarrow & E \\ E & \rightarrow & E + T \mid T \\ T & \rightarrow & T * F \mid F \\ E & \rightarrow & (E) \mid \text{id} \end{array}$$

Solution

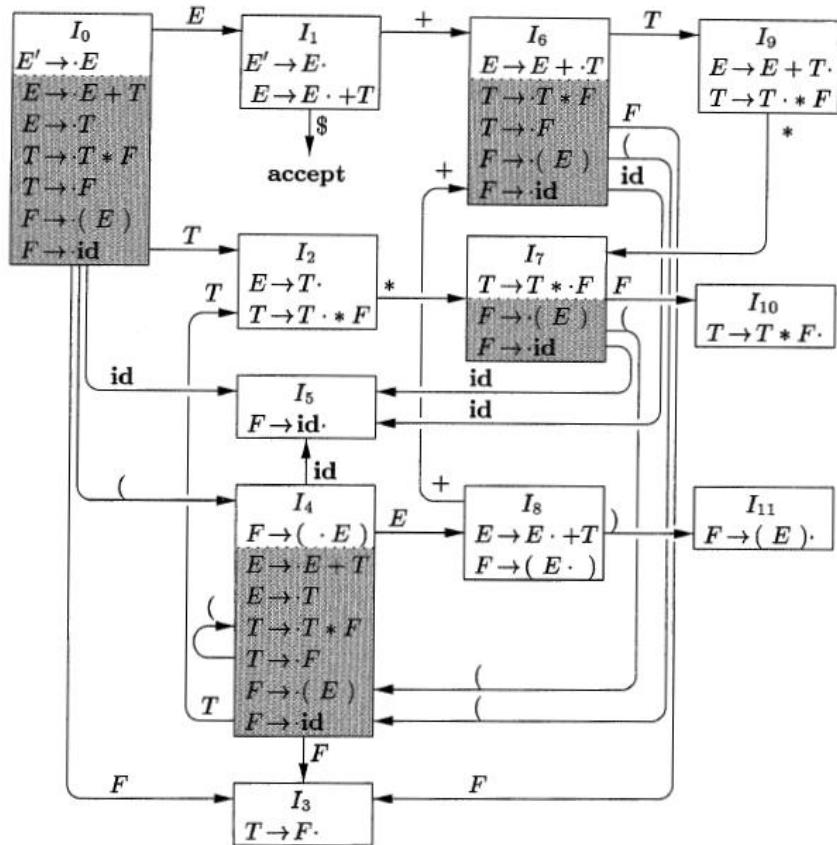


Figure 4.31: LR(0) automaton for the expression grammar (4.1)

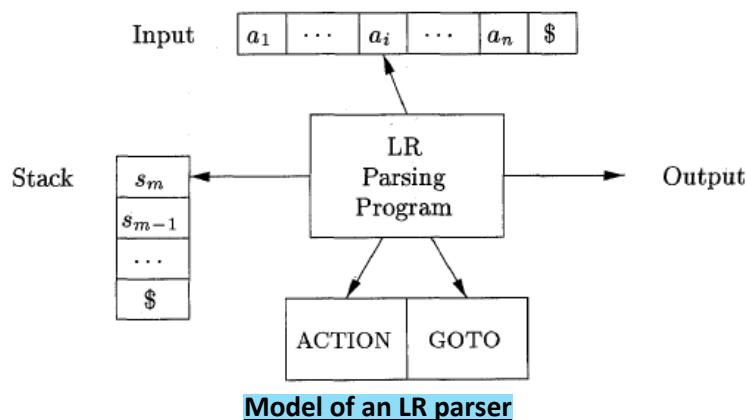
- Kernel items: the initial item, $S' \rightarrow .S$, and all items whose dots are not at the left end.
- Non-kernel items: all items with their dots at the left end, except for $S' \rightarrow .S$

The LR-Parsing Algorithm

A schematic of an **LR parser** consists of an **input**, an **output**, a **stack**, a **driver program**, and a **parsing table** that has two parts (**ACTION** and **GOTO**).

The driver program is the same for all LR parsers; only the parsing table changes from one parser to another.

The parsing program reads characters from an input buffer **one at a time**. Where a shift-reduce parser would shift a symbol, an LR parser shifts a state.



Structure of the LR Parsing Table

The parsing table consists of two parts: a parsing-action function **ACTION** and a goto function **GOTO**

1. The **ACTION** function takes as arguments **a state i and a terminal a** (or \$, the input end marker). The value of $\text{ACTION}[i, a]$ can have one of four forms:
 - a) Shift j , where j is a state. The action taken by the parser effectively shifts **input a** to the stack, but uses state **j to represent a**.
 - b) Reduce $A \rightarrow \beta$.
 - c) Accept.
 - d) Error.
2. We extend the **GOTO** function, defined on **sets of items, to states**: if $\text{GOTO}[I_i, A] = I_j$ then **GOTO** also maps a **state i** and a **nonterminal A** to state **j**.

Example construct LR(0) parsing table for the following grammar

$$\begin{array}{ll}
 (1) \quad E \rightarrow E + T & (4) \quad T \rightarrow F \\
 (2) \quad E \rightarrow T & (5) \quad F \rightarrow (E) \\
 (3) \quad T \rightarrow T * F & (6) \quad F \rightarrow \text{id}
 \end{array}$$

Note: We don't give number to extra added transition $E' \rightarrow E$.

The codes for the actions are:

1. **si** means shift and stack state i ,
2. **rj** means reduce by the production numbered j ,
3. **acc** means accept,
4. blank means error.

States	ACTION						GOTO		
	id	+	*	()	\$	E	T	F
I0	S5				S4		1	2	3
I1		S6				Acc			
I2			S7						
I3	r4	r4	S7/r4	r4	r4	r4			
I4	S5			S4			8	2	3
I5	r6	r6	r6	r6	r6	r6			
I6	S5			S4			9	3	
I7	s5			S4				10	
I8		S6			S11				
I9	r1	r1	r1/S7	r1	r1	r1			
I10	r3	r3	r3	r3	r3	r3			
I11	r5	r5	r5	r5	r5	r5			

LR(0) items table

Note:- that state I3 and I9 has shift-reduce conflicts. A conflict (Shift-reduce or Reduce-reduce) implies that parser does not know handle at the corresponding stage of parsing.

Constructing SLR-Parsing Tables

The SLR method begins with **LR(0) items** and **LR(0) automata**. That is, given a grammar, G, we augment G to produce G', with a new start symbol S'. From G', we construct C, **the canonical collection of sets of items for G' together with the GOT0 function**.

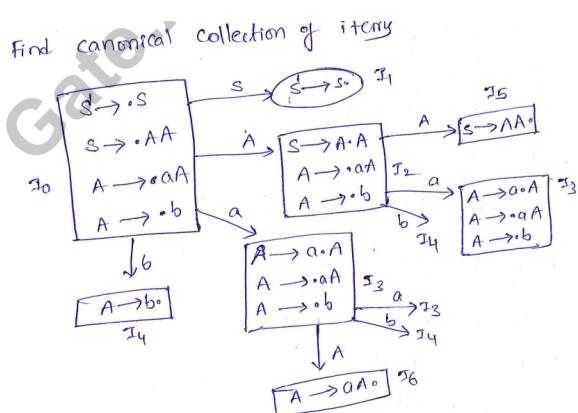
Note: - Every SLR(1) grammar is **unambiguous**, but there are many unambiguous grammars that are **not SLR(1)**.

More Example on LR(0) parser :

$$G_1: \begin{array}{ll} S \rightarrow AA & T_1 \\ A \rightarrow aA & T_2 \\ A \rightarrow b & T_3 \end{array}$$

① Augmented Grammar

$$\begin{array}{l} S' \rightarrow S \\ S \rightarrow AA \\ A \rightarrow aA \\ A \rightarrow b \end{array}$$

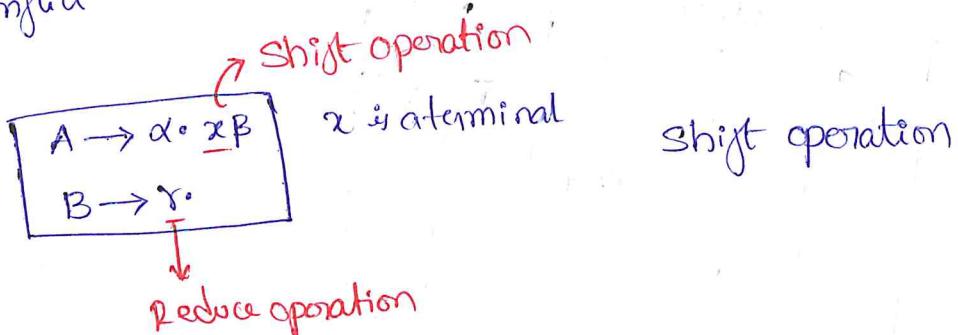
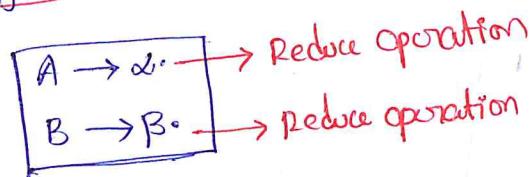
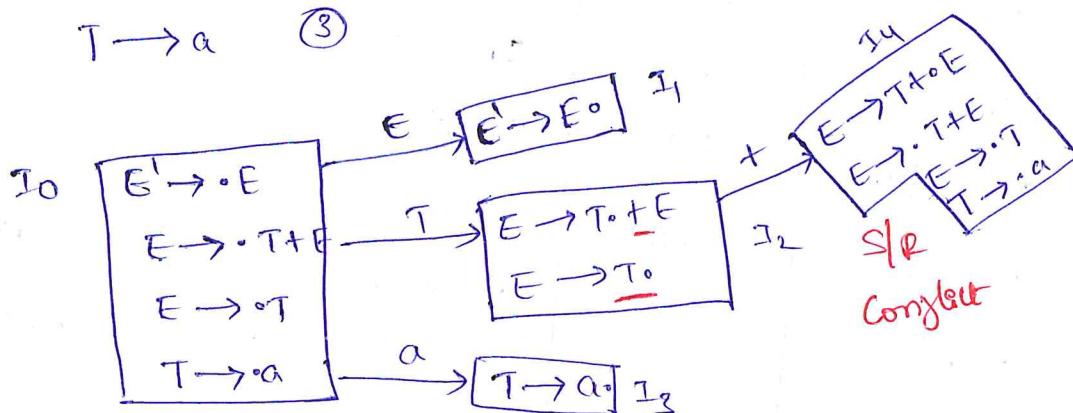
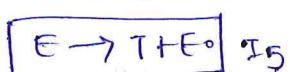


States	Action			GOTO	
	a	b	\$	S	A
I0	S3	S4		1	2
I1			Accept		5
I2	S3	S4			6
I3	S3	S4			
I4	T3	T3	T3		
I5	T1	T1	T2		
I6	T2	T2	T2		

Stack	input	Action
40	abb\$	S3
\$003	b\$	S1
\$003b\$	b\$	T3 A->b
\$003A	b\$	6
\$003A6	b\$	T2 A->aA
\$0A	b\$	2
\$0A2	b\$	S4
\$0A2b\$	\$	T3 A->b
\$0A2b4	\$	T1 B->AA
\$0A2A5	\$	1
40S	\$	Accept
40S1	\$	

abb
wEL(G)

Conflicts: LR(0)

↳ if \exists multiple entries in a cell in the parse-table.① SR Conflict② RR ConflictRR Conflict:Eg 1: $E \rightarrow T+E$ ① $E \rightarrow T$ ② $T \rightarrow a$ ③GOTO(I_4, E)GOTO (I_4, T)GOTO (I_4, a)

Example 4.48: Every SLR(1) grammar is unambiguous, but there are many unambiguous grammars that are not SLR(1). Consider the grammar with productions

$$\begin{array}{l} S \rightarrow L = R \mid R \\ L \rightarrow *R \mid \text{id} \\ R \rightarrow L \end{array} \quad (4.49)$$

Think of L and R as standing for l -value and r -value, respectively, and $*$ as an operator indicating "contents of."⁵ The canonical collection of sets of LR(0) items for grammar (4.49) is shown in Fig. 4.39.

$I_0: S' \rightarrow \cdot S$	$I_5: L \rightarrow \text{id} \cdot$
$S \rightarrow \cdot L = R$	$I_6: S \rightarrow L = \cdot R$
$S \rightarrow \cdot R$	$R \rightarrow \cdot L$
$L \rightarrow \cdot * R$	$L \rightarrow \cdot * R$
$L \rightarrow \cdot \text{id}$	$L \rightarrow \cdot \text{id}$
$R \rightarrow \cdot L$	
$I_1: S' \rightarrow S \cdot$	$I_7: L \rightarrow * R \cdot$
$I_2: S \rightarrow L \cdot = R$	$I_8: R \rightarrow L \cdot$
$R \rightarrow L \cdot$	
$I_3: S \rightarrow R \cdot$	$I_9: S \rightarrow L = R \cdot$
$I_4: L \rightarrow * \cdot R$	
$R \rightarrow \cdot L$	
$L \rightarrow \cdot * R$	
$L \rightarrow \cdot \text{id}$	

Canonical collection of sets of LR(0) items

Consider the set of items I_2 . Since there is both a shift and a reduce entry in ACTION [2, =], **state 2 has a shift/reduce conflict on input symbol =**.

Note:- Grammar (4.9) is not ambiguous still we can't have SLR parser for it. SLR parser construction method is not powerful enough to remember enough left context to decide what action the parser should take on input =.

Note:- There are **unambiguous grammars** for which **every LR parser** construction method will produce a **parsing action table with parsing action conflicts**.

SLR(1) parser:
 \hookrightarrow Simple Extension to LR(0) parser.
 \hookrightarrow parse-table (Hinb)

Instead of placing the reduce operation under all the terminals.
 place them under follow (LHS) variables.

SLR(1) parse table for the Grammar

	t	a	$\$$	E	T
I_0		S_2		1 2	
I_1			Accept		
I_2	S_4		t_2		
I_3	t_3		t_3		
I_4		S_3		5 6	
I_5			t_1		

$$\begin{array}{l} \text{①} \\ E \rightarrow T+E \\ \text{②} \\ E \rightarrow T \\ \text{③} \\ T \rightarrow a \end{array}$$

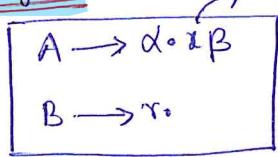
$\text{Follow}(E) = \{ \$ \}$
 $\text{Follow}(T) = \{ \$, t \}$
 $E \rightarrow T+E \cdot \quad r_1$
 $E \rightarrow T+E \cdot \quad \text{Follow}(E) = \{ \$ \}$

(50)

SR & RR conflicts in SLR(1) parse-table:

Ph: 844-844-0102

SR Conflict:



SR-Conflict in LR(0)

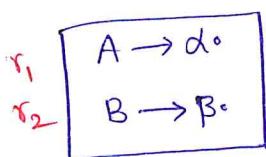
SLR(1)

First(x) ⊂ Follow(B)

$x \in \text{Follow}(B) \neq \emptyset$

S/R Conflict in SLR(1)

RR Conflict:



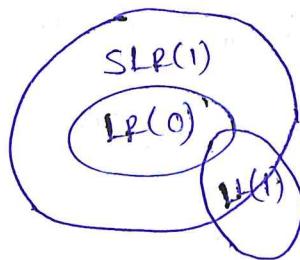
Follow(A) ⊂ Follow(B) ≠ ∅

RR Conflict

SLR(1) Grammar:

↳ if SLR(1) parse table has no conflicts.

Fewer entries
than LR(0)



SLR(1) is more powerful than
LR(0) parser.

LL(1) is not a proper subset
of LR(0)

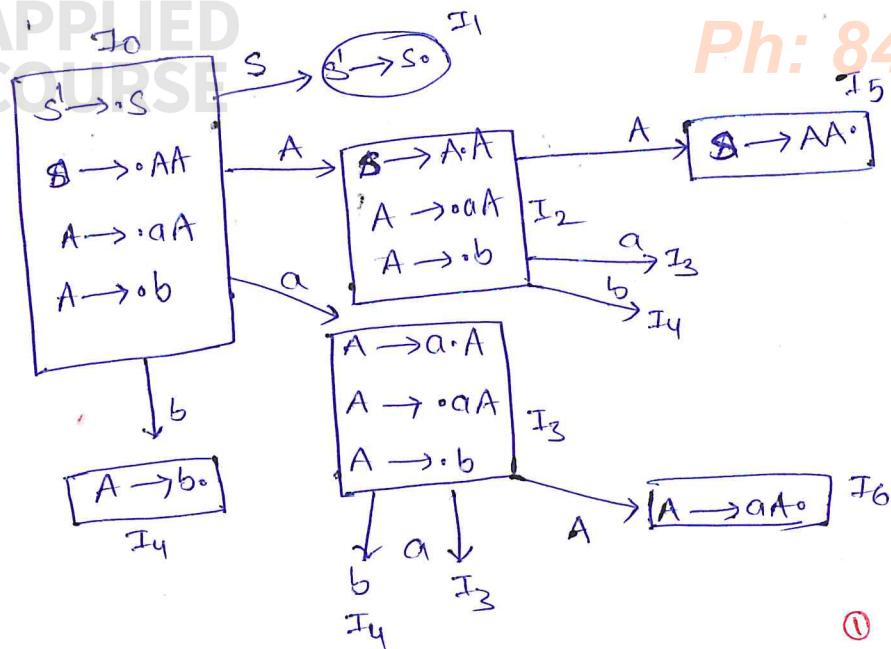
Gr:

$$S' \rightarrow^* S$$

$$1. \quad S \rightarrow^* A A$$

$$2. \quad A \rightarrow^* a A$$

$$3. \quad A \rightarrow^* b$$



- ① $S \rightarrow AA$
- ② $A \rightarrow aA$
- ③ $A \rightarrow b$

$$\text{Follow}(A) = \{a, b, \$\}$$

$$\text{Follow}(S) = \{\$\}$$

	a	b	\$	S	A
I0	S_3	S_4		1	2
I1					5
I2	S_3	S_4			6
I3	S_3	S_4			
I4	r_3	r_3	r_3		
I5			r_1		
I6	r_2	r_2	r_2		

Both

LR(0)

SLR(1)

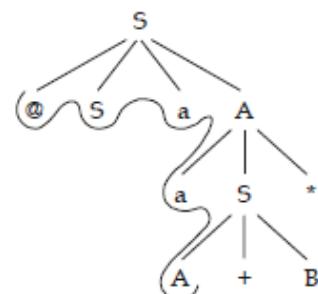
No S|R|R|R
conflicts.

Viable prefix :

Viable prefix means complete or half complete string produced by the tree formed by given grammar. Viable prefixes are the prefixes of right sentential forms that do not extend beyond the end of its handle. A viable prefix either has no handle or just one possible handle on the extreme RIGHT which can be reduced.

Example :

$$G = \{S \rightarrow @SaA \mid A + B, A \rightarrow aS^* \mid a, B \rightarrow b - B \mid b\}$$

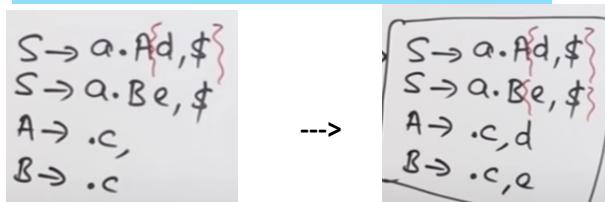


@SaA is viable prefix

More Powerful LR Parsers

1. The "canonical-LR" or just "LR" method, which makes full use of the lookahead symbol(s). This method uses a large set of items, called the LR(1) items. CLR(1) and LR(1) are same.
2. The "lookahead-LR" or "LALR" method, which is based on the LR(0) sets of items, and has many fewer states than typical parsers based on the LR(1) items. By carefully introducing look-aheads into the LR(0) items, we can handle many more grammars with the LALR method than with the SLR method, and build parsing tables that are no bigger than the SLR tables.

NOTE : Lookahead means `first()` of immediate part of variable whose closure has been put.
for example, consider following productions



Here after opening the closure of A we have d, dollar sign means we will find first of d, dollar which is d. Which will become lookahead of A production.

$A \rightarrow \cdot \in$ is considered a reduce state

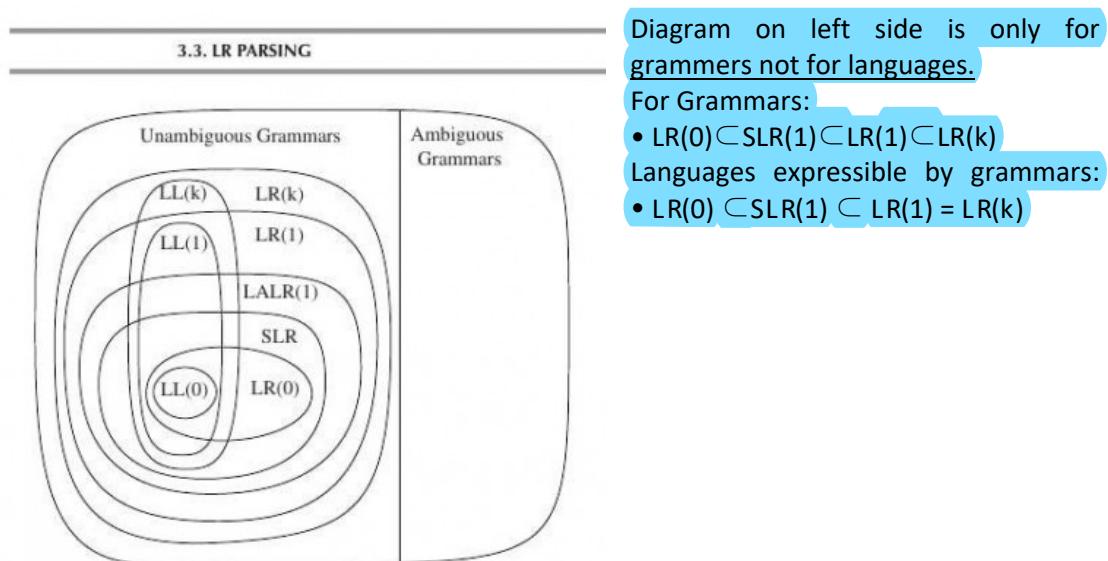


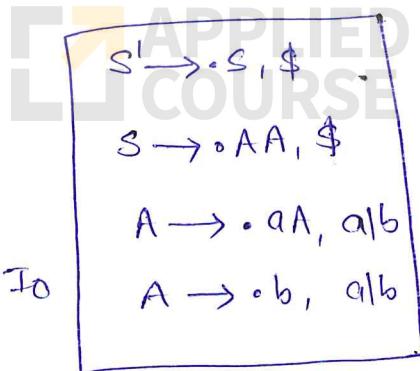
FIGURE 3.29. A hierarchy of grammar classes.

CLR(1) & LALR(1) :
LR(1) items: LR(0) items + lookahead

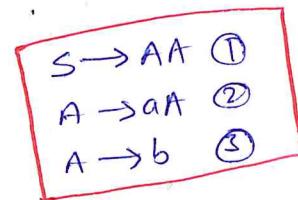
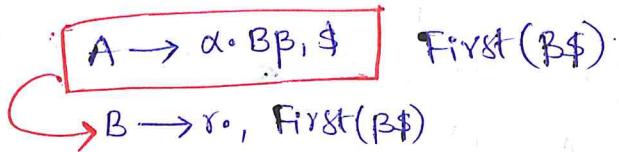
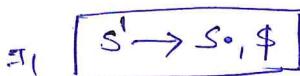
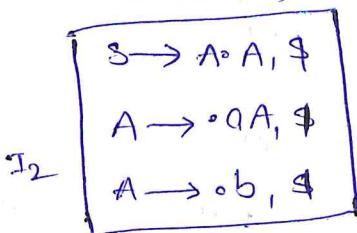
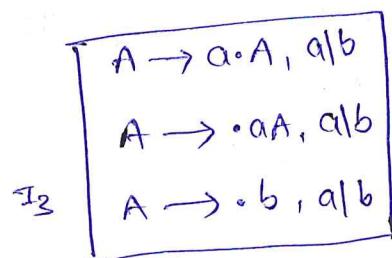
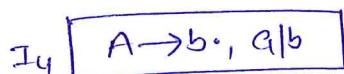
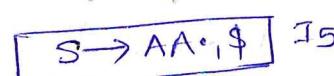
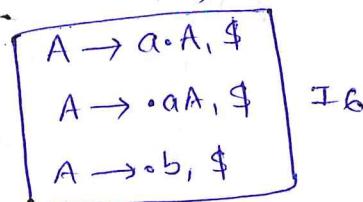
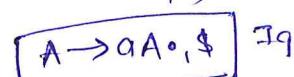
Eg: $S \rightarrow \dot{A}A \Rightarrow$ Augmented Grammar

$A \rightarrow aA$
 $A \rightarrow b$

$S' \rightarrow S$
 $S \rightarrow AA$
 $A \rightarrow aA$
 $A \rightarrow b$



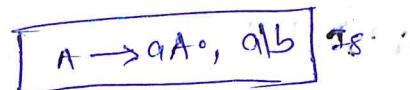
Ph: 844-844-0102

GOTO (I_0, S)GOTO (I_0, A)GOTO (I_0, a)GOTO (I_0, b)GOTO (I_2, A)GOTO (I_2, a)GOTO (I_6, A)GOTO (I_6, a)

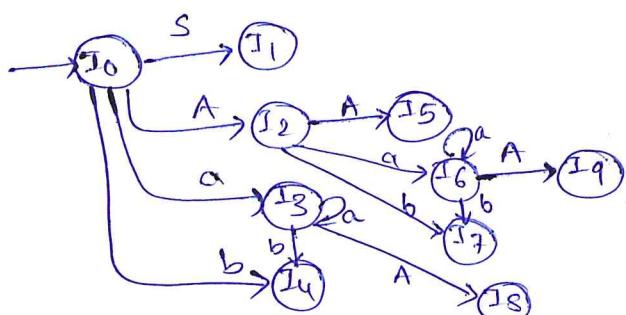
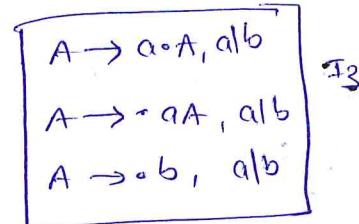
I6

GOTO (I_6, b)

I7

GOTO (I_3, A)GOTO (I_3, b)

I4

GOTO (I_3, a)

	a	b	\$	s	A
I ₀	S ₃	S ₄		1	2
I ₁				Accept	
I ₂	S ₆	S ₇			5
I ₃	S ₃	S ₄			8
I ₄	R ₃	R ₃			
I ₅			R ₁		
I ₆	S ₆	S ₇			9
I ₇			R ₃		
I ₈	R ₂	R ₂			
I ₉			R ₂		

LALR(1) parser:

3₄6, 4₆7, 8₉

I₃₆ I₄₇ I₈₉

	a	b	\$	s	A
I ₀	S ₃₆	S ₄₇		1	2
I ₁				Accept	
I ₂	S ₃₆	S ₄₇			5
I ₃₆	S ₃₆	S ₄₇			89
I ₄₇	R ₃	R ₃	R ₂		
I ₅			R ₁		
I ₈₉	R ₂	R ₂	R ₂		

Ph: 844-844-0103
Number of States in CLR(1)

\leq SLR(1)

CLR(1) parser



placing r_i in fewer cells
than SLR(1)



Lesser chance of Conflict

LR(0): Full row

SLR(1): Follow of LHS

CLR(1): Look ahead

Merge the States

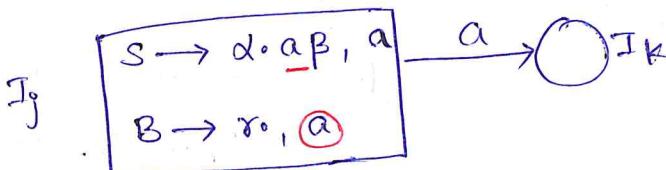
NO. of States in LR(0)

\leq SLR(1) = LALR(1)

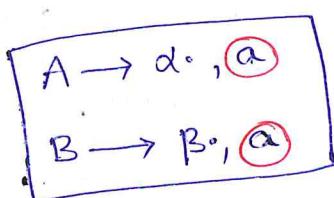
\leq CLR(1)

SLP(1), LR(0)

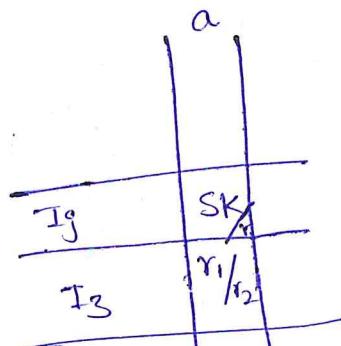
Conflicts : S|R & R|R



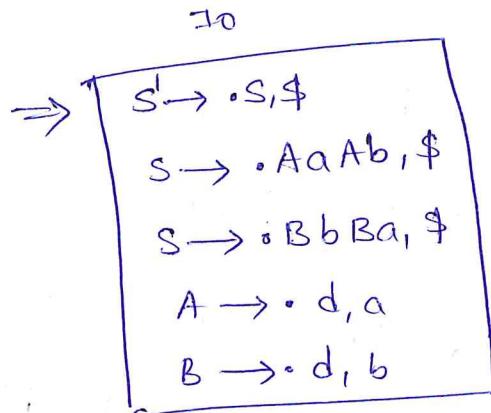
S|R Conflict in CLPC(1)



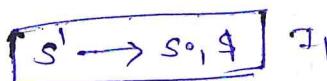
R|R Conflict in CLPC(1)



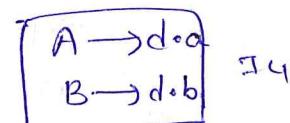
- Given:
- ① $S \rightarrow AaAb$
 - ② $S \rightarrow BbBa$
 - ③ $A \rightarrow d$
 - ④ $B \rightarrow d$



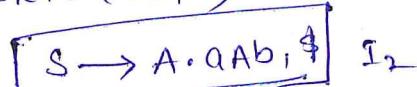
GOTO(I_0, S)



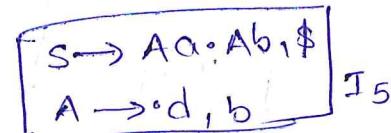
GOTO(I_0, d)



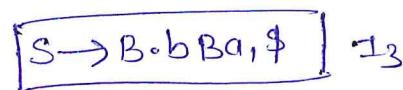
GOTO(I_0, A)



GOTO(I_2, a)



GOTO(I_0, B)



GOTO (I₃, b)

$S \rightarrow Bb \cdot Ba, \$$	I ₆
$B \rightarrow \cdot B, a$	

GOTO (I₅, A)

$S \rightarrow AaA \cdot b, \$$	I ₇
---------------------------------	----------------

GOTO (I₇, b)

$S \rightarrow AaAb \cdot, \$$	I ₁₁
--------------------------------	-----------------

GOTO (I₅, d)

③

$A \rightarrow d \cdot, b$	I ₈
----------------------------	----------------

GOTO (I₉, a)

$S \rightarrow BbBa \cdot, \$$	I ₁₂
--------------------------------	-----------------

GOTO (I₆, B)

$S \rightarrow BbB \cdot a, \$$	I ₉
---------------------------------	----------------

GOTO (I₆, d)

④

$B \rightarrow d \cdot, a$	I ₁₀
----------------------------	-----------------

No SIR and RIR Conflicts. CLR(1) Grammar.

	a	b	d	\$	S	A	B
I ₀			S ₄			1	2 3
I ₁					Accept		
I ₂	S ₅						
I ₃		S ₆					
I ₄	r ₃	r ₄					
I ₅			S ₈			7	
I ₆			S ₁₀			9	
I ₇		S ₁₁					
I ₈			r ₃				
I ₉		S ₁₂					
I ₁₀	r ₄						
I ₁₁			r ₁				
I ₁₂			r ₂				

No conflicts

CLR(1) Grammar

Some Properties of LR parser :

- 1) Reduce entries in SLR(1) can be more than CLR(1) vice versa is also true for a given grammar
- 2) The number of shift entries in SLR(1) and LALR(1) is equal.
- 3) In an LR(1) machine with a shift/reduce conflict, we should shift if the priority of the rule is **lower** than the priority of the incoming symbol, or if the priorities are the same and are both **right associative**.



Combining states in CLR(1) parse table.

→ If CLR(1) parse table has no LR conflicts then no LR conflicts in LALR(1).

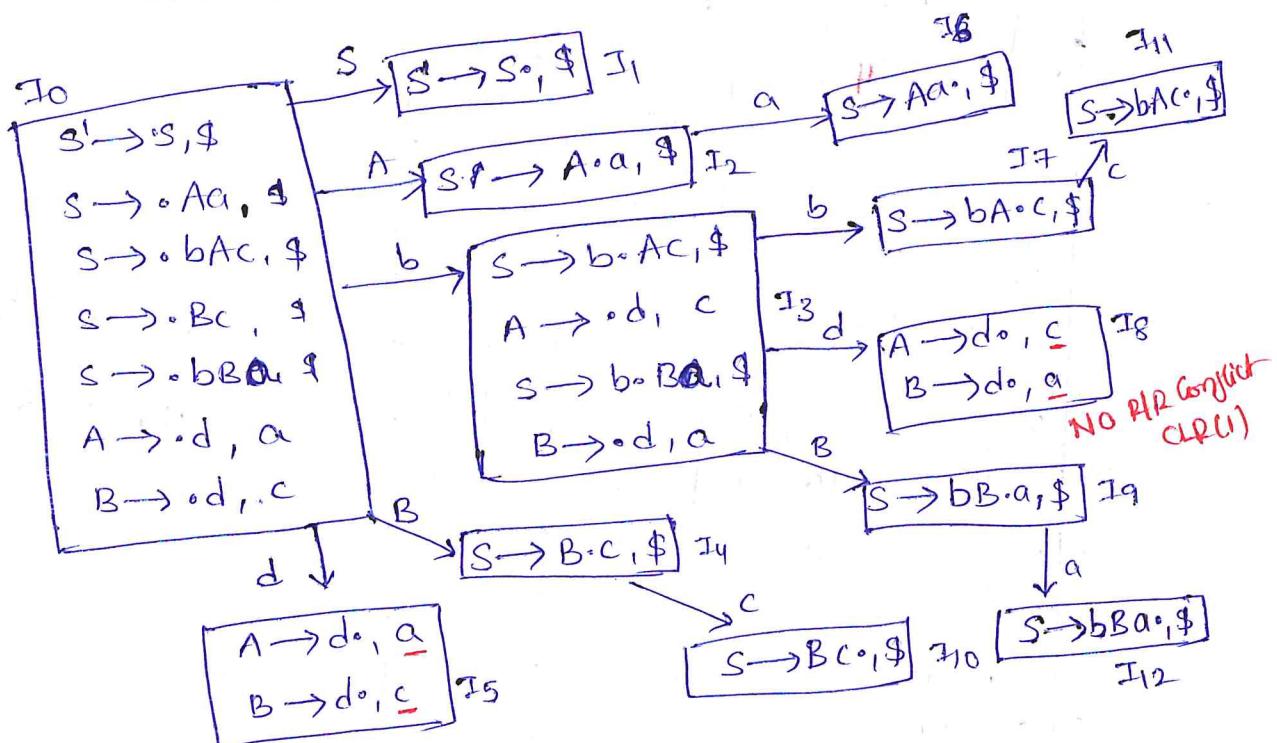
→ If CLR(1) parse table has no LR conflicts, then there may/may not exist LR conflicts in LALR(1).

Eg1:

- 1 $S \rightarrow Aa$
- 2 $S \rightarrow bAc$
- 3 $S \rightarrow Bc$
- 4 $S \rightarrow bB\alpha$
- 5 $A \rightarrow d$
- 6 $B \rightarrow d$

- 1 $S \rightarrow S$
- 2 $S \rightarrow Aa$
- 3 $S \rightarrow bAc$
- 4 $S \rightarrow Bc$
- 5 $S \rightarrow bB\alpha$
- 6 $A \rightarrow d$
- 7 $B \rightarrow d$

CLR(1) but not LALR(1)



No Conflict in CLR(1)

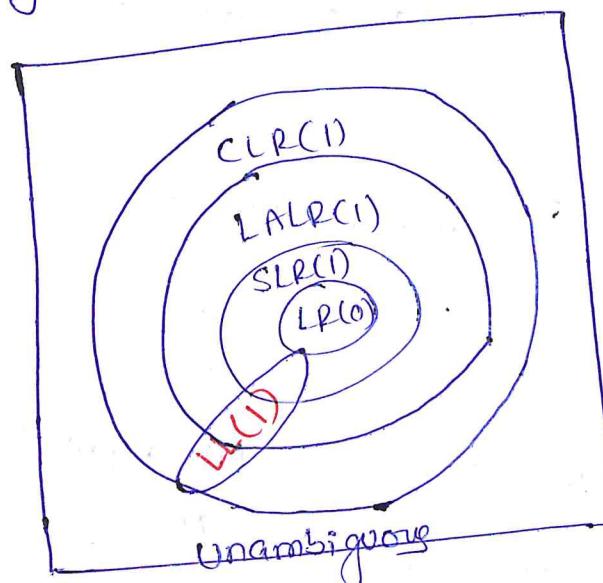
I₅₈

$$\begin{array}{l} ⑤ A \rightarrow d \cdot \underline{a} \underline{c} \\ ⑥ B \rightarrow d \cdot \underline{a} \underline{c} \end{array}$$

R1R Conflict in LALR(1)

	a	c
I ₅₈	r ₅ /r ₆	r ₅ /r ₆

Every LALR(1) is CLR(1), but not vice versa.



CLR(1) parser is more powerful.

LALR(1) parser \rightarrow practice
fewer states

operator precedence parser:

\hookrightarrow BOP, ambiguous & unambiguous

\hookrightarrow cannot work for all CFG

$\hookrightarrow +, -, *, /, \%, \dots$

operator Grammar:

\hookrightarrow CFG $\left\{ \begin{array}{l} \text{no } \epsilon\text{-transitions} \\ \text{No AB on RHS} \end{array} \right.$

\hookrightarrow Adjacent Variables

AaB

all terminals
 \downarrow
operator

Operator Precedence Parsing

It can parse an ambiguous grammar also.

Operator Precedence Parsing.

- * For a small grammar we can easily construct efficient shift-reduce parsers.
- * These grammars are having no production R.H.S is ϵ
- * And no two adjacent non-terminals on R.H.S.

Example-

$$\begin{aligned} E &\rightarrow EAE \mid (E) \mid -E \mid id \\ A &\rightarrow + \mid - \mid * \mid / \mid \uparrow \end{aligned}$$

The given grammar is not Operator grammar, as there are more than 1 consecutive variables (EAE) at RHS. So, modified corresponding operator grammar is:

$$E \rightarrow E+E \mid E-E \mid E*E \mid E/E \mid E\uparrow E \mid (E) \mid -E \mid id$$

Intermediate Code Generation :

Static Single assignment form :

All assignments in SSA are to variables with distinct names; hence the term static single-assignment.

$$\begin{array}{ll} p = a + b & p_1 = a + b \\ q = p - c & q_1 = p_1 - c \\ p = q * d & p_2 = q_1 * d \\ p = e - p & p_3 = e - p_2 \\ q = p + q & q_2 = p_3 + q_1 \end{array}$$

(a) Three-address code. (b) Static single-assignment form.

Example :

$$\begin{array}{ll} \text{if } (a < b + c) & t_1 = b + c; \\ a = a - c; & \Rightarrow t_2 = a < t_1; \\ c = b * c; & \text{if } z \text{ } t_2 \text{ goto } L_0; \\ & a = a - c; \\ & L_0: c = b * c; \end{array}$$

3 address code :

which contains 3 terms in total. and while transferring any code to 3 address code you need not to store value in variable. You can use them directly. like $T1=a+T2$.

Syntax-Directed Translation

Syntax-Directed Definitions

A syntax-directed definition specifies the values of attributes by associating semantic rules with the grammar productions. For example, an infix-to-postfix translator might have a production and rule

PRODUCTION	SEMANTIC RULE
$E \rightarrow E_1 + T$	$E.code = E_1.code \parallel T.code \parallel '+'$

A syntax-directed definition (SDD) is a context-free grammar together with, attributes and rules. Attributes are associated with grammar symbols and rules are associated with productions. If X is a symbol and a is one of its attributes, then we write X.a to denote the value of a at a particular parse-tree node labeled X.

Note - Attributes may be of any kind: numbers, types, table references, or strings, for instance.

Inherited and Synthesized Attributes

1. A synthesized attribute for a nonterminal A at a parse-tree node N is defined by a semantic rule associated with the production at N. A synthesized attribute at node N is defined only in terms of attribute values at the children of N and at N itself.
2. An inherited attribute for a nonterminal B at a parse-tree node N is defined by a semantic rule associated with the production at the parent of N. An inherited attribute at node N is defined only in terms of attribute values at N's parent, N itself, and N's siblings.

S-Attributed Definitions

An SDD is S-attributed if every attribute is synthesized.

L-Attributed Definitions

1. Synthesized, or
2. Inherited, but they can ask from parent or the left side siblings only. L-attributed definitions can always be evaluated by the depth first visit of the parse tree so S-attributed.

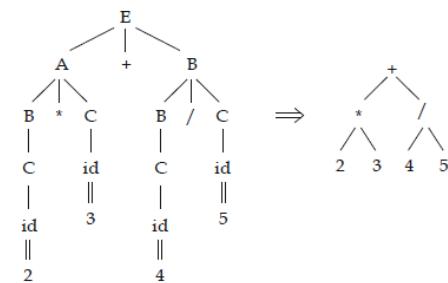
NOTE :

- 1) Only the inherited attributes that do not depend on right sibling can be evaluated by a classical preorder traversal.

Abstract syntax tree :

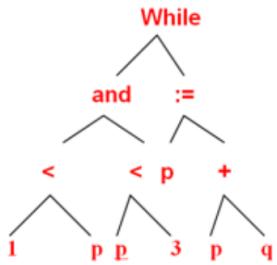
An abstract syntax tree does not involve any non-terminals. It is constructed using terminals from the grammar. Example :

$E \rightarrow A + B$	$[E.val = A.val * B.val]$
$E \rightarrow A$	$[E.val = A.val]$
$E \rightarrow B$	$[E.val = B.val]$
$A \rightarrow B * C$	$[A.val = B.val - C.val]$
$A \rightarrow C$	$[A.val = C.val]$
$B \rightarrow B/C$	$[B.val = B.val + C.val]$
$B \rightarrow C$	$[B.val = C.val]$
$C \rightarrow id$	$[C.val = id]$



Input $\rightarrow 2 * 3 + 4/5$

WHILE (1 < P) AND (P < 3) { P = P + Q}



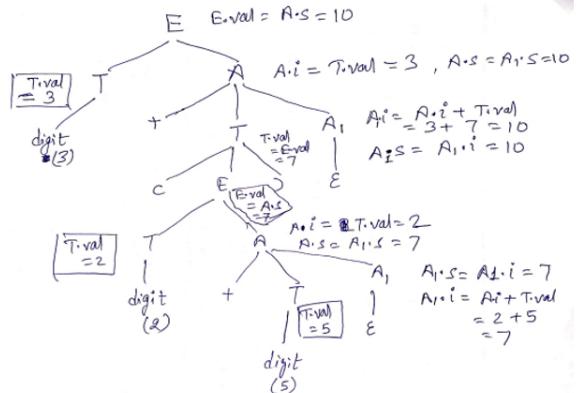
Therefore, number of edges = 12

New question :

Consider the following SDD:

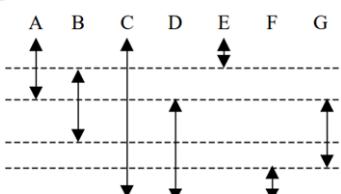
$E \rightarrow TA$	$\{A.i = T.val; E.val = A.s\}$
$A \rightarrow + T A_1$	$\{A_1.i = A.i + T.val; A.s = A_1.s\}$
$A \rightarrow \epsilon$	$\{A.s = A.i\}$
$T \rightarrow (E)$	$\{T.val = E.val\}$
$T \rightarrow \text{digit}$	$\{T.val = \text{digit.lexval}\}$

For the string $3 + (2 + 5)$, what is the value of the 'val' attribute of root node E (i. e., E.val)? _____



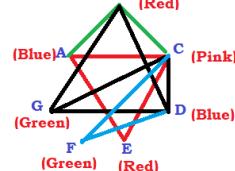
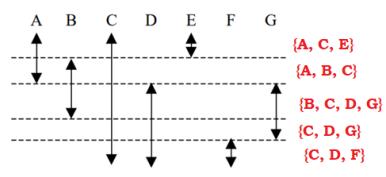
For the time sequence of variables shown in the figure below, what is the minimum number of registers that will be needed to run this code without having to store any data to memory?

Answer: (b)
Solution:



[Note: Arrows represent time that variable is active]

- (a) 3 (b) 4 (c) 5 (d) 6

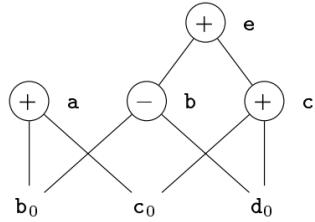


Here four colors are required. So, required number of registers = 4.

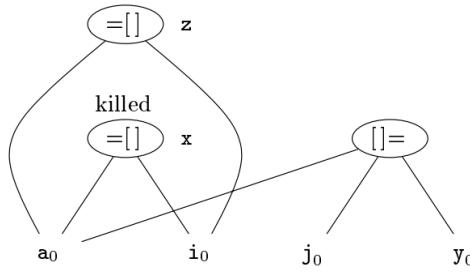
Code Generation

DAG representation :

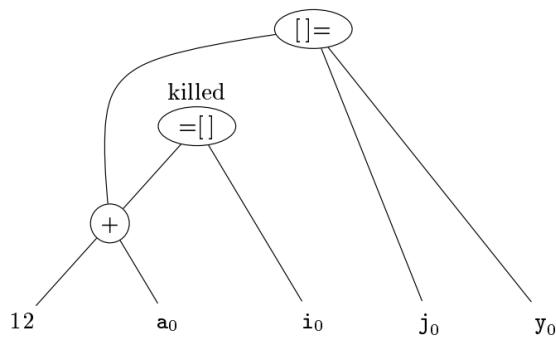
$a = b + c$
 $b = b - d$
 $c = c + d$
 $e = b + c$



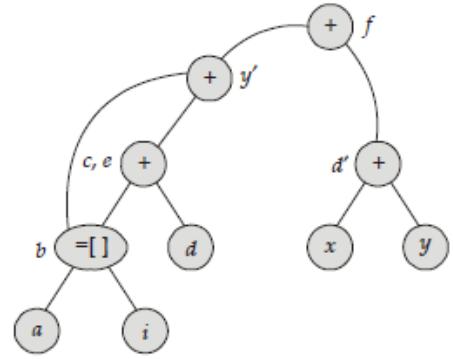
$x = a[i]$
 $a[j] = y$
 $z = a[i]$



$b = 12 + a$
 $x = b[i]$
 $b[j] = y$



$b = a[i]$
 $c = b + d$
 $e = b + d$
 $d = x + y$
 $y = b + e$
 $f = d + y$



Elimination of unnecessary instructions in object code, or the replacement of one sequence of instructions by a faster sequence of instructions that does the same thing is usually called **code improvement** or **code optimization**.

Example :

```
void quicksort(int m, int n)
    /* recursively sorts a[m] through a[n] */
{
    int i, j;
    int v, x;
    if (n <= m) return;
    /* fragment begins here */
    i = m-1; j = n; v = a[n];
    while (1) {
        do i = i+1; while (a[i] < v);
        do j = j-1; while (a[j] > v);
        if (i >= j) break;
        x = a[i]; a[i] = a[j]; a[j] = x; /* swap a[i], a[j] */
    }
    x = a[i]; a[i] = a[n]; a[n] = x; /* swap a[i], a[n] */
    /* fragment ends here */
    quicksort(m,j); quicksort(i+1,n);
}
```

Figure 9.1: C code for quicksort

(1)	i = m-1	(16)	t7 = 4*i
(2)	j = n	(17)	t8 = 4*j
(3)	t1 = 4*n	(18)	t9 = a[t8]
(4)	v = a[t1]	(19)	a[t7] = t9
(5)	i = i+1	(20)	t10 = 4*j
(6)	t2 = 4*i	(21)	a[t10] = x
(7)	t3 = a[t2]	(22)	goto (5)
(8)	if t3 < v goto (5)	(23)	t11 = 4*i
(9)	j = j-1	(24)	x = a[t11]
(10)	t4 = 4*j	(25)	t12 = 4*i
(11)	t5 = a[t4]	(26)	t13 = 4*n
(12)	if t5 > v goto (9)	(27)	t14 = a[t13]
(13)	if i >= j goto (23)	(28)	a[t12] = t14
(14)	t6 = 4*i	(29)	t15 = 4*n
(15)	x = a[t6]	(30)	a[t15] = x

Figure 9.2: Three-address code for fragment in Fig. 9.1

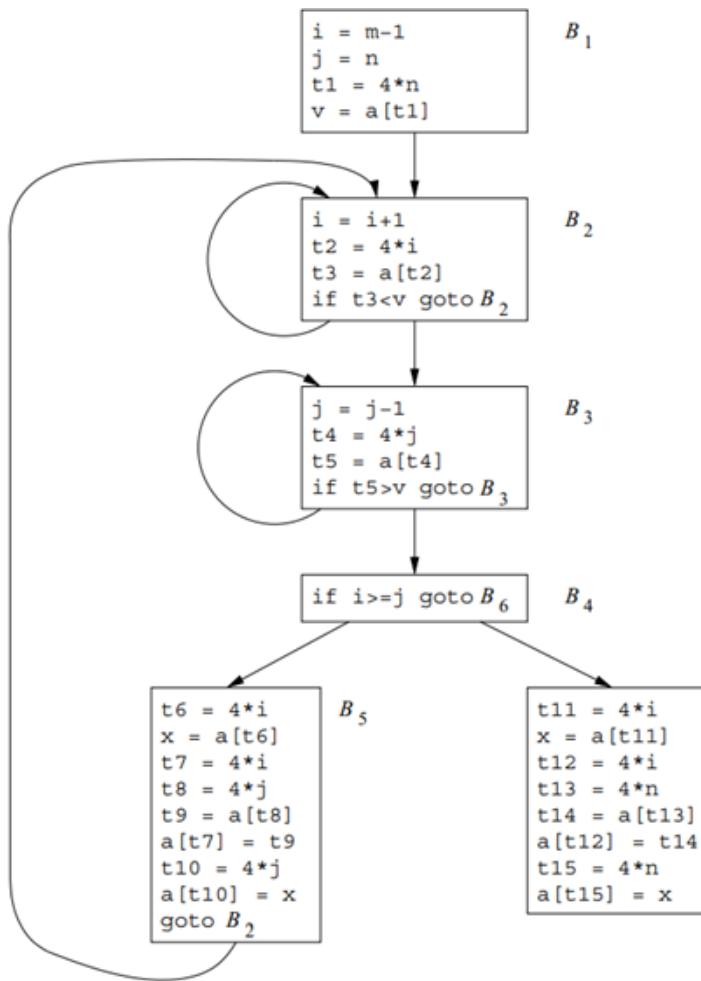


Figure 9.3: Flow graph for the quicksort fragment

Example of Common subexpression...

In the following code:

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

it may be worth transforming the code to:

```
tmp = b * c;
a = tmp + g;
d = tmp * e;
```

if the cost of storing and retrieving tmp is less than the cost of calculating $b * c$ an extra time.

1) Global common subexpressions elimination :

An expression E is called a **common subexpression** if E was previously computed and the values of the variables in E have not changed since the previous computation. We avoid recomputing E if we can use its previously computed values.

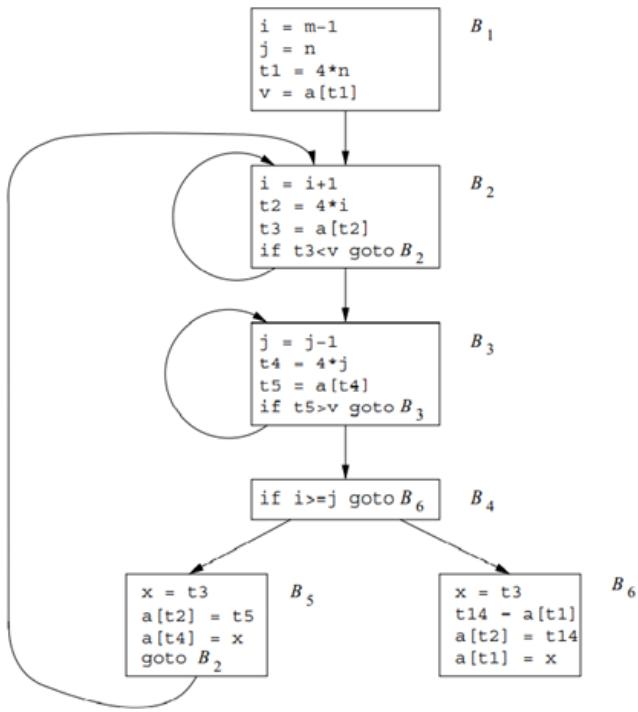


Figure 9.5: B_5 and B_6 after common-subexpression elimination

block B_5 shown in Fig. 9.4(a) recalculates $4*i$ and $4*j$, although none of these calculations were requested explicitly by the programmer.

$t6 = 4*i$	$t6 = 4*i$
$x = a[t6]$	$x = a[t6]$
$t7 = 4*i$	$t7 = 4*i$
$t8 = 4*j$	$t8 = 4*j$
$t9 = a[t8]$	$t9 = a[t8]$
$a[t7] = t9$	$a[t7] = t9$
$t10 = 4*j$	$t10 = 4*j$
$a[t10] = x$	$a[t10] = x$
<code>goto B2</code>	<code>goto B2</code>

(a) Before.

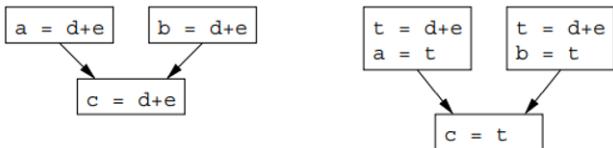
B_5

$t6 = 4*i$
$x = a[t6]$
$t8 = 4*j$
$t9 = a[t8]$
$a[t6] = t9$
$a[t8] = x$
<code>goto B2</code>

(b) After.

Figure 9.4: Local common-subexpression elimination

2) Copy progression :



$x = t3$
$a[t2] = t5$
$a[t4] = t3$
<code>goto B2</code>

Figure 9.7: Basic block B_5 after copy propagation

3) Dead code Elimination :

A variable is live at a point in a program if its value can be used subsequently otherwise, it is dead at that point. for example, $x=4$; $x=y+z$; $a=x+4$; then x is dead at first statement, but if it is $x=4$; $a=x+2$; then live. If the value of an expression is constant and using constant value itself we can eliminate one assignment expression. This is known as **constant folding**.

$a[t2] = t5$
$a[t4] = t3$
<code>goto B2</code>

4) Code motion :

If the value of an expression is not changing while it is in loop than we can transfer that expression outside the loop, maybe at block before loop. This is known as code motion.

5) Induction variable and strength reduction :

A variable x is said to be an induction variable if there is a positive or negative constant c such that each time x is assigned, its value increased by c . For instant, i and $t2$ are induction variables in the loop containing B_2 in fig. 9.5.

The transformation of replacing an expensive operation, such as multiplication, by a cheaper one, such as addition, is known as strength reduction. Note the values of j and $t4$ remain in lock step; every time the value of j decreases by 1, the value of $t4$ decreases by 4, because $4*j$ is assigned to $t4$. These variables, j and $t4$, thus form a good example of a pair of induction variables. One more example $2*x$ can be replace by $x+x$.

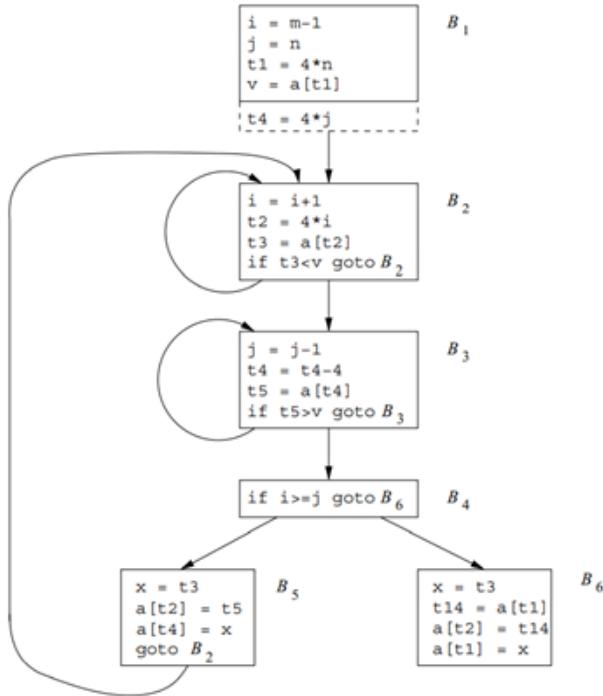


Figure 9.8: Strength reduction applied to $4*j$ in block B_3

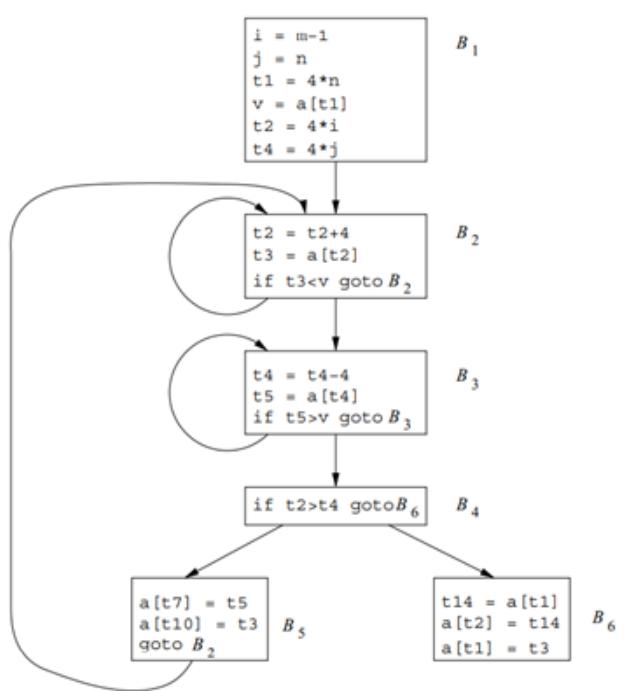
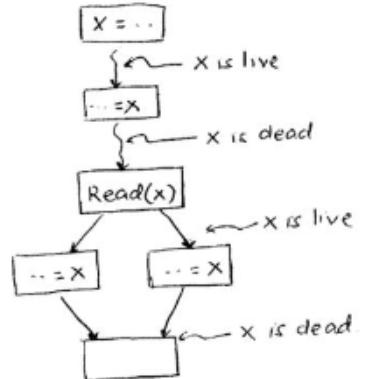


Figure 9.9: Flow graph after induction-variable elimination

Note:

- 1) Another definition of a **live variable**, a variable v is live at a program point p , if some path from p to program exit contains an r-value occurrence of v which is not preceded by an l-value occurrences of v . (l-value first then second r-value is not permitted)
- 2) Basic Block in CFG does not include start and end state.



DATA FLOW ANALYSIS

Data flow analysis refers to a body of techniques that derive information about the flow of data along program execution paths. For example, one way to implement global common subexpression elimination requires us to determine whether two textually identical expressions evaluates to the same value along any possible execution path of the program.

1) Transfer function : (The relationship between the data flow value before and after the assignment statement. Usually denoted by f_s .

usually denote f_s , takes the data-flow value before the statement and produces a new data-flow value after the statement. That is,

$$\text{OUT}[s] = f_s(\text{IN}[s]).$$

Conversely, in a backward-flow problem, the transfer function f_s for statement s converts a data-flow value after the statement to a new data-flow value before the statement. That is,

$$\text{IN}[s] = f_s(\text{OUT}[s]).$$

2) Control flow constraints :

The second set of constraints on data-flow values is derived from the flow of control. Within a basic block, control flow is simple. If a block B consists of statements s_1, s_2, \dots, s_n in that order, then the control-flow value out of s_i is the same as the control-flow value into s_{i+1} . That is,

$$\text{IN}[s_{i+1}] = \text{OUT}[s_i], \text{ for all } i = 1, 2, \dots, n-1.$$

$$\text{IN}[B] = \bigcup_{P \text{ a predecessor of } B} \text{OUT}[P].$$

When the data-flow is backwards as we shall soon see in live-variable analysis, the equations are similar, but with the roles of the IN's and OUT's reversed. That is,

$$\text{LIVE}_{in}[s] = \text{GEN}[s] \cup (\text{LIVE}_{out}[s] - \text{KILL}[s])$$

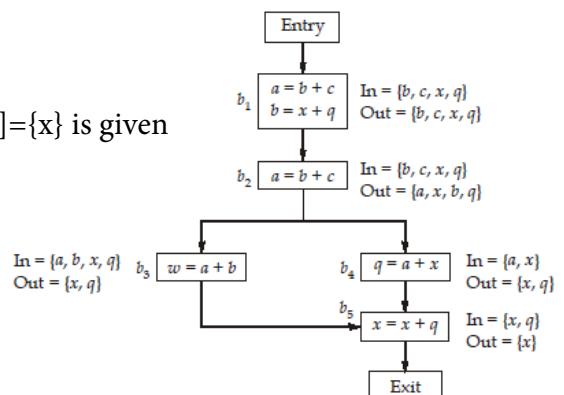
$$\text{IN}[B] = f_B(\text{OUT}[B])$$

$$\text{OUT}[B] = \bigcup_{S \text{ a successor of } B} \text{IN}[S].$$

$$\text{LIVE}_{out}[\text{final}] = \emptyset$$

$$\text{LIVE}_{out}[s] = \bigcup_{p \in \text{succ}[s]} \text{LIVE}_{in}[p]$$

out[b5]={x} is given



$$\text{gen } [b_5] = \{x, q\}$$

$$\text{kill } [b_5] = \{x\}$$

$$\text{gen } [b_4] = \{a, x\}$$

$$\text{kill } [b_4] = \{q\}$$

$$\text{gen } [b_3] = \{a, b\}$$

$$\text{kill } [b_3] = \{w\}$$

$$\text{gen } [b_2] = \{b, c\}$$

$$\text{kill } [b_2] = \{a\}$$

$$\text{gen } [b_1] = \{b, c, x, q\}$$

$$\text{kill } [b_1] = \{a, b\}$$

$$\text{In} = \text{Gen} \cup \{\text{out} - \text{kill}\}$$

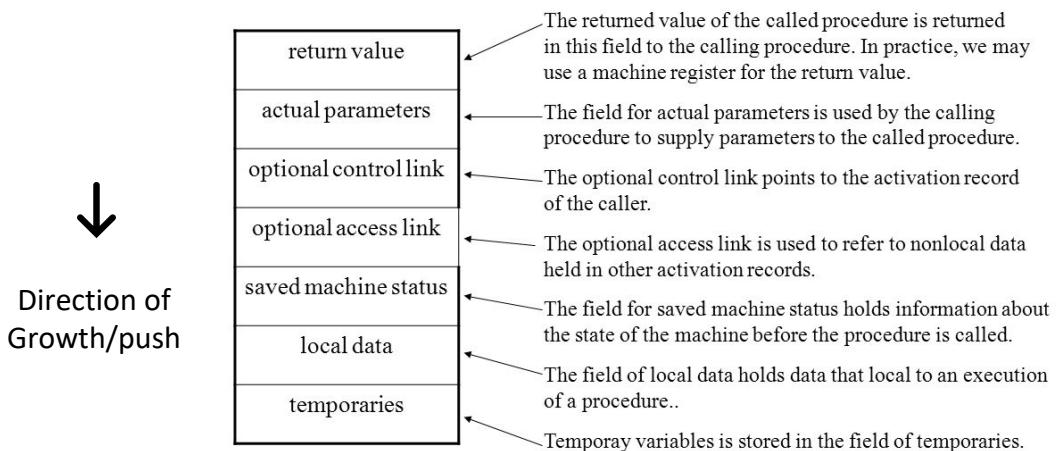
Runtime Environment

The compiler must cooperate with the operating system and other systems software to support abstractions like name, datatype, scopes operator, etc,... on the target machine.

NOTE : The process of removing physical, spatial, or temporal details or attributes is known as *abstraction*.

To do so, the compiler creates and manages a **run-time environment** in which it assumes its target programs are being executed. This environment deals with a variety of issues such as the layout and allocation of storage locations for the objects named in the source program, the mechanisms used by the target program to access variables, the linkages between procedures, the mechanisms for passing parameters, and the interfaces to the operating system, input/output devices, and other programs.

1. Activation records : Procedure calls and returns are usually managed by a run-time stack called the **control stack**. Each live activation has an activation record (sometimes called aframe) on the control stack.



Every time a procedure starts getting executed, its contents are loaded in stack in form of an activation record. **A procedure is recursive** if a new activation record begins before an earlier activation record of the same procedure has ended. **Control link** also known as **Dynamic link** in an activation record points to the activation record of the caller.