



Compiler Design

Suggested Book

Compilers: Principles, Techniques, and Tools

by Aho, Sethi, Ullman and Lam

Syllabus

CS207	Compiler Design	L 3	T 1	P 0
-------	-----------------	--------	--------	--------

Introduction: Translators, Various phases of compiler, tool based approach to compiler construction.

Lexical analysis: token, lexeme and patterns, difficulties in lexical analysis, error reporting, implementation, regular definition, transition diagrams, LEX.

Syntax Analysis: top down parsing (recursive descent parsing, predictive parsing), operator precedence parsing, bottom-up parsing (SLR, LALR, Canonical LR), YACC.

Syntax directed definitions: inherited and synthesized attributes, dependency graph, evaluation order, bottom-up and top-down evaluation of attributes, L-attributed and S-attributed Definitions.

Type checking: type system, type expressions, structural and name equivalence of types, type conversion, overloaded functions and operators, polymorphic functions.

Run time system: storage organization, activation tree, activation record, parameter passing, dynamic storage allocation, symbol table: hashing, linked list, tree structures.

Intermediate code generation: intermediate representation, translation of declarations, assignments, control flow, Boolean expressions and procedure calls, implementation issues.

Code generation and Optimization: issues, basic blocks and flow graphs, register allocation, code generation, dag representation of programs, code generation from DAGs, peephole optimization.

Suggested Readings:

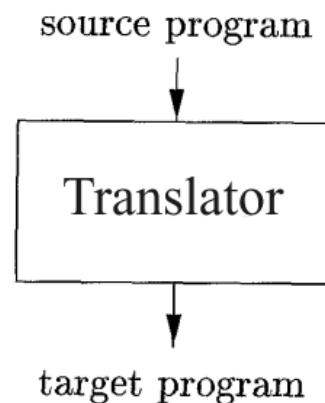
1. A. V. Aho, J. D. Ullman, Principles of Compiler Design, Narosa Publishing House.
2. J. P. Trembley, P. G. Sorensen, The Theory and Practice of Compiler Writing, McGraw Hill.
3. A. Holub, Compiler Design in C, PHI

What do we expect to achieve by the end of the course?

- Knowledge to design, develop, understand, modify/enhance, and maintain compilers for (even complex!) programming languages.
- Confidence to use language processing technology for software development.
- Become a better programmer and better software developer.

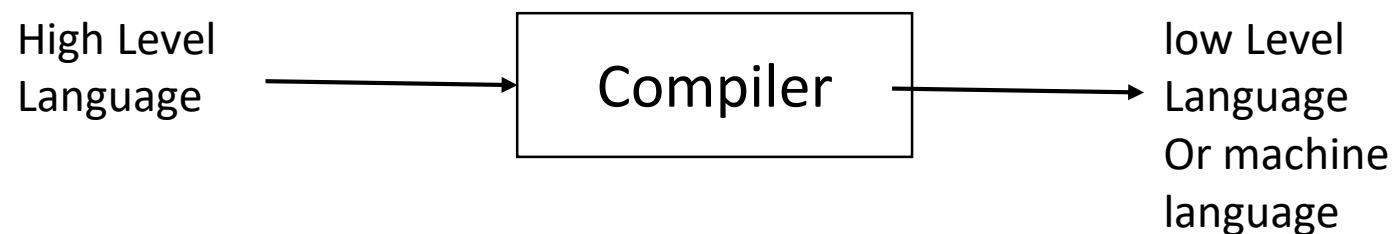
Translator

- A Translator 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

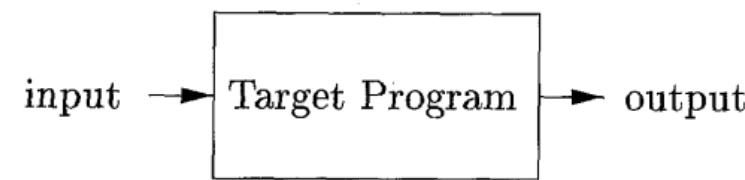


Compiler

- If the source program is a high level language such as C++ and the target language is a low level language such as assembly or machine language, then such a translator is called Compiler

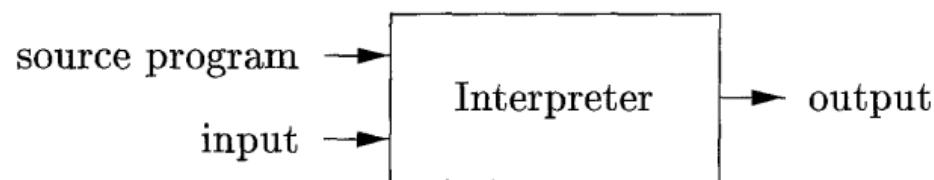


- If the target program is an executable machine-language program, it can then be called by the user to process inputs and produce outputs.



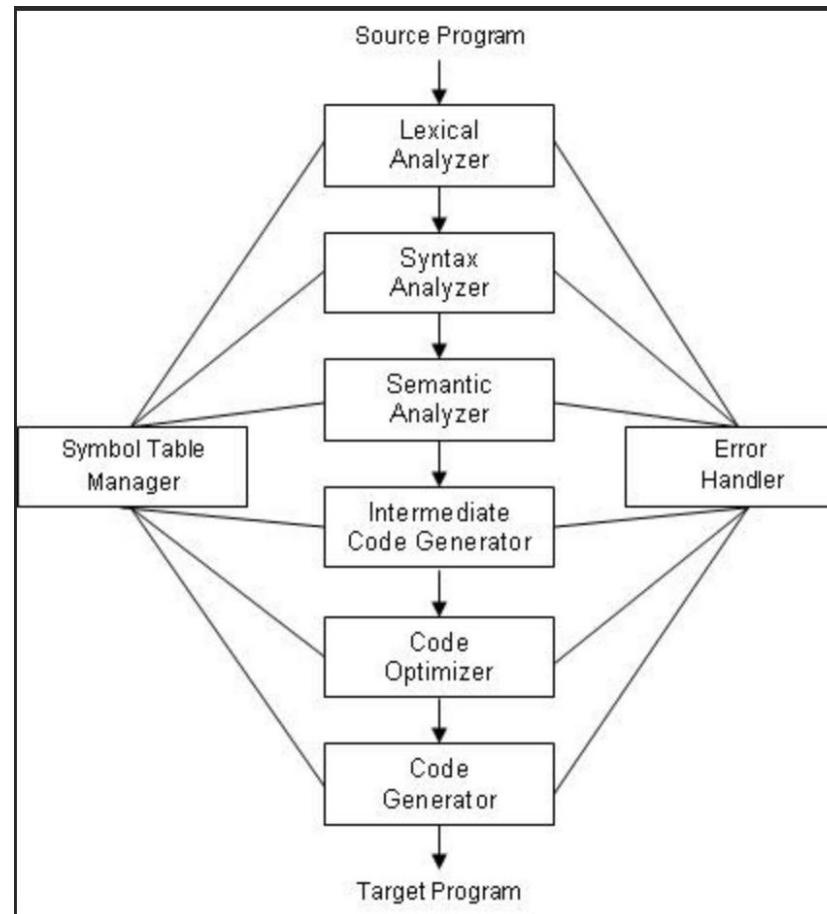
Interpreter

- 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

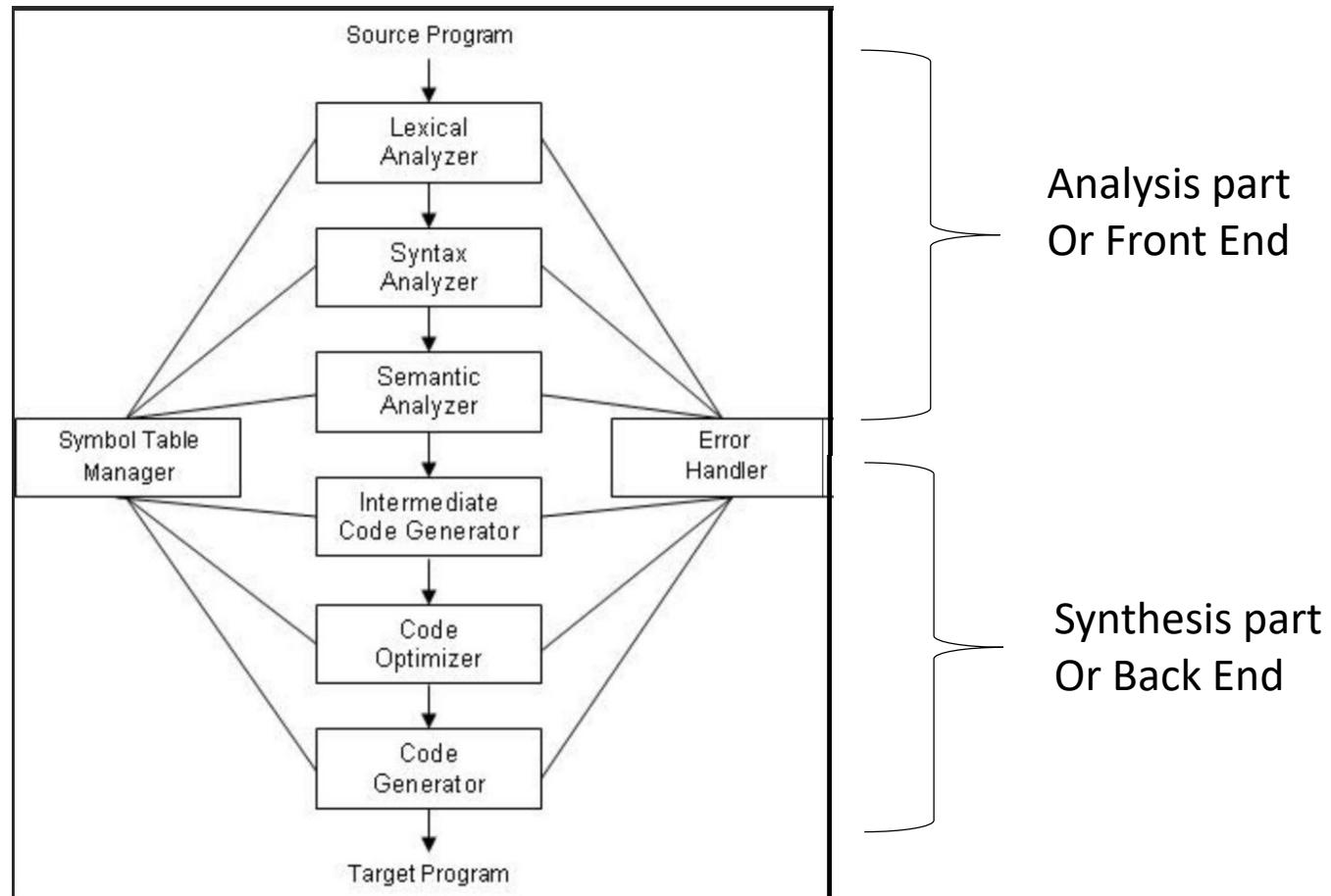


- The machine-language target program produced by a compiler is usually much faster than an interpreter at mapping inputs to outputs.
- An interpreter, however, can usually give better error diagnostics than a compiler, because it executes the source program statement by statement.

Phases of a Compiler

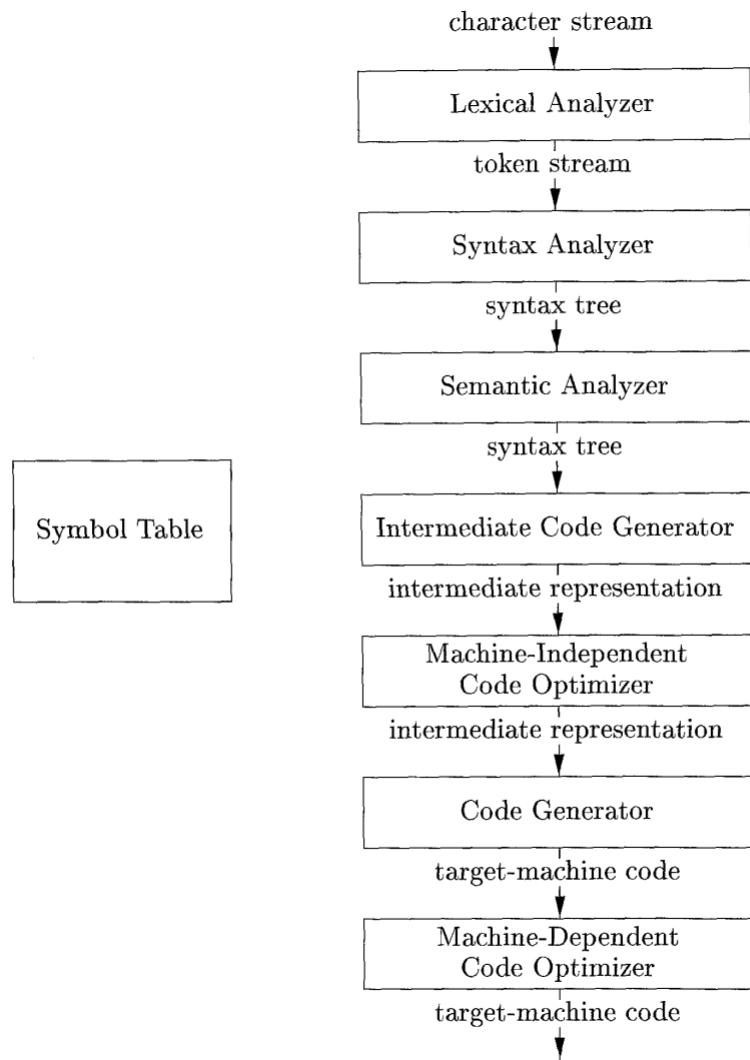


Phases of a Compiler



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

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

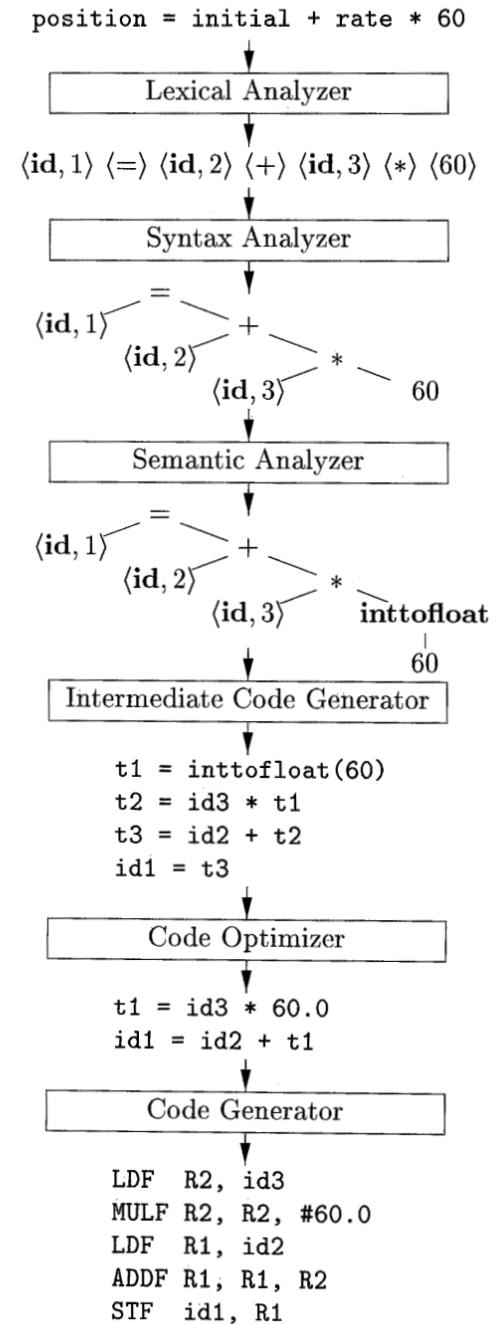


For example, suppose a source program contains the assignment

“position = initial + rate * 60”

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

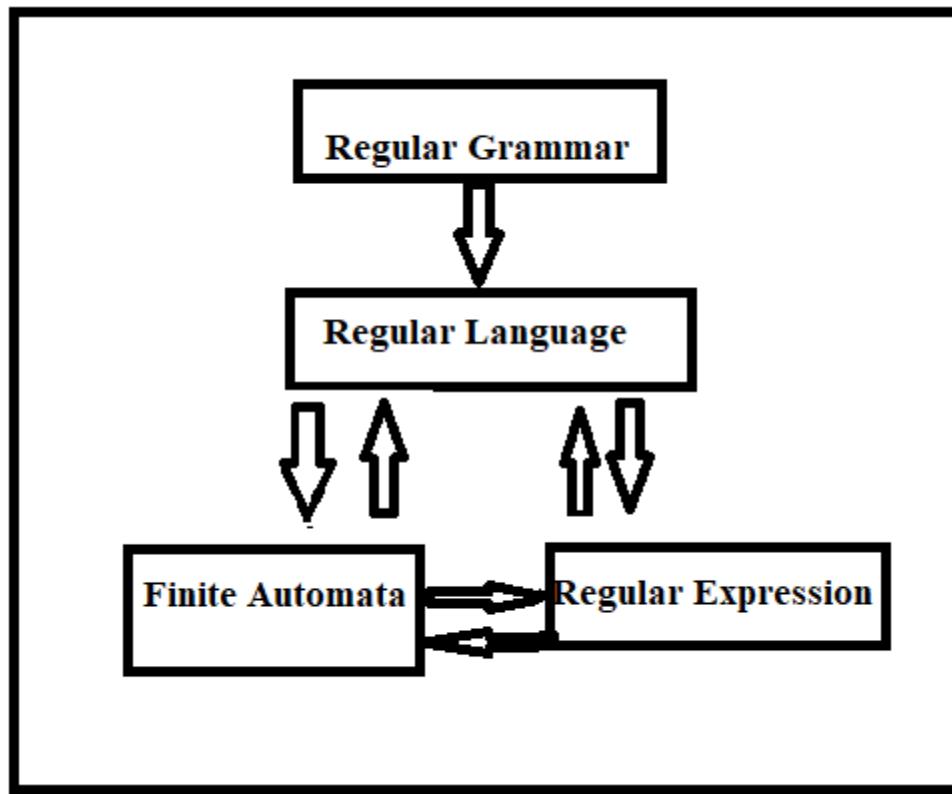
SYMBOL TABLE



Basics of ToC for Compiler Design

Content

- Finite Automata
 - Deterministic Finite Accepters
 - Examples
- Non Deterministic Finite Accepters
 - Examples
- Regular Expression
- Grammars
- Regular Languages



Finite Automata

1. Deterministic Finite Automata
2. Non-Deterministic Finite Automata

Deterministic Finite Accepters

Definition

A **deterministic finite accepter** or **dfa** is defined by the quintuple

$$M = (Q, \Sigma, \delta, q_0, F)$$

where Q is a finite set of **internal states**,

Σ is a finite set of symbols called the **input alphabet**,

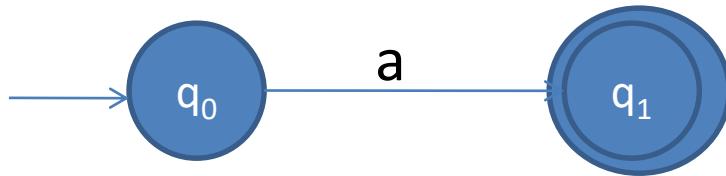
$\delta: Q \times \Sigma \rightarrow Q$ is a **total** function called the **transition function**,

$q_0 \in Q$ is the **initial state**,

$F \subseteq Q$ is a set of **final states**.

Transition Graph

- To visualize and represent finite automata, we use transition graphs, in which vertices represents states and the edges represents transitions.
- The labels on the vertices are the name of the states, while the labels on the edges are the current values of the input symbol.



- The initial state is identified by an incoming unlabeled arrow not originated at any vertex.
- Final states are drawn with a double circle.

Example

Transition Graph of a dfa $M = (Q, \Sigma, \delta, q_0, F)$

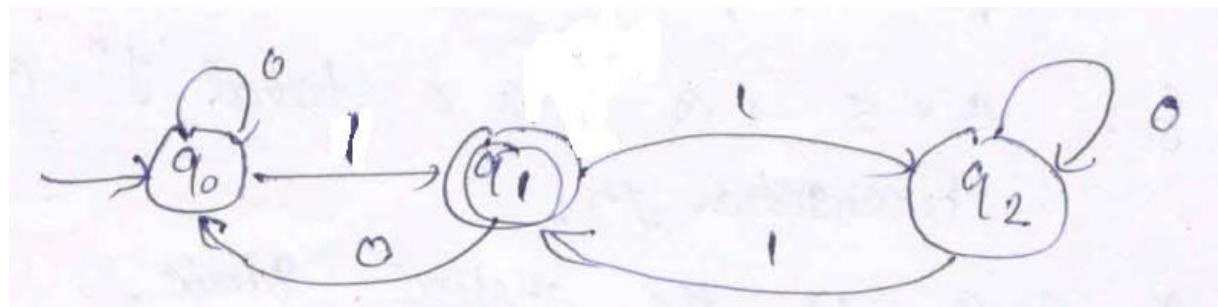
Vertex labeled with q_i : state $q_i \in Q$,

Edge from q_i to q_j labeled with a : transition $\delta(q_i, a) = q_j$.

Example .1 $M = (\{q_0, q_1, q_3\}, \{0,1\}, \delta, q_0, \{q_1\})$, where δ is given by

$$\delta(q_0, 0) = q_0, \delta(q_0, 1) = q_1, \delta(q_1, 0) = q_0,$$

$$\delta(q_1, 1) = q_2, \delta(q_2, 0) = q_2, \delta(q_2, 1) = q_1$$



Transition Graph of a dfa $M = (Q, \Sigma, \delta, q_0, F)$

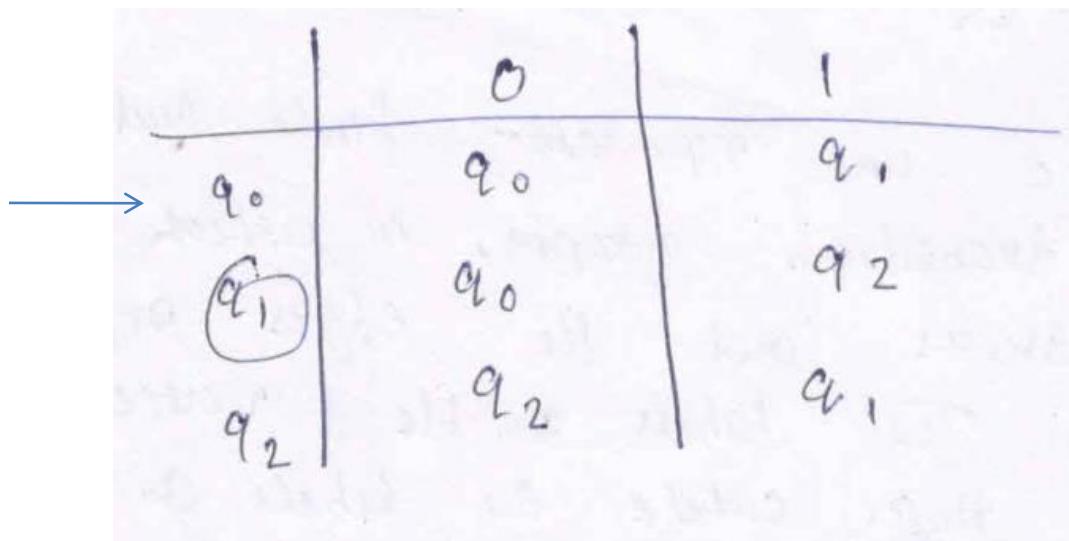
Vertex labeled with q_i : state $q_i \in Q$,

Edge from q_i to q_j labeled with a : transition $\delta(q_i, a) = q_j$.

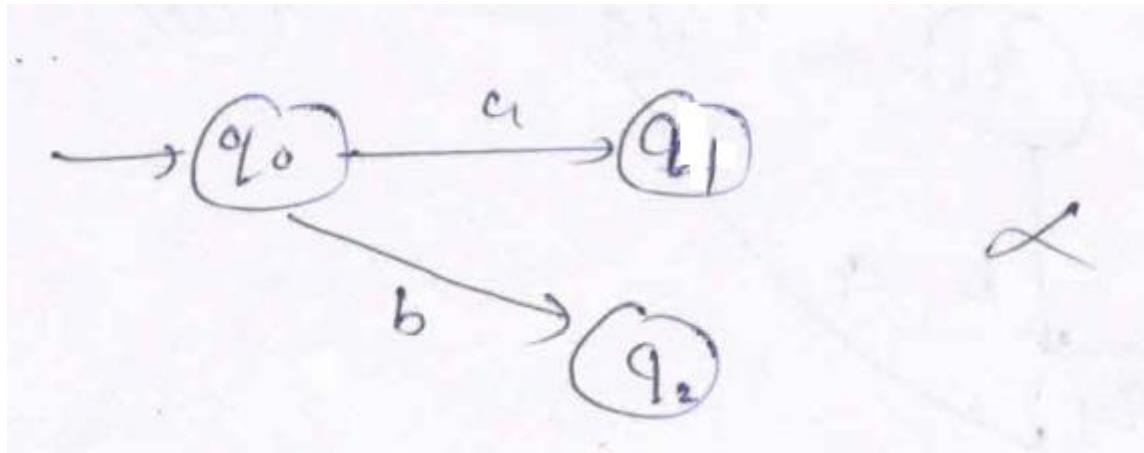
Example .1 $M = (\{q_0, q_1, q_3\}, \{0,1\}, \delta, q_0, \{q_1\})$, where δ is given by

$$\delta(q_0, 0) = q_0, \delta(q_0, 1) = q_1, \delta(q_1, 0) = q_0,$$

$$\delta(q_1, 1) = q_2, \delta(q_2, 0) = q_2, \delta(q_2, 1) = q_1$$

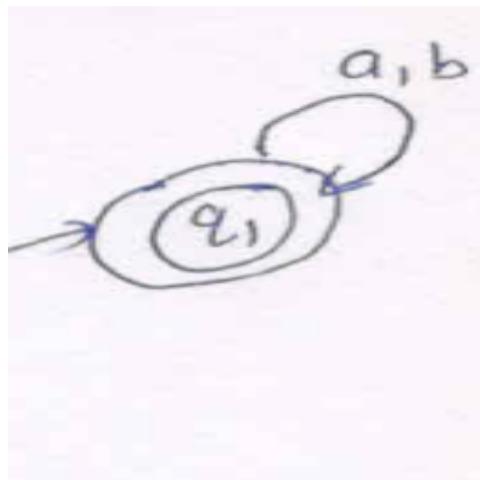


- Note: Machine should be complete



- The processed symbol is remembered by changing the state.

Number of final states

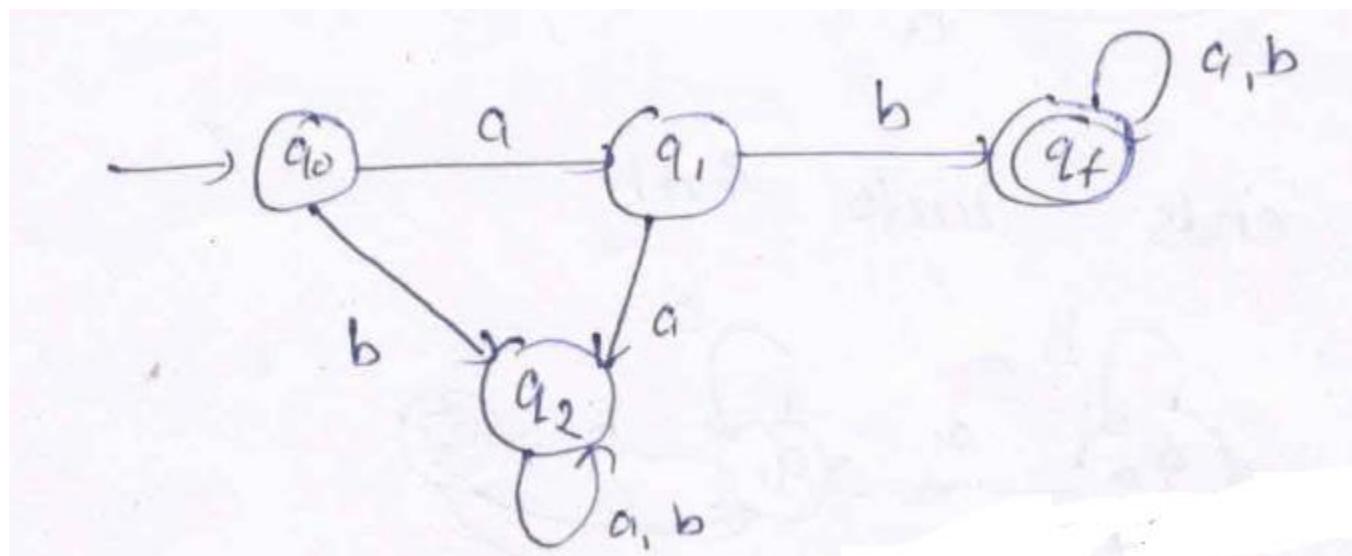


Examples

1. Draw a DFA which accepts all the strings on $\Sigma=\{a,b\}$ with the prefix 'ab'.

Input string

a		b		a		b		b		b
---	--	---	--	---	--	---	--	---	--	---



Nondeterministic Finite Accepters

A nondeterministic finite accepter or nfa is defined by the quintuple

$$M = (Q, \Sigma, \delta, q_0, F)$$

where Q is a finite set of internal states,

Σ is a finite set of symbols called the input alphabet,

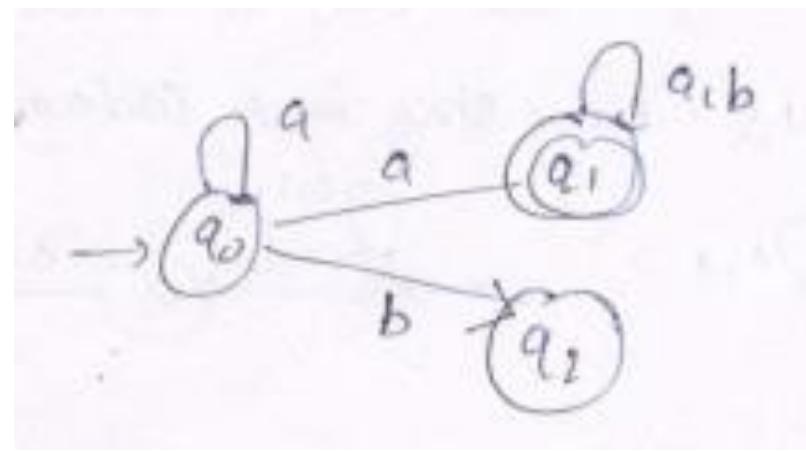
$q_0 \in Q$ is the initial state,

$F \subseteq Q$ is a set of final states.

$$\delta: Q \times (\Sigma \cup \{\lambda\}) \rightarrow 2^Q$$

\forall NFA \exists a DFA

\Rightarrow DFA \subseteq NFA



$$\delta(q_0, a) = \{q_0, q_1\}$$

$$\delta(q_2, b) = \{ \lambda \}$$

Transition graph

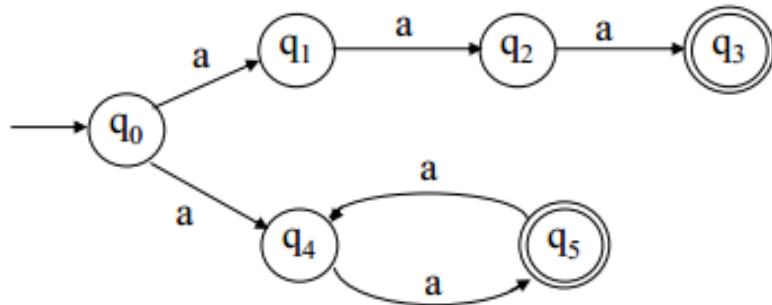
Transition Graph of an nfa $M = (Q, \Sigma, \delta, q_0, F)$

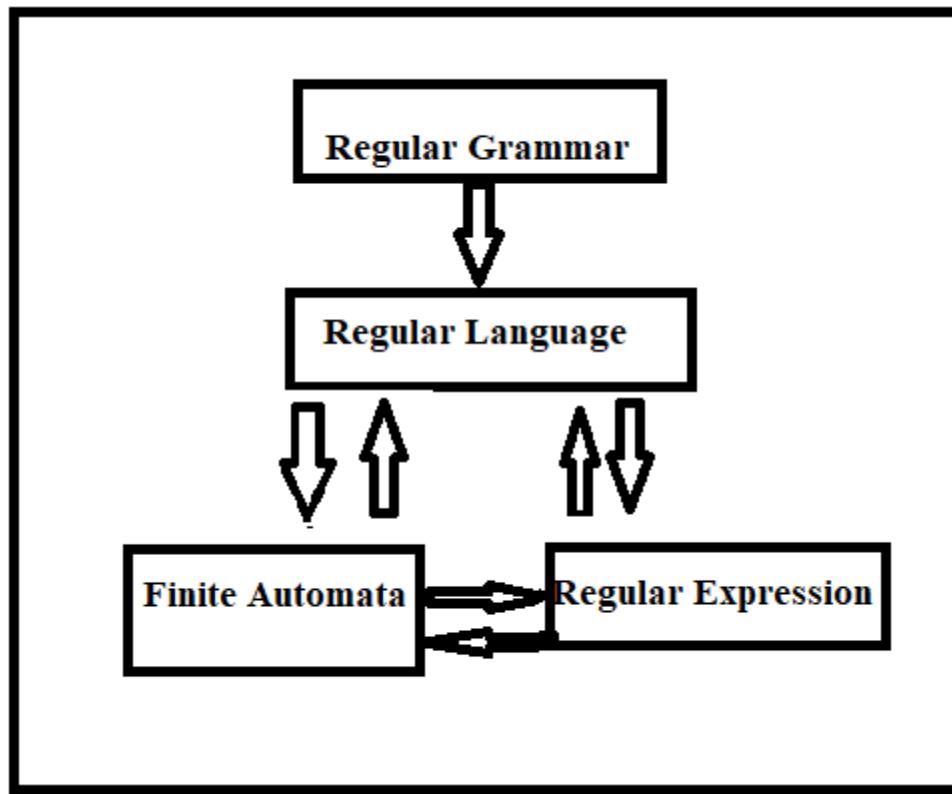
Vertex labeled with q_i : state $q_i \in Q$,

Edge from q_i to q_j labeled with a : $q_j \in \delta(q_i, a)$

Example

An nfa is shown as below





Formal definition of Regular Expression

Let Σ be a given input alphabet, then

1. λ and $a \in \Sigma$ are all regular expression, called primitive Regular Expression (R.E).
2. If r_1 and r_2 are R. E. then $(r_1 + r_2)$, $(r_1 . r_2)$, r_1^* and (r_1) are also R.E.
3. A string is a R.E. if we can derive it from the primitive R.E. by a finite number of application of the rule 2.
4. Order of operator, $()$, $*$, $.$, $+$

Regular Grammar (RG)

A third way to describe Regular Language (RL) is by means of RG.

- Right Linear Grammar
- Left Linear Grammar

Right Linear Grammar

A Grammar $G = (V, T, S, P)$ is said to be Right Linear if all productions are of the form

$$A \rightarrow nB$$

or

$$A \rightarrow n$$

where, A & $B \in V$ and $n \in T^*$

Left Linear Grammar

A Grammar $G = (V, T, S, P)$ is said to be Left Linear if all productions are of the form

$$A \rightarrow Bn$$

or $A \rightarrow n$

where, A & $B \in V$ and $n \in T^*$

cont...

- A Regular Grammar is either right linear or left linear

Note:

- ❖ In RG at most one variable appear on the R.H.S. of any production and that variable must be either right most or left most in the production.
- ❖ A language L is called regular if there is a finite automata.

Example

$$G_1 = (\{S\}, \{a, b\}, S, P_1)$$

P_1 given as

$$S \rightarrow abS \mid a \quad \text{is right linear.}$$

$$G_2 = (\{S, S_1, S_2\}, \{a, b\}, S, P_2)$$

P_2 is given as

$$\begin{aligned} S &\rightarrow S_1ab \\ S_1 &\rightarrow S_1ab \mid S_2 \end{aligned}$$

$S_2 \rightarrow a$ is left linear.

Both G_1 & G_2 are RG.

Cont...

➤ For G_1

$$S \rightarrow abS \rightarrow ababS \rightarrow aaaba$$

is a derivation

$$L(G_1) = L((ab)^*a)$$

➤ For G_2

$$S \rightarrow S_1 ab \rightarrow S_1 abab \rightarrow S_2 abab \rightarrow aabab$$

$$\text{R.E.} \rightarrow a(ab)^*$$

$$L(G_2) = L(a(ab)^*)$$

The $G = (\{S, A, B\}, \{a, b\}, S, P)$

with production

$$S \rightarrow A$$

$$A \rightarrow aB \mid \lambda$$

$$B \rightarrow Ab$$

is not regular, although the production are either right linear or left linear.

- The Grammar itself is neither right linear & left linear.
- This language is linear language

In linear language, at most one variable can occur on R.H.S. of productions without restriction on the position of this variable.

- ❖ Every RG is linear but not all linear grammar are regular.

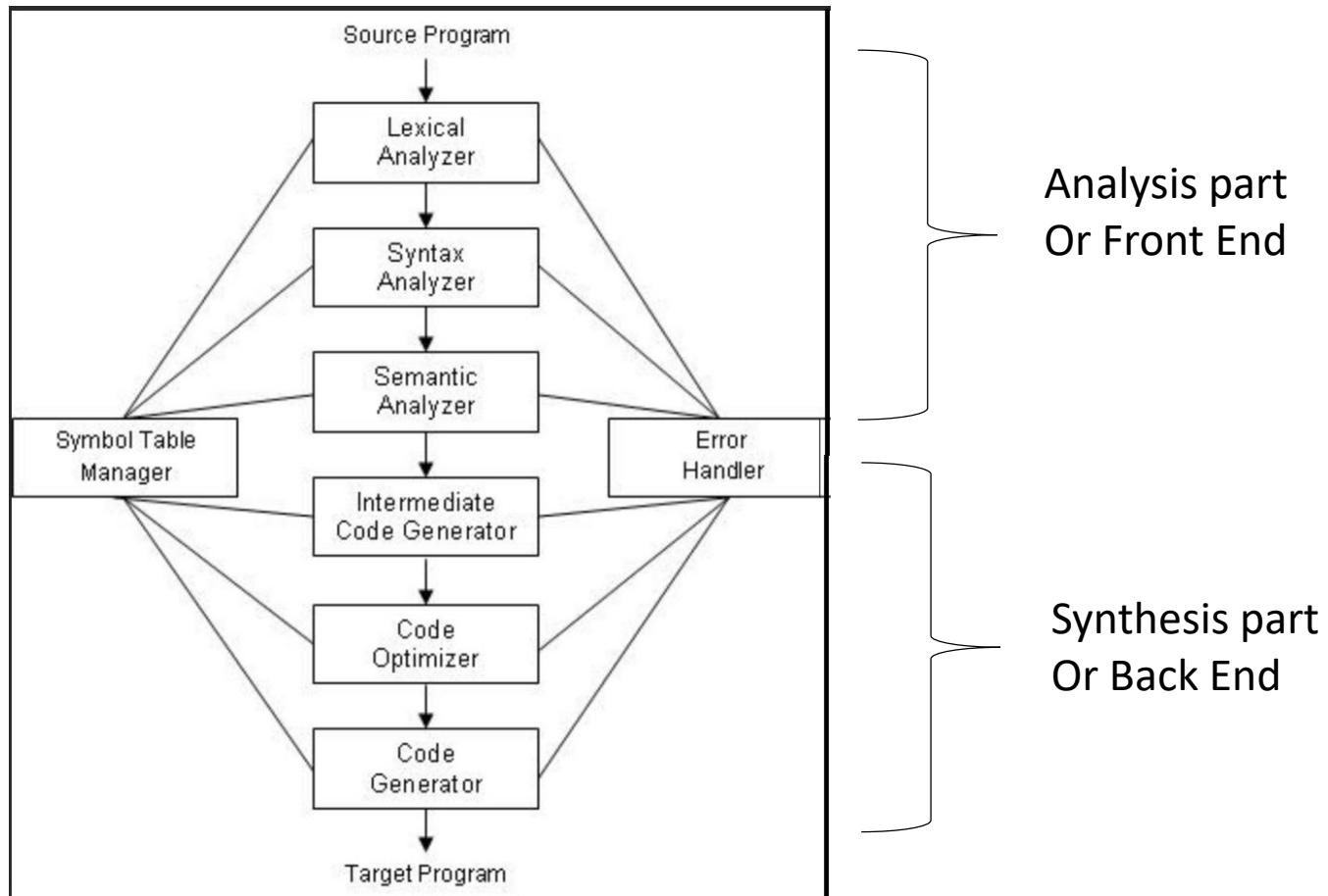
Suggested readings

1. An introduction to FORMAL LANGUAGES and AUTOMATA by PETER LINZ.
2. Introduction to Automata Theory, Languages, And Computation by JOHN E. HOPCROFT, RAJEEV MOTWANI, JEFFREY D. ULLMAN
3. Theory of computer science: automata, languages and computation by K.L.P MISHRA

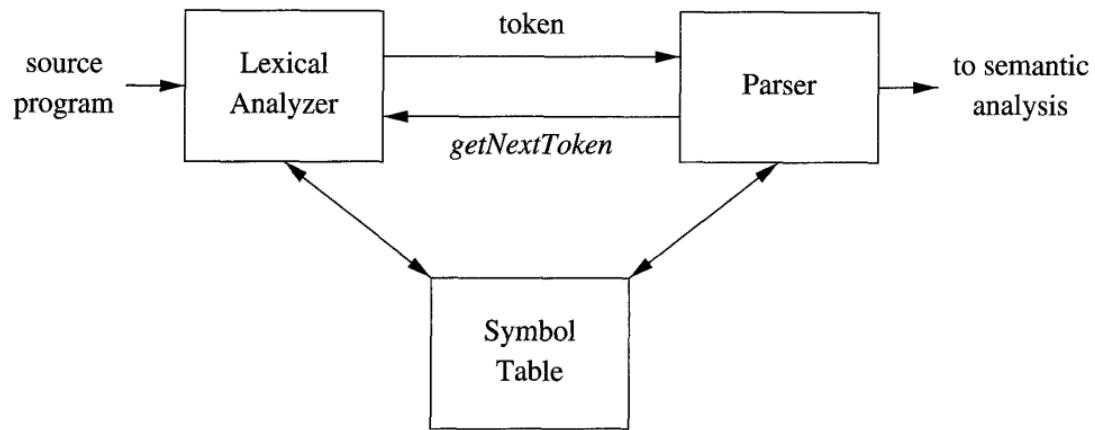


Compiler Design

Phases of a Compiler



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 lexical analyzer must know the **keyword**, **identifiers**, **operators**, **delimiters** and **punctuation symbols** of the language to be implemented.

- The lexical analyzer is the part of the compiler that reads the source text, it may perform certain other tasks besides identification of **lexemes**.
- One such task is stripping out comments and whitespace (**blank**, **newline**, **tab**, and perhaps **other characters** that are used to separate tokens in the input).

Example-1

```
int main()
{
/* find sum of two numbers */
int x=5, y=10, S;
S=x+y;
printf("sum is %d", S);
return 0;
}
```

Example-2

```
int main()
{
/* find sum of two numbers */
S=x+y;
int x=5, y=10, S;
printf("sum is %d", S);
return 0;
}
```

Example-3

```
int main()
{
    x=10, y=2, z;
    z= x+++y--+x;
    printf("the value of z is %d", z);
    return 0;
}
```

Tokens, Patterns, and Lexemes

Tokens

- 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, e.g., a particular keyword, or a sequence of input characters denoting an identifier.

Patterns

- 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.
- For identifiers and some other tokens, the pattern is a more complex structure that is matched by many strings.

Lexemes

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.

Example: int a=30;

	Lexeme	Tokens
1.	int	Keyword
2.	a	Identifier
3.	=	Operator
4.	20	Constant
5.	;	punctuation

Pattern: out of the given lexeme which one is keyword, identifier etc. is identified by some pattern.

* Regular expressions are commonly used to describe patterns

Formal definition of Regular Expression

Let Σ be a given input alphabet, then

1. λ and $a \in \Sigma$ are all regular expression, called primitive Regular Expression (R.E).
2. If r_1 and r_2 are R. E. then $(r_1 + r_2)$, $(r_1 \cdot r_2)$, r_1^* and (r_1) are also R.E.
3. A string is a R.E. if we can derive it from the primitive R.E. by a finite number of application of the rule 2.
4. Order of operator, $()$, $*$, $.$, $+$

C identifiers are strings of letters, digits, and underscores. Here is a regular definition for the language of C identifiers.

letter → A | B |.....| Z | a | b |.....| z | _

digit → 0 | 1 | 2 | 3....| 9

id → letter(letter | digit)*

Lexical Errors

It is hard for a lexical analyzer to tell, without the aid of other components, that there is a source-code error.

For instance, if the string “**fi**” is encountered for the first time in a C program in the context:

```
fi ( a == f(x)) ...
```

- Since **fi** is a valid lexeme for the **token id**, the lexical analyzer must return the **token id** to the **parser** and let some other phase of the compiler - probably the **parser** in this case - handle an error due to **transposition of the letters**.
- However, suppose a situation arises in which the lexical analyzer is unable to proceed because **none of the patterns for tokens matches any prefix of the remaining input**. The simplest recovery strategy is "**panic mode**" recovery.

"panic mode" recovery

- delete successive characters from the remaining input, until the lexical analyzer can find a well-formed token at the beginning of what input is left.
- This recovery technique may confuse the parser, but in an interactive computing environment it may be quite adequate.

Other possible error-recovery actions are:

1. Delete one character from the remaining input.

Eg. intt → int

2. Insert a missing character into the remaining input.

Eg. fr → for

while →while

3. Replace a character by another character.

Eg. wyile →while

4. Transpose two adjacent characters.

Eg. fro →for

Suggested Book

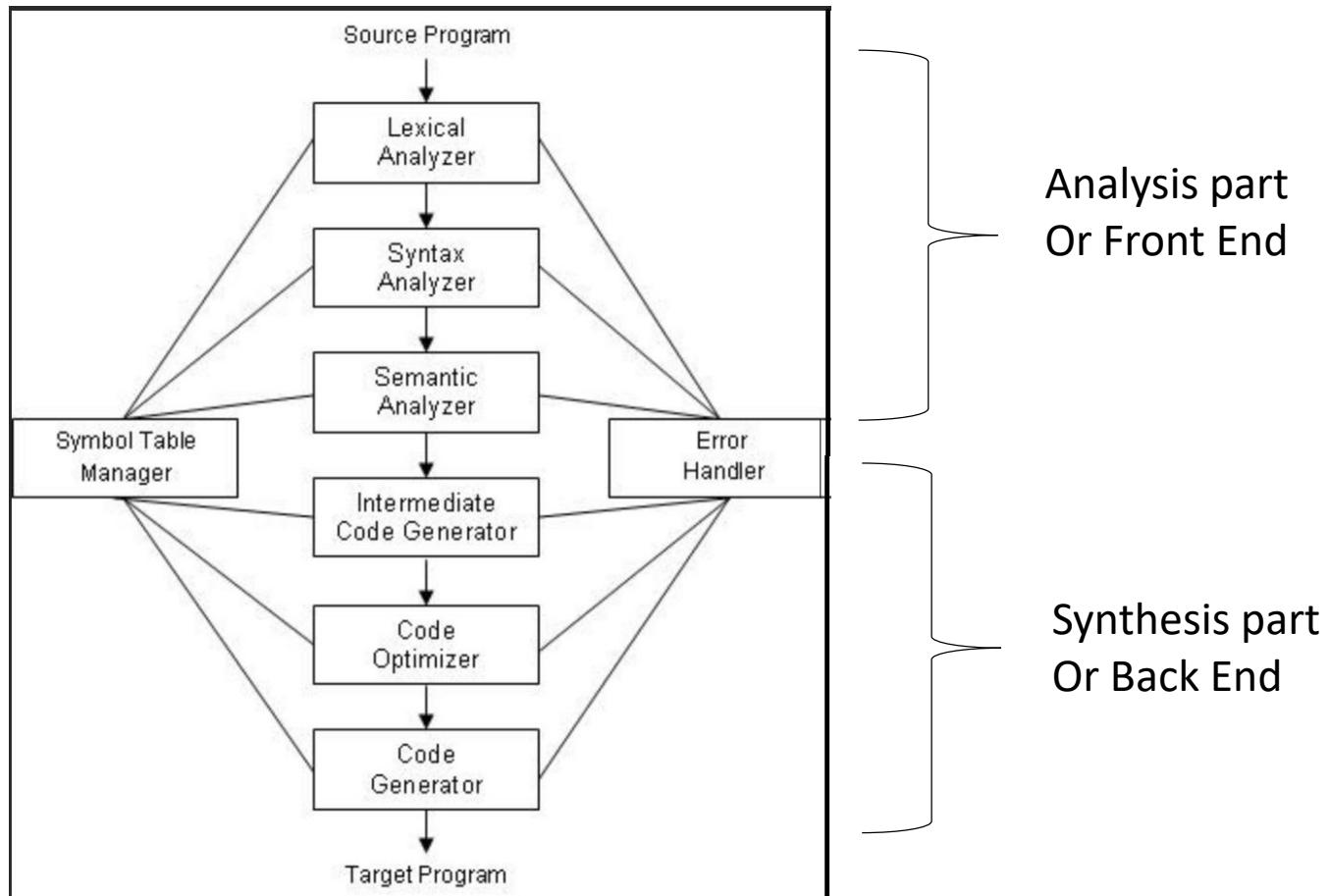
Compilers: Principles, Techniques, and Tools

by Aho, Sethi, Ullman and Lam

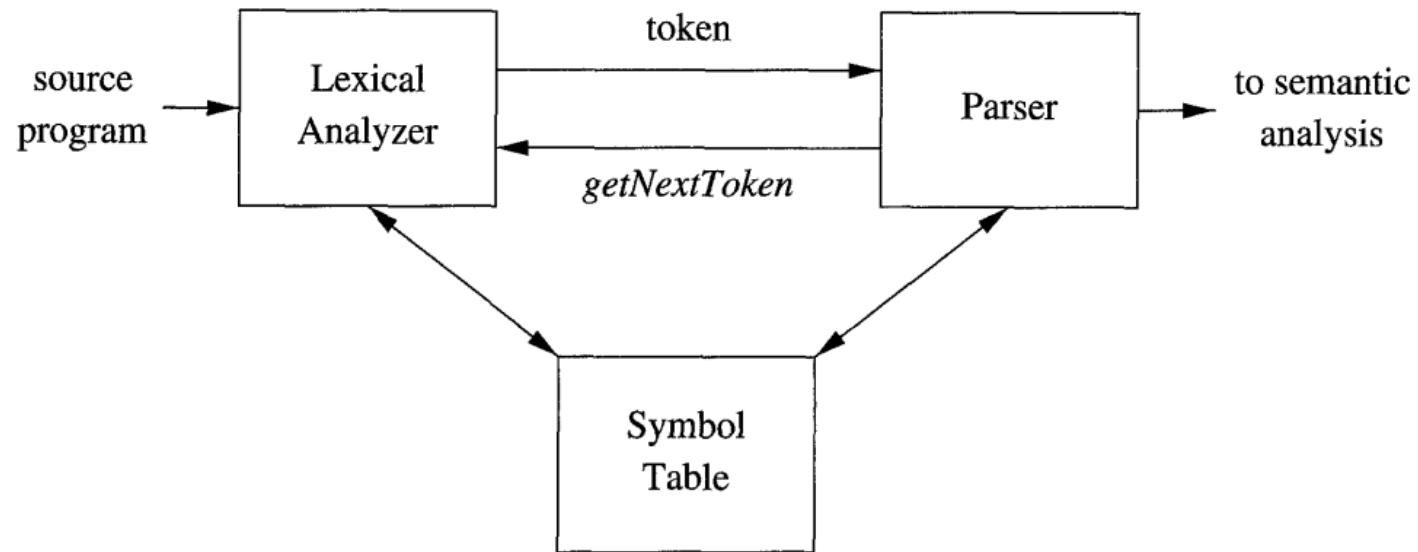


Compiler Design

Phases of a Compiler



Lexical Analyzer



Symbol Table

- Symbol tables are data structures that are used by compilers to hold information about source-program constructs.
- The information is **collected** incrementally by the **analysis phases** of a compiler and **used** by the **synthesis phases** to generate the target code.
- Entries in the symbol table contain information about an **identifier** such as its character string (or lexeme) , its **type**, **its position** in storage, and any other relevant information.
- Symbol tables typically need **to support multiple declarations** of the **same identifier** within a program.

Symbol Table (continue..)

Example-1

```
line1    main()
line2    {
3        char x[10];
4
5    .
6    .
7    .
8    z=y+2;
9    .
10   .
11   .
12   k=y+z;
}
```

Symbol Table			Associated Information		
Name	Data type	Size	Dimension	Line of declaration	Line of usage
x	char	10	1	3	-
y	int	4	0	4	8, 12

Tokens, Patterns, and Lexemes

Example: int a=30;

	Lexeme	Tokens
1.	int	Keyword
2.	a	Identifier
3.	=	Operator
4.	20	Constant
5.	;	punctuation

Pattern: out of the given lexeme which one is keyword, identifier etc. is identified by some pattern.

* Regular expressions are commonly used to describe patterns

How to describe tokens?

- Programming language tokens can be described by regular languages.
- Regular languages have been discussed in great detail in the “Theory of Computation” course.

Examples

- My email address
ankitavaish@bhu.ac.in

Here, $\Sigma = \text{letter} \cup \{@, .\}$

letter $\rightarrow a | b | \dots | z | A | B | \dots | Z$

name $\rightarrow \text{letter}^*$

address $\rightarrow \text{name '@' name '.' name '.' name}$

Try this

- My mobile number with country code

91-8565123486

Regular definition for the Unsigned number

Unsigned number in C: 2, 2.e0, 20.e-01, 2.000

digit → 0 | 1 | ... | 9

digits → digit⁺

fraction → ‘.’ digits | ε

exponent → (E (‘+’ | ‘-’ | ε) digits) | ε

number → digits fraction exponent

C identifiers are strings of letters, digits, and underscores. Here is a regular definition for the language of C identifiers.

letter → A | B |.....| Z | a | b |.....| z | _

digit → 0 | 1 | 2 | 3....| 9

id → letter(letter | digit)*

Transition Diagrams

- Regular expression are declarative specifications
- Transition diagram is an implementation

Deterministic Finite Acceptors

Definition

A **deterministic finite accepter** or **dfa** is defined by the quintuple

$$M = (Q, \Sigma, \delta, q_0, F)$$

where **Q** is a finite set of **internal states**,

Σ is a finite set of symbols called the **input alphabet**,

$\delta: Q \times \Sigma \rightarrow Q$ is a **total** function called the **transition function**,

$q_0 \in Q$ is the **initial state**,

$F \subseteq Q$ is a set of **final states**.

Nondeterministic Finite Accepters

A nondeterministic finite accepter or nfa is defined by the quintuple

$$M = (Q, \Sigma, \delta, q_0, F)$$

where Q is a finite set of internal states,

Σ is a finite set of symbols called the input alphabet,

$q_0 \in Q$ is the initial state,

$F \subseteq Q$ is a set of final states.

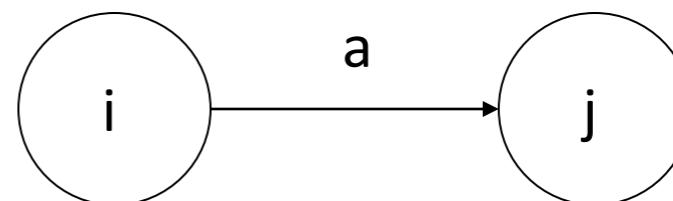
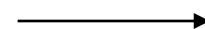
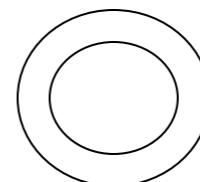
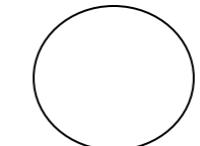
$$\delta: Q \times (\Sigma \cup \{\lambda\}) \rightarrow 2^Q$$

$\forall \text{ NFA } \exists \text{ a DFA}$

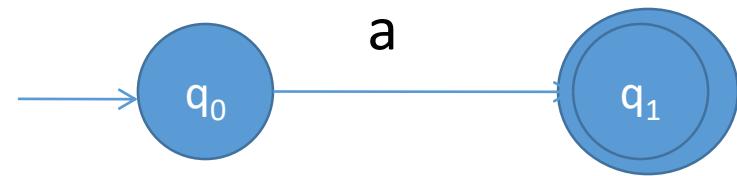
$\Rightarrow \text{DFA} \subseteq \text{NFA}$

Transition Graph

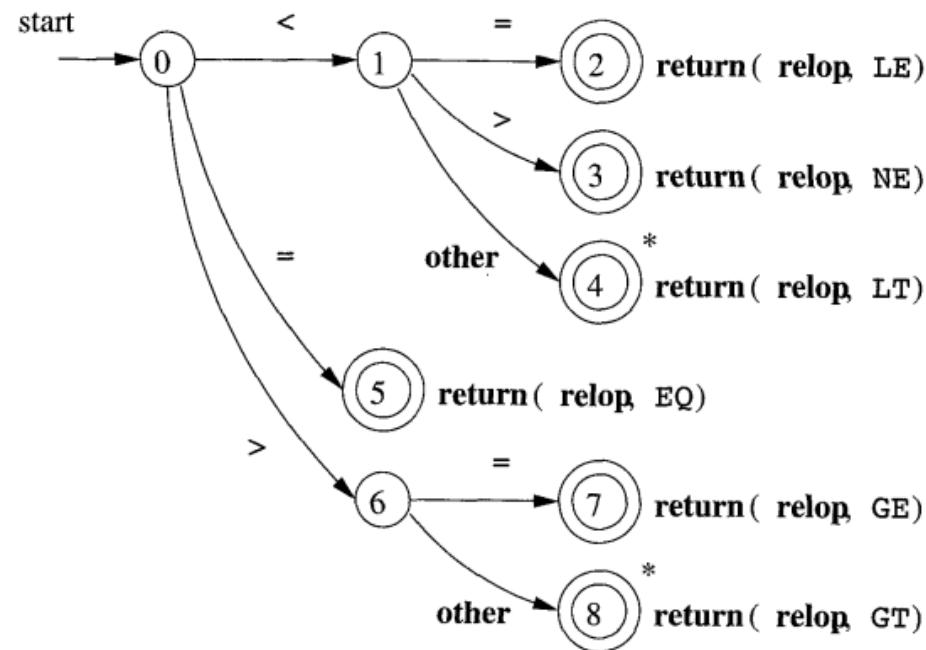
- A state
- A final state
- Transition
- Transition from state i to state j on an input a



- The initial state is identified by an incoming unlabeled arrow not originated at any vertex.

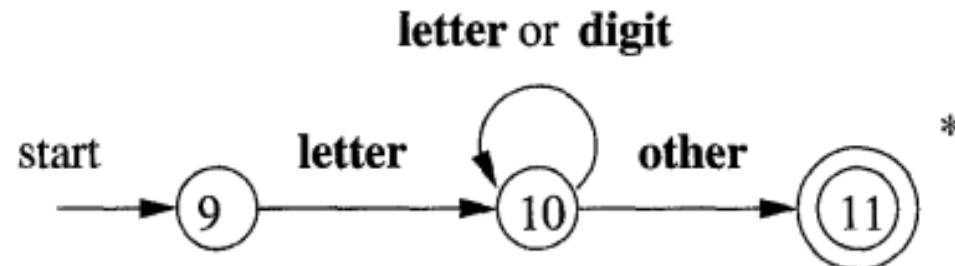


Transition diagram for “relop”



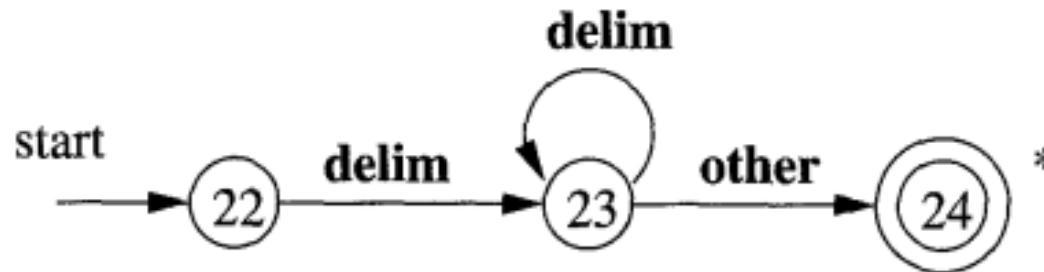
Recognition of Reserved Words and Identifiers

- Recognizing keywords and identifiers presents a problem.
- Usually, keywords like `if` or `then` are reserved, so they are not identifiers even though they look like identifiers.



- There are two ways that we can handle reserved words that look like identifiers:
 - Install the reserved words in the symbol table initially. A field of the symbol-table entry indicates that these strings are never ordinary identifiers, and tells which token they represent.
 - Create separate transition diagrams for each keyword.

Transition diagram for whitespace

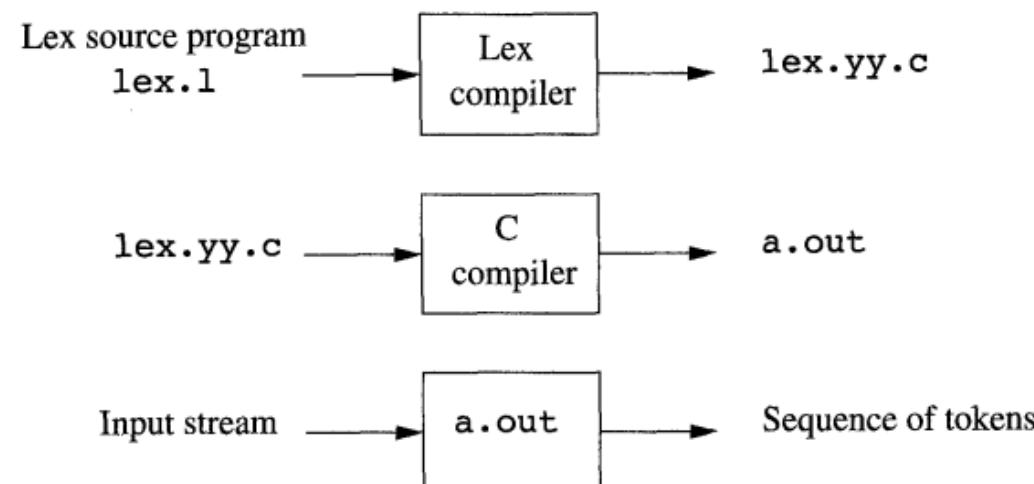


- Note that in state 24, we have found a block of consecutive whitespace characters, followed by a non-whitespace character.
- We retract the input to begin at the nonwhitespace, but we do not return to the parser.
- Rather, we must restart the process of lexical analysis after the whitespace

Draw transition diagram for unsigned numbers

The Lexical-Analyzer Generator Lex

- Lex is a tool that allows one to specify a lexical analyzer by specifying regular expressions to describe patterns for tokens.
- The input notation for the Lex tool is referred to as the Lex language and the tool itself is the Lex compiler.
- Behind the scenes, the Lex compiler transforms the input patterns into a transition diagram and generates code, in a file called `lex.yy.c`, that simulates this transition diagram.



How does LEX work?

- Regular expressions describe the languages that can be recognized by finite automata.
- Translate each token regular expression into a non deterministic finite automaton (NFA)
- Convert the NFA into an equivalent DFA.
- Minimize the DFA to reduce number of states

Suggested Book

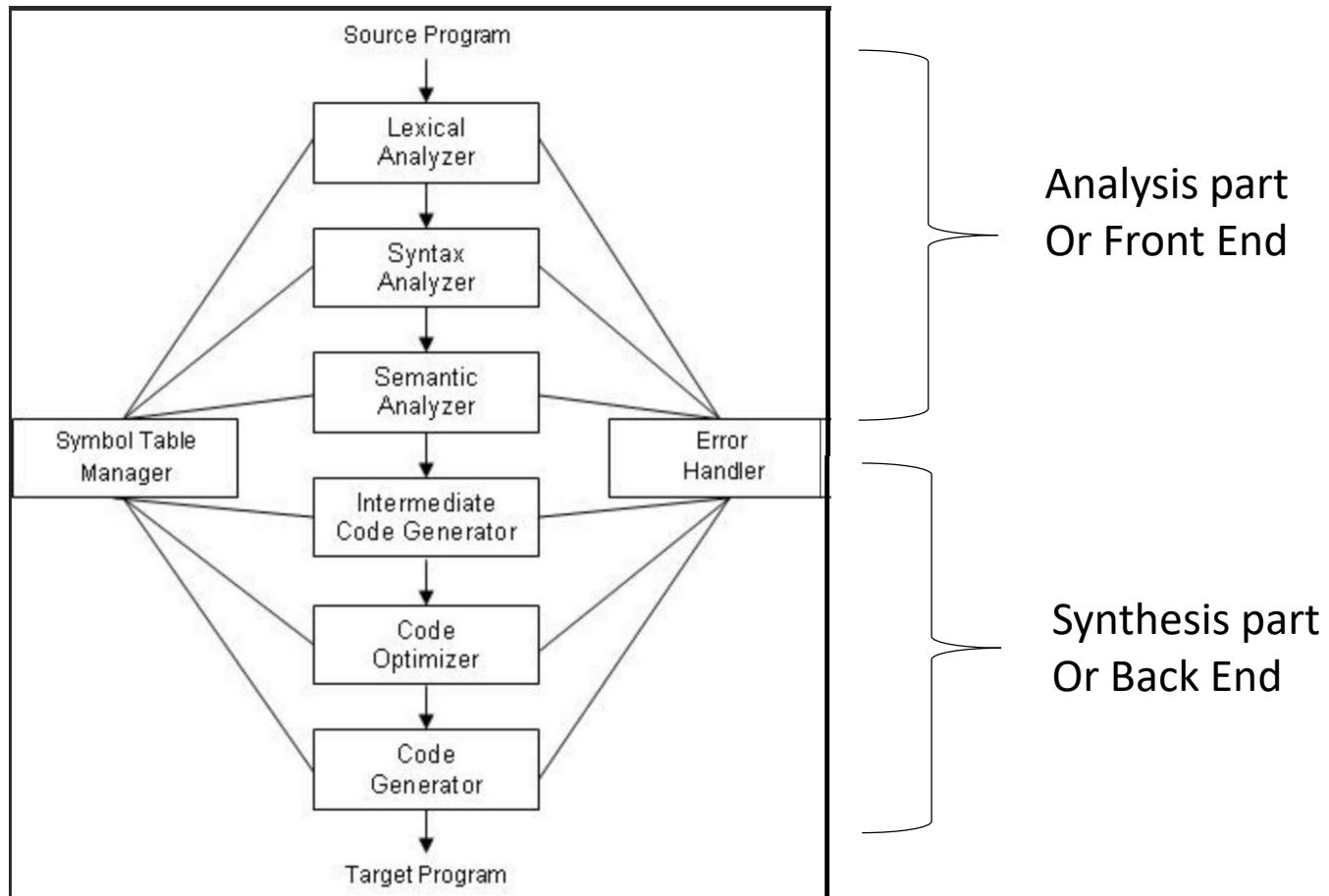
Compilers: Principles, Techniques, and Tools

by Aho, Sethi, Ullman and Lam

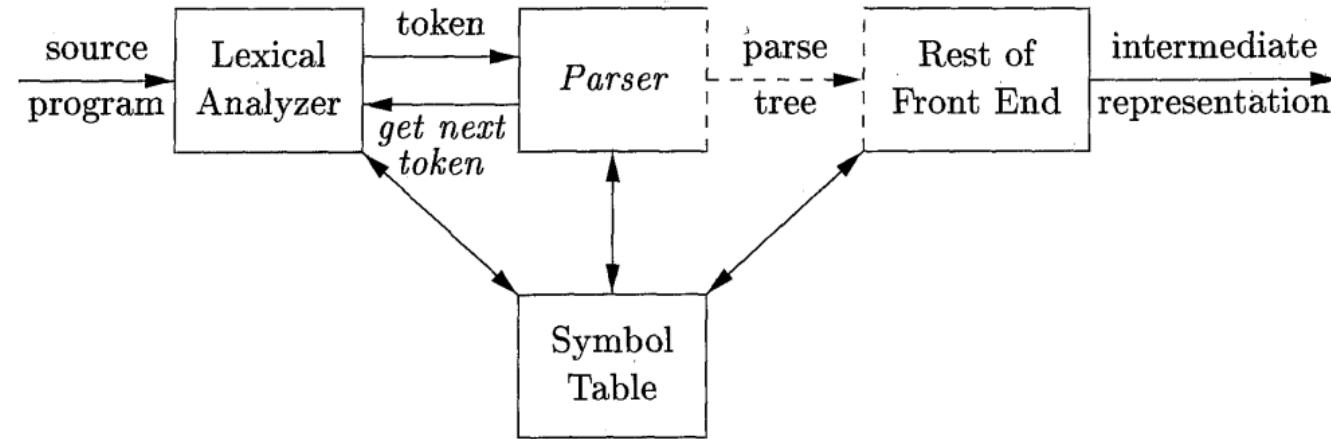


Compiler Design

Phases of a Compiler



Syntax Analyzer



- Check syntax and construct parse or syntax tree.
- Error reporting and recovery.
- Model using context free grammars
- Recognize using Push down automata/Table Driven Parsers

Limitations of regular language

- How to describe language syntax precisely and conveniently. Can regular expressions be used?
- Many languages are not regular, for example, string of balanced parentheses
 - $((((\dots))))$
 - $\{ (')^i \mid i \geq 0 \}$
 - There is no regular expression for this language.
- A finite automata may repeat states, however, it cannot remember the number of times it has been to a particular state.
- A more powerful language is needed to describe a valid string of tokens.

Syntax definition

Context free grammars (V, T, S, P)

V : a set of non terminal symbols

T : a set of terminal symbols

S : a start symbol

P : a set of productions of the form

nonterminal \rightarrow String of terminals & non terminals

Or $V \rightarrow (V \cup T)^*$

- A grammar derives strings by beginning with a start symbol and repeatedly replacing a non terminal by the right hand side of a production for that non terminal.
- The strings that can be derived from the start symbol of a grammar G , form the language $L(G)$ defined by the grammar.

Examples

- Consider the following grammar:

$$S \rightarrow (S) S \mid \epsilon$$

String of balanced parentheses

Syntax analyzer

- Testing for membership whether w belongs to $L(G)$ is just a “yes” or “no” answer
- However the syntax analyzer
 - Must generate the parse tree
 - Handle errors gracefully if string is not in the language

What syntax analysis cannot do!

- To check whether variables are of types on which operations are allowed.
- To check whether a variable has been declared before use.
- To check whether a variable has been initialized.
- These issues will be handled in semantic analysis.

Derivation

- Consider another Grammar and derive the string $9 - 5 + 2$

$\text{list} \rightarrow \text{list} + \text{digit} \mid \text{list} - \text{digit} \mid \text{digit}$

$\text{digit} \rightarrow 0 \mid 1 \mid \dots \mid 9$

$\text{list} \rightarrow \text{list} + \text{digit}$

$\rightarrow \text{list} - \text{digit} + \text{digit}$

$\rightarrow \text{digit} - \text{digit} + \text{digit}$

$\rightarrow 9 - \text{digit} + \text{digit}$

$\rightarrow 9 - 5 + \text{digit}$

$\rightarrow 9 - 5 + 2$

Therefore, the string $9-5+2$ belongs to the language specified by the grammar

Derivation

- If there is a production $A \rightarrow \alpha$ then we say that A derives α and is denoted by $A \Rightarrow \alpha$
- $\alpha A \beta \Rightarrow \alpha \gamma \beta$ if $A \rightarrow \gamma$ is a production.
- If $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \dots \Rightarrow \alpha_n$ then $\alpha_1 \Rightarrow \alpha_n$
- Given a grammar G and a string w of terminals in $L(G)$, we can write $S \Rightarrow w$.
- If $S \Rightarrow \alpha$ where α is a string of terminals and non terminals of G then we say that α is a **sentential form** of G.

Derivation

Left-most and right-most derivations are two important concepts in formal language theory and the study of context-free grammars, often used in the context of parsing and generating strings according to a given grammar.

Left-Most Derivation:

- In a left-most derivation, one starts with the start symbol of a grammar and repeatedly replace the leftmost non-terminal symbol in the current string with a production rule's right-hand side until you derive the desired string.

1. $S \rightarrow ABC$
2. $A \rightarrow a$
3. $B \rightarrow b$
4. $C \rightarrow c$

$S \rightarrow ABC$ (Replace S with ABC)
 $\rightarrow aBC$ (Replace A with a)
 $\rightarrow abC$ (Replace B with b)
 $\rightarrow abc$ (Replace C with c)

Right-Most Derivation:

- In a right-most derivation, you start with the start symbol of a grammar and repeatedly replace the rightmost non-terminal symbol in the current string with a production rule's right-hand side until you derive the desired string.

$S \rightarrow ABC$ (Replace S with ABC)

$\rightarrow ABc$ (Replace C with c)

$\rightarrow Abc$ (Replace B with b)

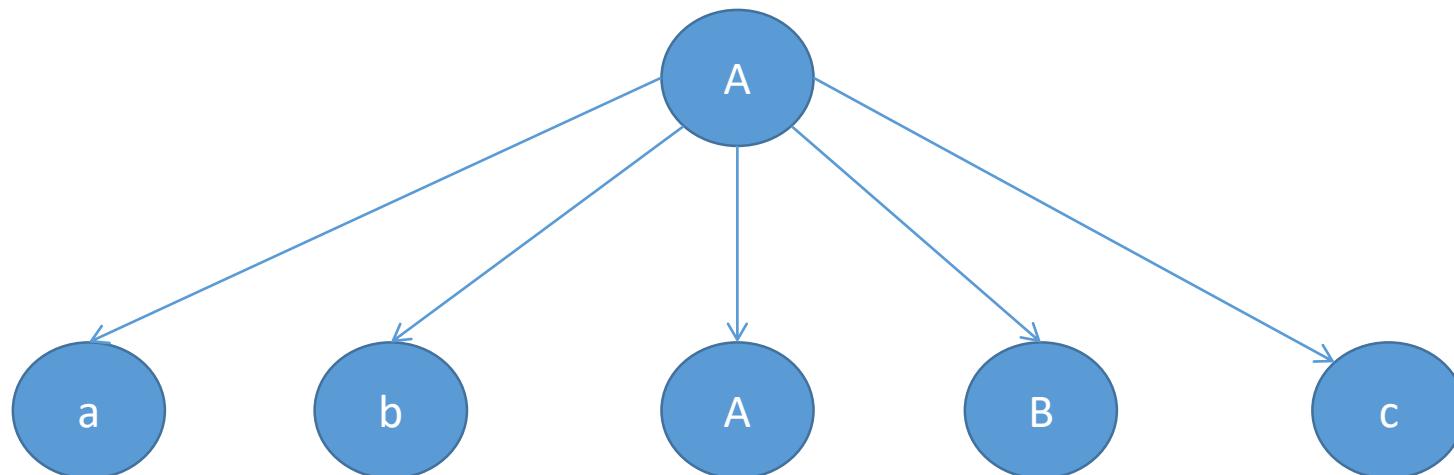
$\rightarrow abc$ (Replace A with a)

Note: An ambiguous grammar is one that produces more than one leftmost (rightmost) derivation of a sentence.

Parse/syntax or Derivation tree

It is an ordered tree in which nodes are labeled with the left sides of productions and in which the children of a node represent its corresponding right side.

$$A \rightarrow abABc$$



Definition of Derivation Tree

Let $G = (V, T, S, P)$ be a CFG. An ordered tree is a derivation tree for G iff it has the following properties:

1. The root is labeled S .
2. Every leaf has a label from $T \cup \{\lambda\}$.
3. Every interior vertex (a vertex i.e. not leaf) has a leaf.
4. If a vertex has a label $A \in V$ & its children are labeled (L to R) a_1, a_2, \dots, a_n then P must contain a production of form

$$A \rightarrow a_1 a_2 \dots a_n$$

5. A leaf labeled λ has no sibling (no children)

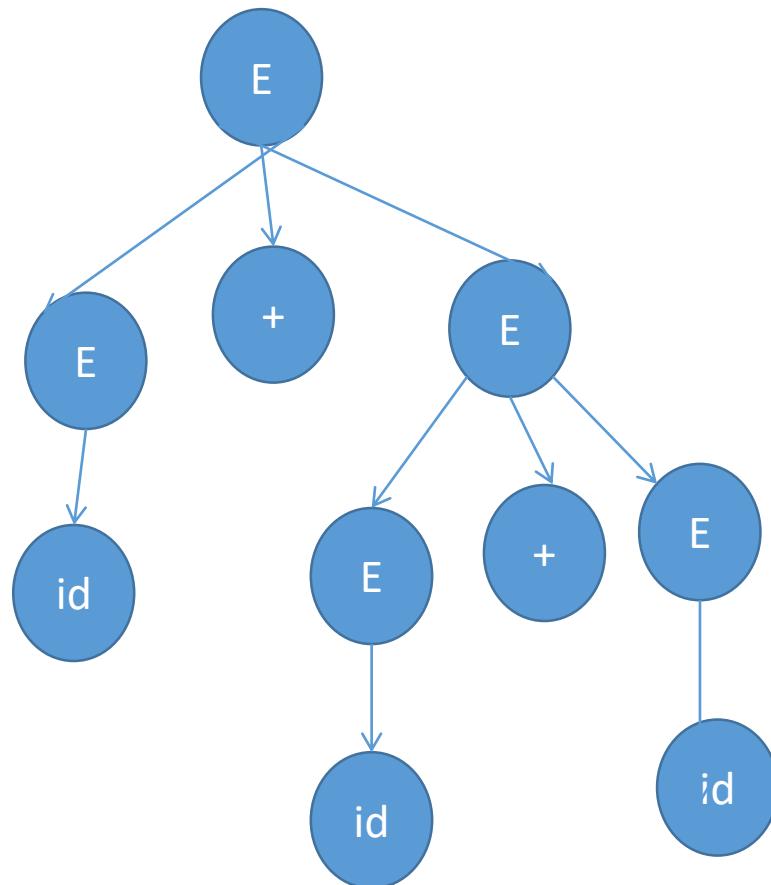
Note: A tree has properties 3, 4 & 5, but in which property 2 does not necessarily hold then property 2 can be replaced by 2(a).

2 (a) Every leaf has a label from $V \cup T \cup \{\lambda\}$ is said to be a partial derivation tree.

Example:

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

string $id + id + id$



Ambiguity

- A Grammar can have more than one parse tree for a string

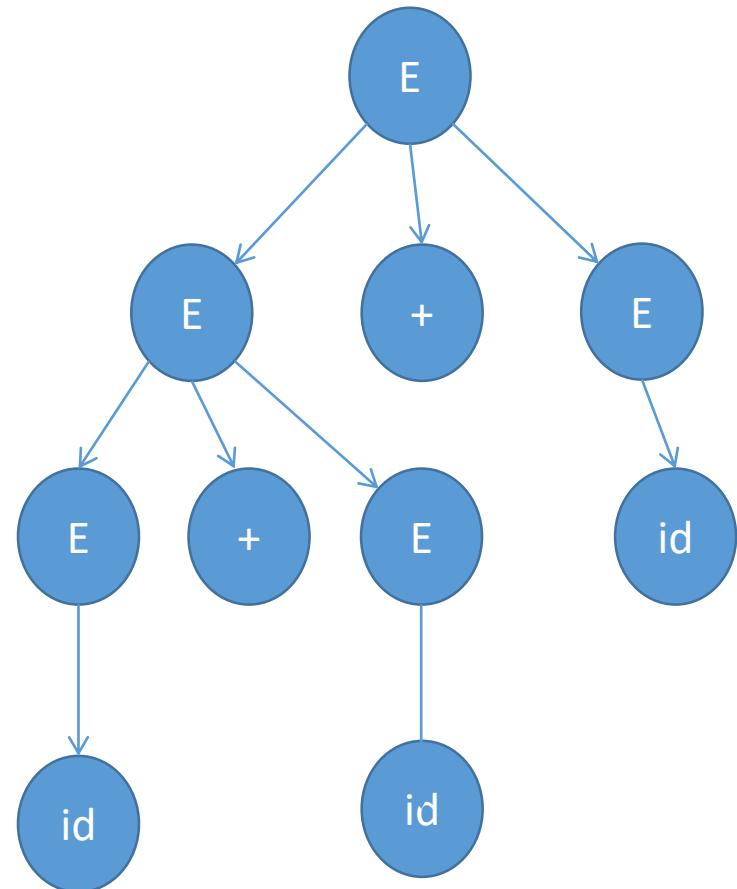
Consider the grammar G with productions

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

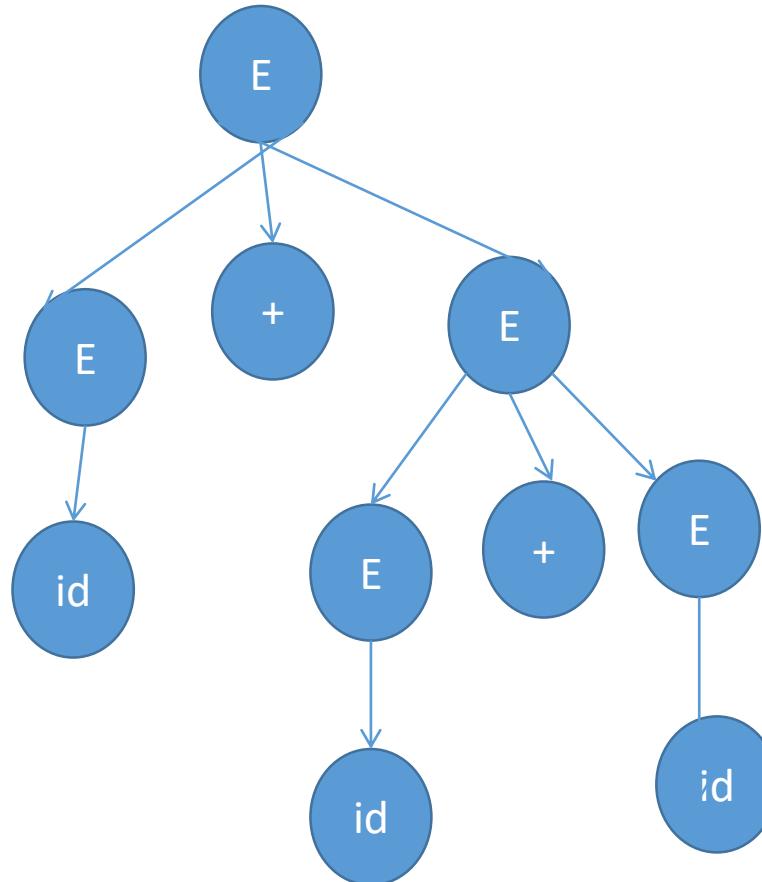
Consider grammar $E \rightarrow E + E \mid E * E \mid id$

Derive "id+id+id".

String "id+id+id" or 9+5+2 has two parse trees



(a)

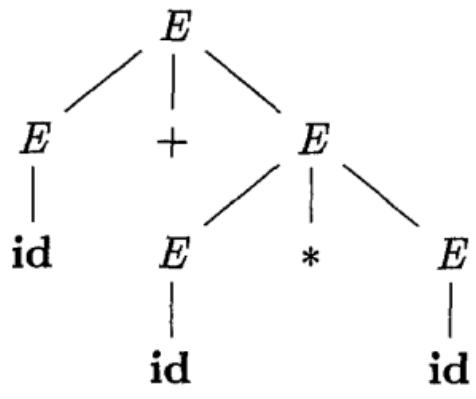


(b)

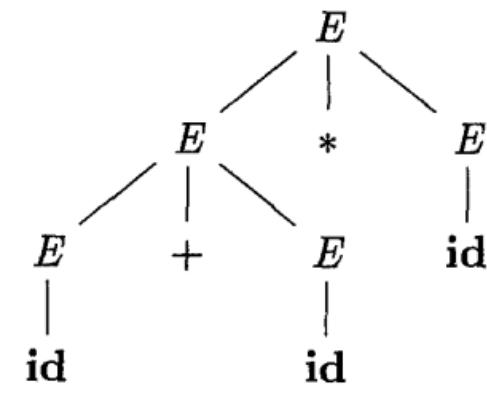
Id+id+id has two parse trees

Consider grammar $E \rightarrow E + E \mid E * E \mid id$

Derive “id+id*id”.



(a)



(b)

String “id+id*id” or 9+5*2 has two parse trees

Associativity

- If an operand has operator on both the sides, the side on which operator takes this operand is the associativity of that operator.
 - In $a+b+c$, b is taken by left +
 - +, -, *, / are left associative
 - ^, = are right associative

- Grammar to generate strings with right associative operators

$\text{right} \rightarrow \text{letter} = \text{right} \mid \text{letter}$

$\text{letter} \rightarrow a \mid b \mid \dots \mid z$

- Grammar to generate strings with left associative operators, if you want $=$ to be left associative

$\text{left} \rightarrow \text{left} = \text{letter} \mid \text{letter}$

Consider grammar $E \rightarrow E + E \mid id$

Derive “id+id+id”.

Make grammar left recursive

$E \rightarrow E + id \mid id$

Consider grammar $E \rightarrow E^{\wedge} E \mid id$

Derive “ $id^{\wedge}id^{\wedge}id$ ” or $2^{\wedge} 2^{\wedge} 3$.

Make grammar right recursive

$E \rightarrow id^{\wedge} E \mid id$

or $E \rightarrow F^{\wedge} E \mid F$,

$F \rightarrow id$

Suggested Book

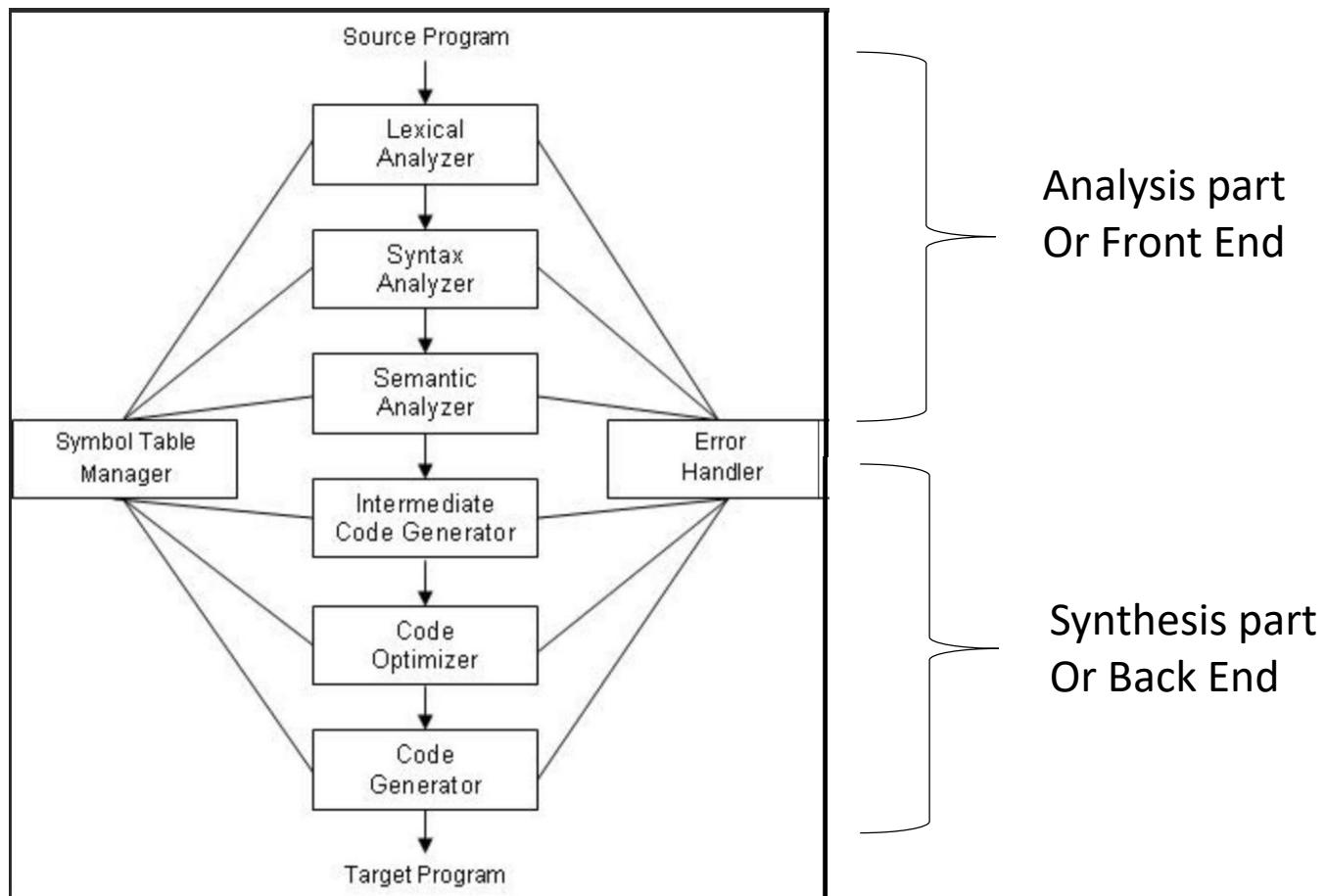
Compilers: Principles, Techniques, and Tools

by Aho, Sethi, Ullman and Lam

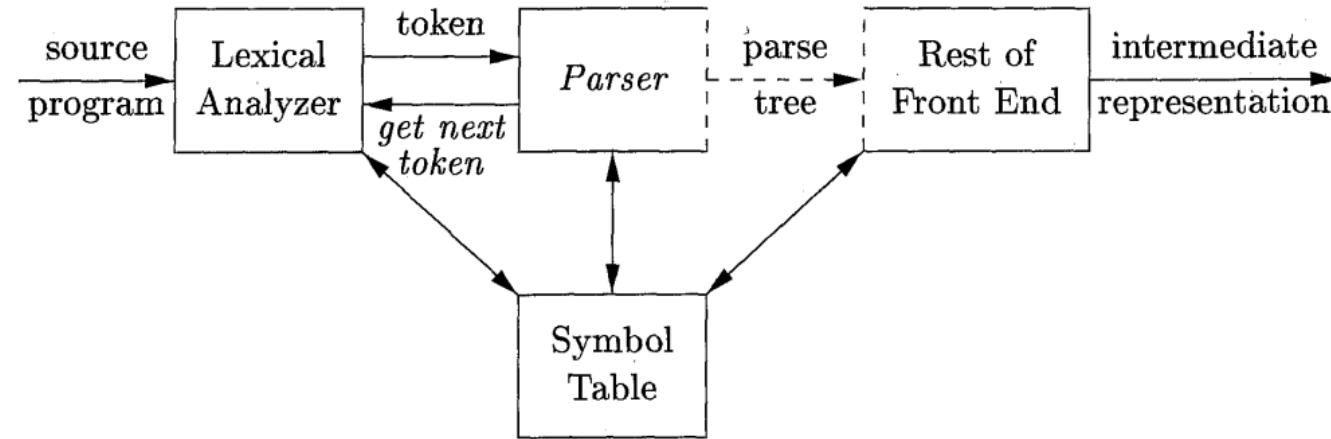


Compiler Design

Phases of a Compiler



Syntax Analyzer



- Check syntax and construct parse or syntax tree.
- Error reporting and recovery.
- Model using context free grammars
- Recognize using Push down automata/Table Driven Parsers

Syntax analyzer

- Testing for membership whether w belongs to $L(G)$ is just a “yes” or “no” answer
- However the syntax analyzer
 - Must generate the parse tree
 - Handle errors gracefully if string is not in the language

What syntax analysis cannot do!

- To check whether variables are of types on which operations are allowed.
- To check whether a variable has been declared before use.
- To check whether a variable has been initialized.

These issues will be handled in semantic analysis.

Derivation

Left-most and right-most derivations are two important concepts in formal language theory and the study of context-free grammars, often used in the context of parsing and generating strings according to a given grammar.

Ambiguity

- A Grammar can have more than one parse tree for a string

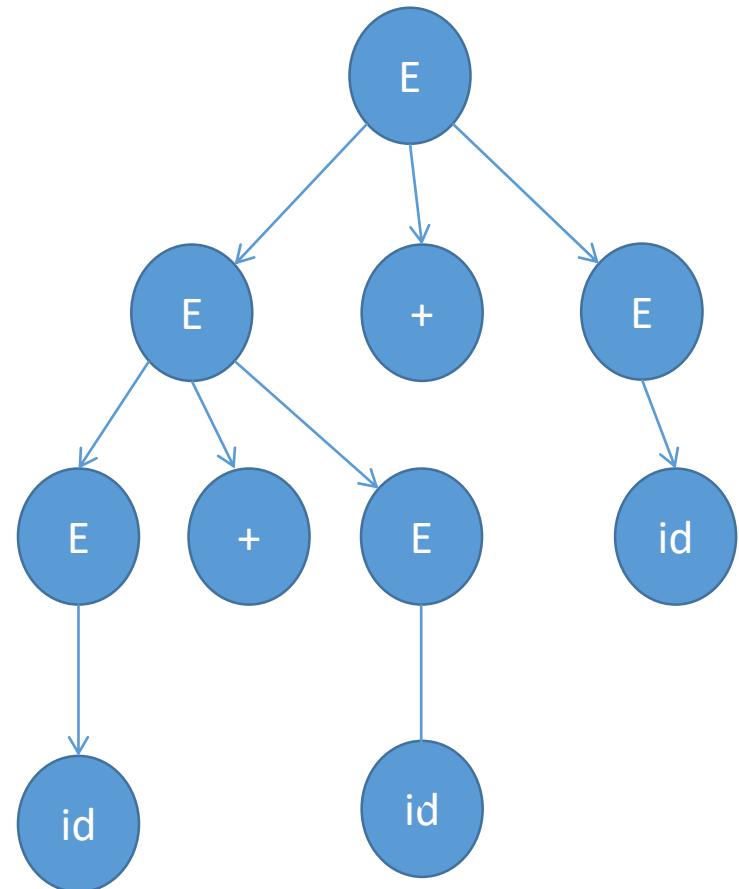
Consider the grammar G with productions

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

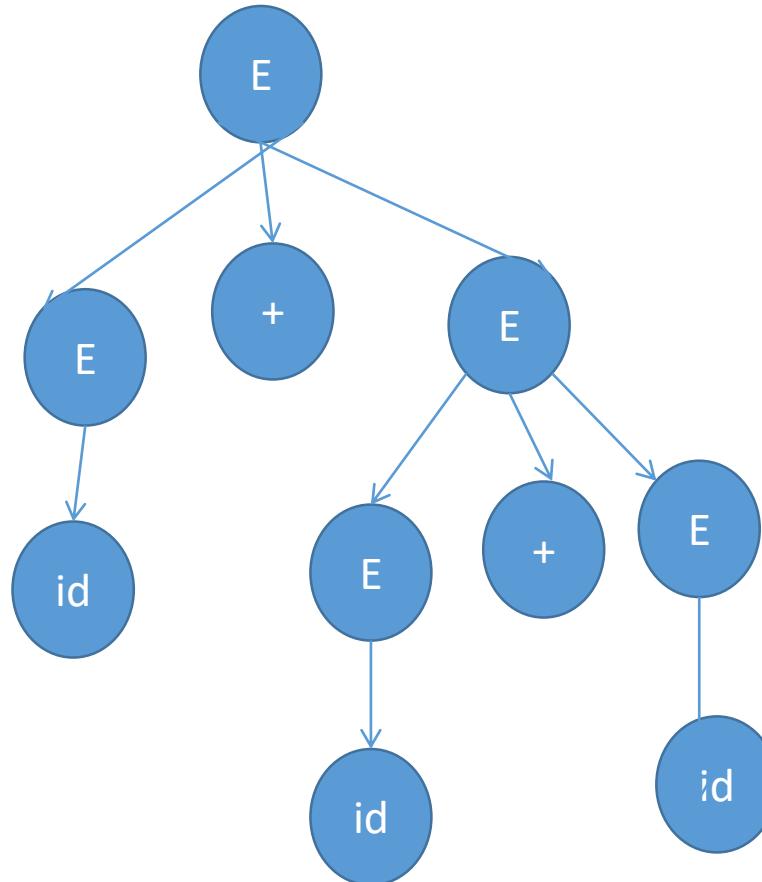
Consider grammar $E \rightarrow E + E \mid E * E \mid id$

Derive "id+id+id".

String "id+id+id" or 9+5+2 has two parse trees



(a)



(b)

Id+id+id has two parse trees

Associativity

- If an operand has operator on both the sides, the side on which operator takes this operand is the associativity of that operator.
 - In $a+b+c$, b is taken by left +
 - +, -, *, / are left associative
 - ^, = are right associative

Consider grammar $E \rightarrow E + E \mid id$

Derive “id+id+id”.

Make grammar left recursive

$E \rightarrow E + id \mid id$

Consider grammar $E \rightarrow E^{\wedge} E \mid id$

Derive “ $id^{\wedge}id^{\wedge}id$ ” or $2^{\wedge} 2^{\wedge} 3$.

Make grammar right recursive

$E \rightarrow id^{\wedge} E \mid id$

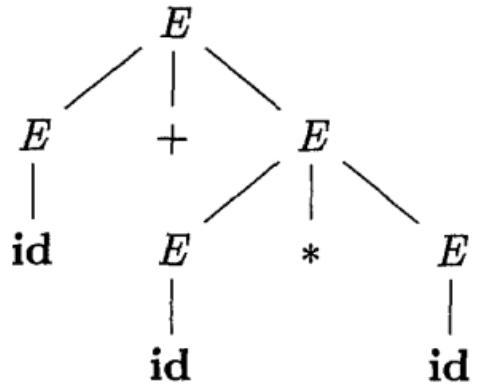
or $E \rightarrow F^{\wedge} E \mid F$,

$F \rightarrow id$

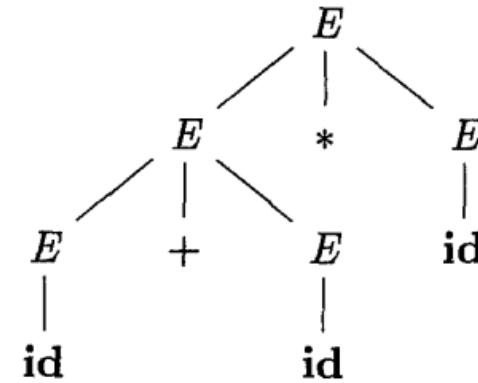
Precedence

Consider grammar $E \rightarrow E + E \mid E * E \mid id$

Derive “id+id*id”.



(a)



(b)

- String $id+id*id$ has two possible interpretations because of two different parse trees corresponding to $id+(id*id)$ and $(id+id)*id$.
- Precedence determines the correct interpretation.

Precedence in the Grammar for Arithmetic Expressions

- Ambiguous

$$E \rightarrow E + E$$
$$E \rightarrow E * E$$
$$E \rightarrow id$$

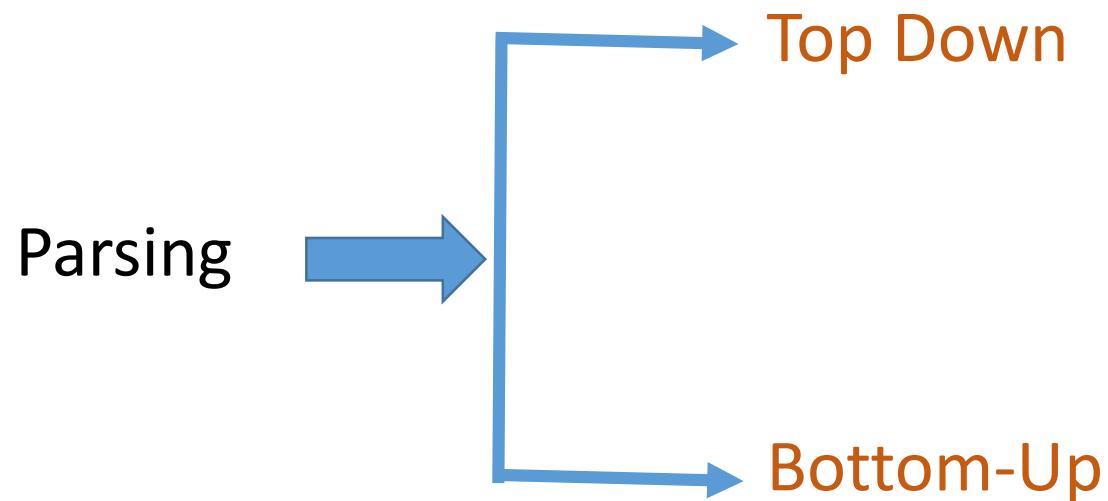
Associativity and precedence rules honored

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

Note: highest precedence operator should be kept on lower level.

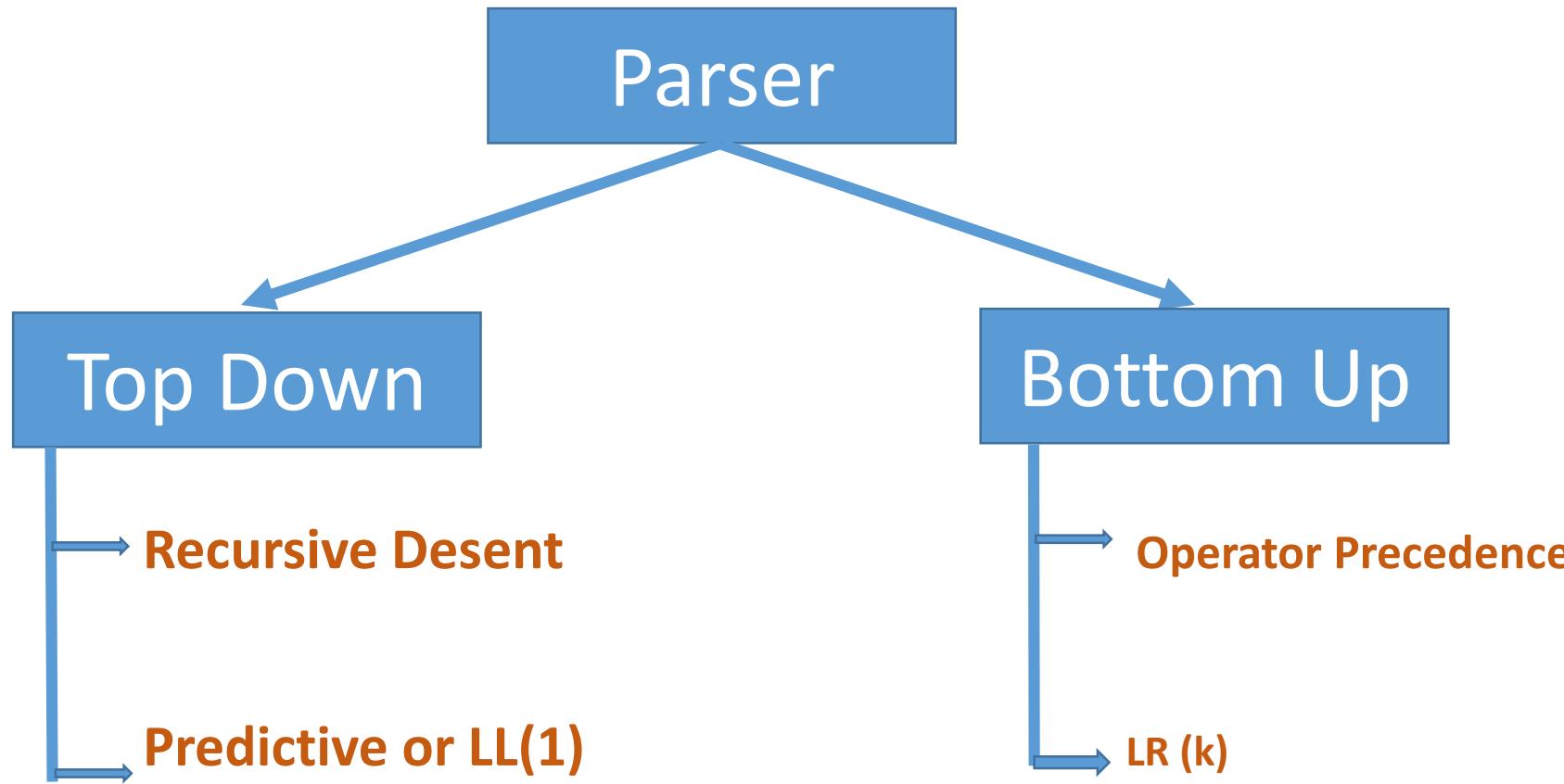
Parsing

- Process of determination whether a string can be generated by a grammar.



Types of parsing

- **Top-down parsing:** Construction of the parse tree starts at the root (from the start symbol) and proceeds towards leaves (token or terminals).
- **Bottom-up parsing:** Construction of the parse tree starts from the leaf nodes (tokens or terminals of the grammar) and proceeds towards root (start symbol).

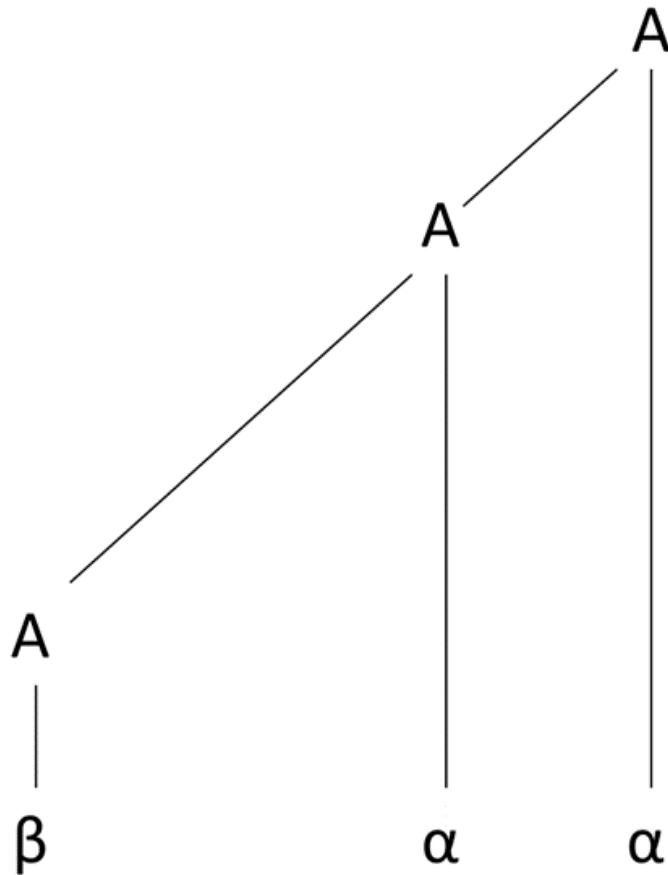


Note: except operator precedence parser all parsers work on unambiguous grammar

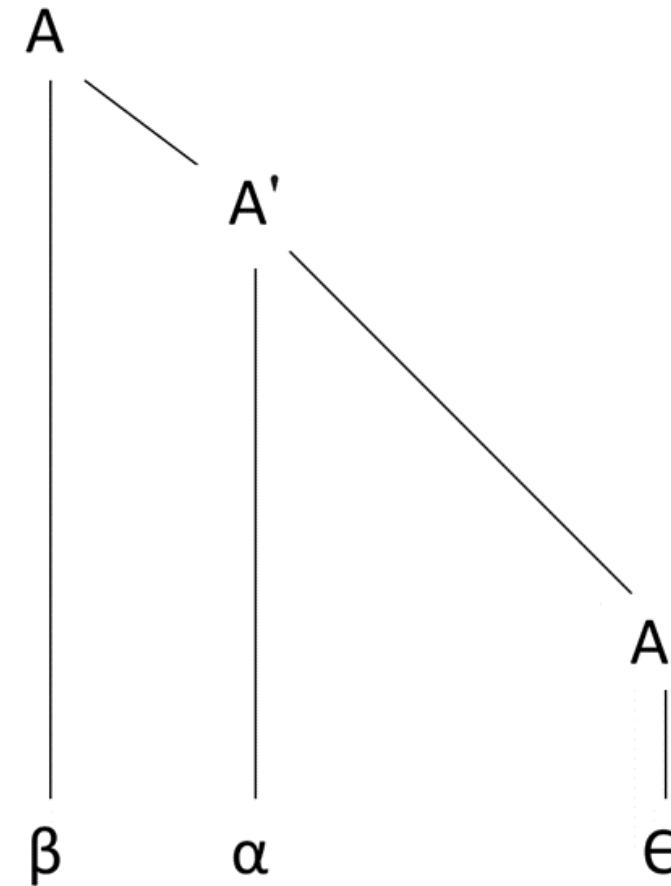
Left recursion

- A top down parser with production
 $A \rightarrow A \alpha$ may loop forever
- From the grammar $A \rightarrow A \alpha \mid \beta$
left recursion may be eliminated by transforming the grammar to
 $A \rightarrow \beta A'$
 $A' \rightarrow \alpha A' \mid \epsilon$

Parse tree corresponding to a left recursive grammar



Parse tree corresponding to the modified grammar



Example

- Consider grammar for arithmetic expressions

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

After removal of left recursion the grammar becomes

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

Removal of left recursion

In general

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

Where, no β begin with A

Then

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

$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \dots \mid \alpha_m A' \mid \epsilon$$

Left recursion hidden due to many productions

Left recursion may also be introduced by two or more grammar rules.

For example:

$$S \rightarrow Aa \mid b$$

$$A \rightarrow Ac \mid Sd \mid \epsilon$$

there is a left recursion because $S \rightarrow Aa \rightarrow Sda$

In such cases, left recursion is removed systematically

- Starting from the first rule and replacing all the occurrences of the first non terminal symbol
- Removing left recursion from the modified grammar

Removal of left recursion due to many productions

- After the first step (substitute S by its RHS in the rules) the grammar becomes

$$S \rightarrow Aa \mid b$$
$$A \rightarrow Ac \mid Aad \mid bd \mid \epsilon$$

After the second step (removal of left recursion) the grammar becomes

$$S \rightarrow Aa \mid b$$
$$A \rightarrow bdA' \mid A'$$
$$A' \rightarrow cA' \mid adA' \mid \epsilon$$

Left factoring

- In top-down parsing when it is not clear which production to choose for expansion of a symbol
defer the decision till we have seen enough input.

In general if $A \rightarrow \alpha \beta_1 \mid \alpha \beta_2$
defer decision by expanding A to $\alpha A'$

we can then expand A' to $\beta_1 \mid \beta_2$

- Therefore $A \rightarrow \alpha \beta_1 \mid \alpha \beta_2$
transforms to $A \rightarrow \alpha A'$
 $A' \rightarrow \beta_1 \mid \beta_2$

Suggested Book

Compilers: Principles, Techniques, and Tools

by Aho, Sethi, Ullman and Lam



Compiler Design

Associativity

- If an operand has operator on both the sides, the side on which operator takes this operand is the associativity of that operator.
 - In $a+b+c$, b is taken by left +
 - +, -, *, / are left associative
 - ^, = are right associative

Consider grammar $E \rightarrow E + E \mid id$

Derive “id+id+id”.

Make grammar left recursive

$E \rightarrow E + id \mid id$

Consider grammar $E \rightarrow E^{\wedge} E \mid id$

Derive “ $id^{\wedge}id^{\wedge}id$ ” or $2^{\wedge} 2^{\wedge} 3$.

Make grammar right recursive

$E \rightarrow id^{\wedge} E \mid id$

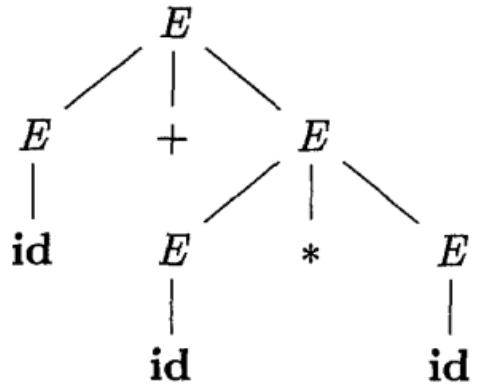
or $E \rightarrow F^{\wedge} E \mid F$,

$F \rightarrow id$

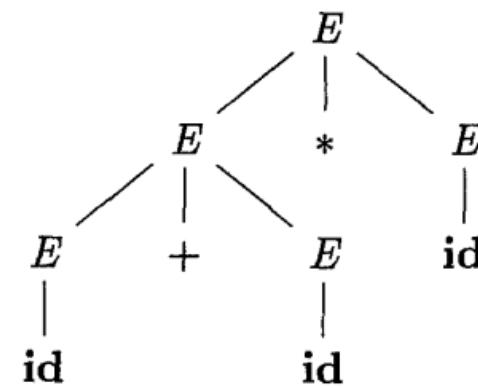
Precedence

Consider grammar $E \rightarrow E + E \mid E * E \mid id$

Derive “id+id*id”.



(a)



(b)

- String $id+id*id$ has two possible interpretations because of two different parse trees corresponding to $id+(id*id)$ and $(id+id)*id$.
- Precedence determines the correct interpretation.

Precedence in the Grammar for Arithmetic Expressions

- Ambiguous

$$E \rightarrow E + E$$
$$E \rightarrow E * E$$
$$E \rightarrow id$$

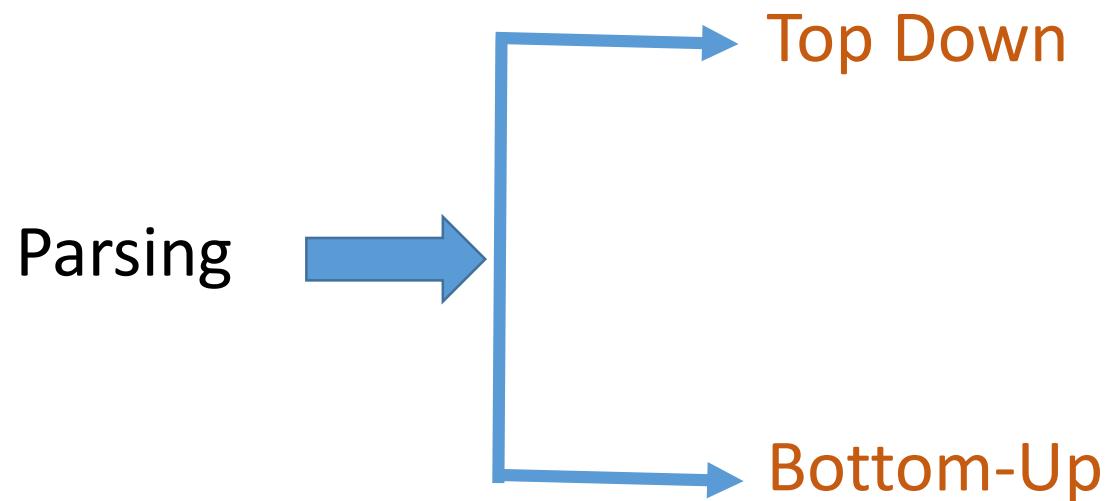
Associativity and precedence rules honored

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

Note: highest precedence operator should be kept on lower level.

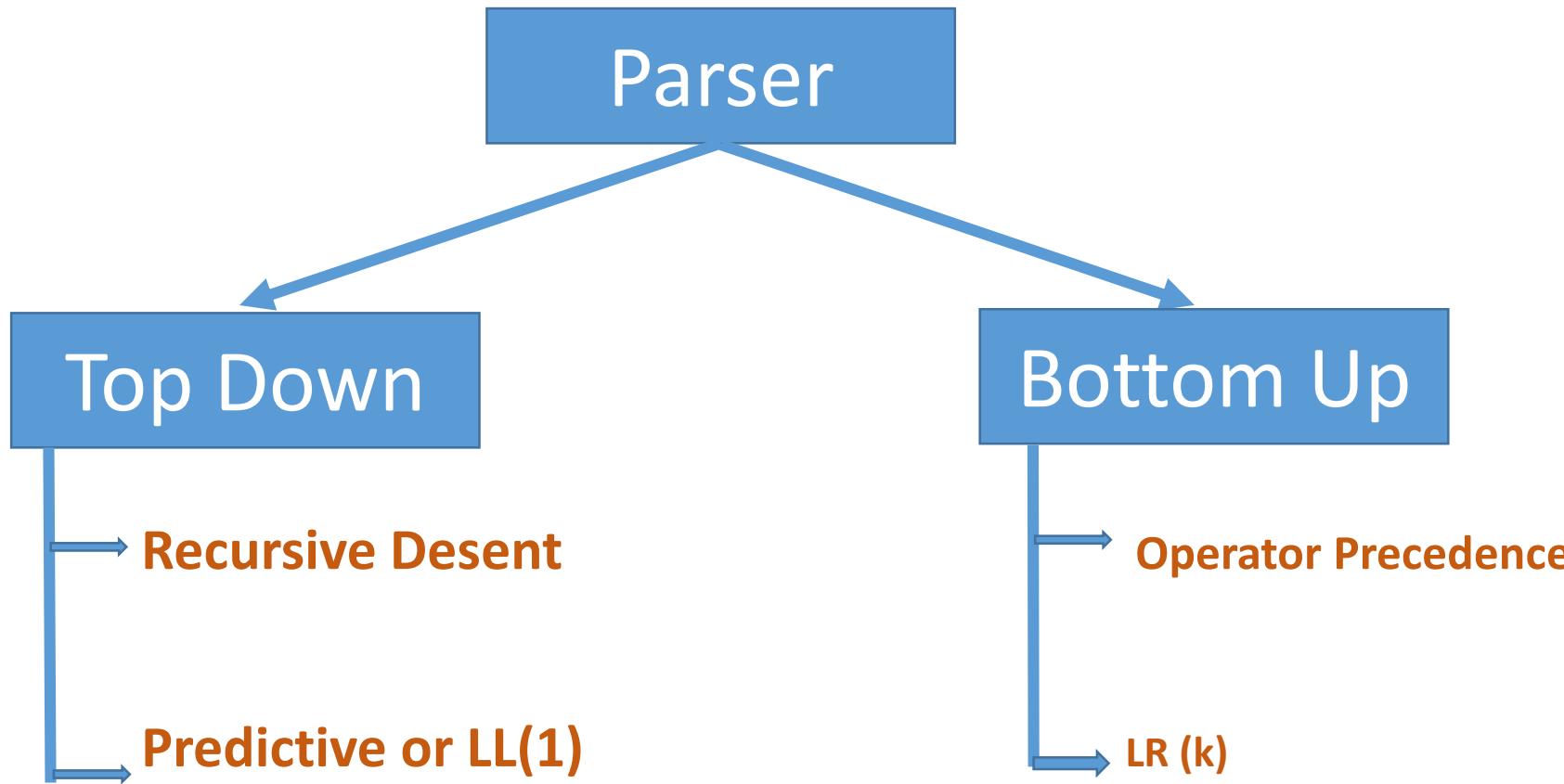
Parsing

- Process of determination whether a string can be generated by a grammar.



Types of parsing

- **Top-down parsing:** Construction of the parse tree starts at the root (from the start symbol) and proceeds towards leaves (token or terminals).
- **Bottom-up parsing:** Construction of the parse tree starts from the leaf nodes (tokens or terminals of the grammar) and proceeds towards root (start symbol).

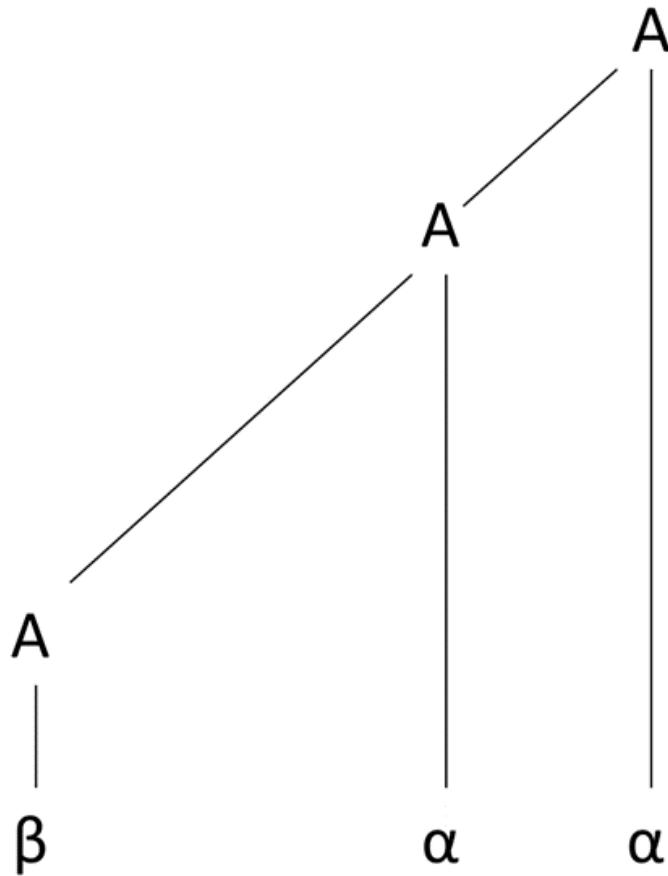


Note: except operator precedence parser all parsers work on unambiguous grammar

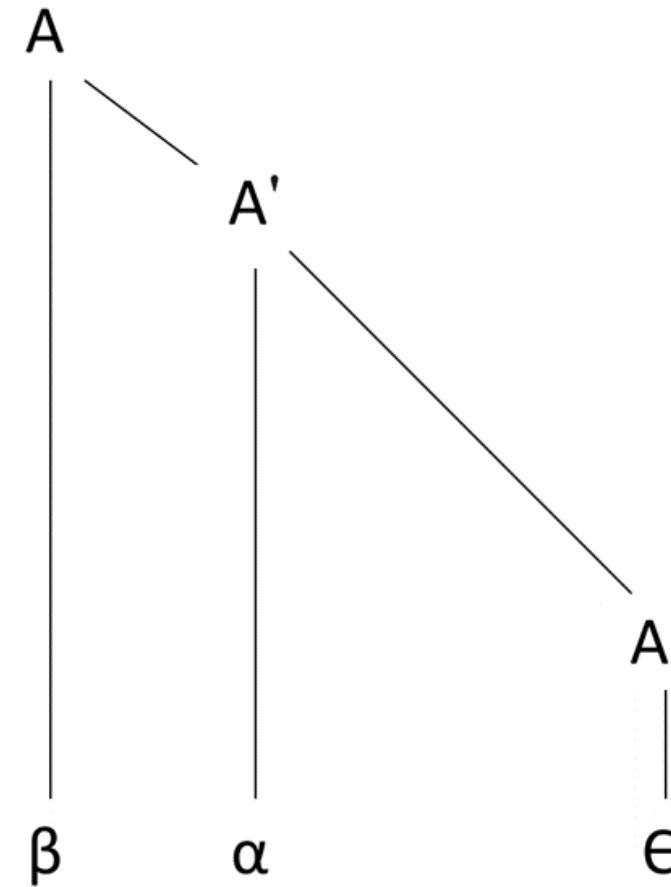
Left recursion

- A top down parser with production
 $A \rightarrow A \alpha$ may loop forever
- From the grammar $A \rightarrow A \alpha \mid \beta$
left recursion may be eliminated by transforming the grammar to
 $A \rightarrow \beta A'$
 $A' \rightarrow \alpha A' \mid \epsilon$

Parse tree corresponding to a left recursive grammar



Parse tree corresponding to the modified grammar



Example

- Consider grammar for arithmetic expressions

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

After removal of left recursion the grammar becomes

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

Removal of left recursion

In general

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

Where, no β begin with A

Then

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

$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \dots \mid \alpha_m A' \mid \epsilon$$

Left recursion hidden due to many productions

Left recursion may also be introduced by two or more grammar rules.

For example:

$$S \rightarrow Aa \mid b$$

$$A \rightarrow Ac \mid Sd \mid \epsilon$$

there is a left recursion because $S \rightarrow Aa \rightarrow Sda$

In such cases, left recursion is removed systematically

- Starting from the first rule and replacing all the occurrences of the first non terminal symbol
- Removing left recursion from the modified grammar

Removal of left recursion due to many productions

- After the first step (substitute S by its RHS in the rules) the grammar becomes

$$S \rightarrow Aa \mid b$$
$$A \rightarrow Ac \mid Aad \mid bd \mid \epsilon$$

After the second step (removal of left recursion) the grammar becomes

$$S \rightarrow Aa \mid b$$
$$A \rightarrow bdA' \mid A'$$
$$A' \rightarrow cA' \mid adA' \mid \epsilon$$

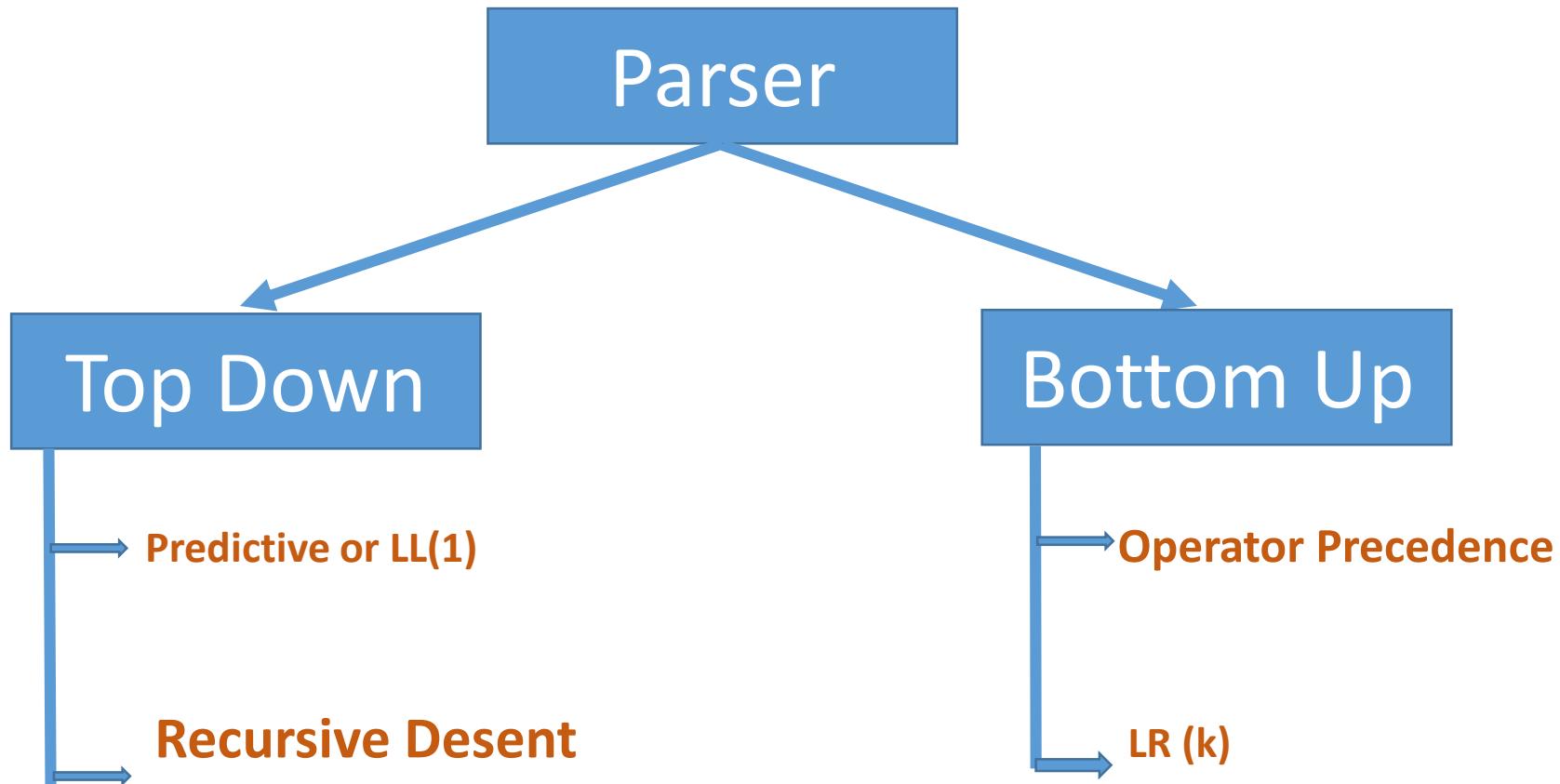
Left factoring

- In top-down parsing when it is not clear which production to choose for expansion of a symbol
defer the decision till we have seen enough input.

In general if $A \rightarrow \alpha \beta_1 \mid \alpha \beta_2$
defer decision by expanding A to $\alpha A'$

we can then expand A' to $\beta_1 \mid \beta_2$

- Therefore $A \rightarrow \alpha \beta_1 \mid \alpha \beta_2$
transforms to $A \rightarrow \alpha A'$
 $A' \rightarrow \beta_1 \mid \beta_2$



Note: except operator precedence parser all parsers work on unambiguous grammar

Top Down Parser

- Recursive Descent parser
- Predictive parser or LL(1) parser

Example

- Consider grammar for arithmetic expressions

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

After removal of left recursion the grammar becomes

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

Consider the modified grammar and parse
the string $\text{id}+\text{id}^*\text{id}$

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \epsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \epsilon$

$F \rightarrow (E) \mid \text{id}$

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.

Consider the modified grammar and parse the string $\text{id} + \text{id}^* \text{id}$

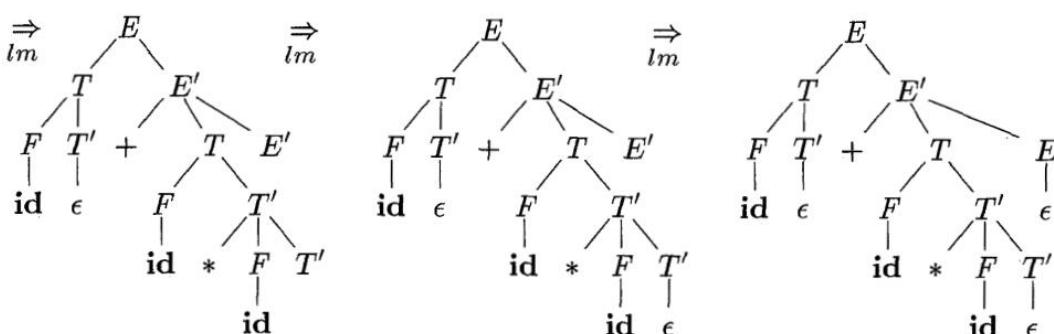
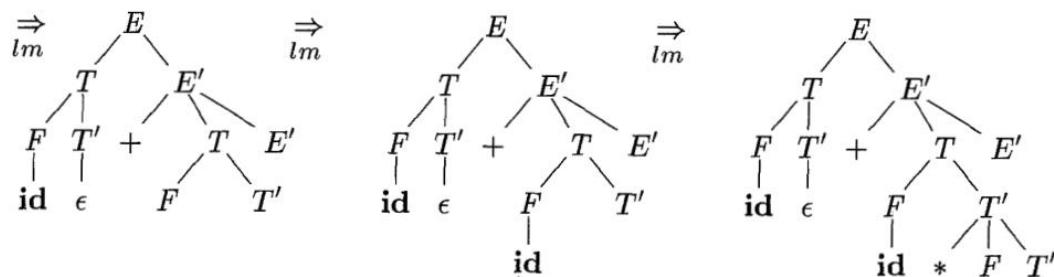
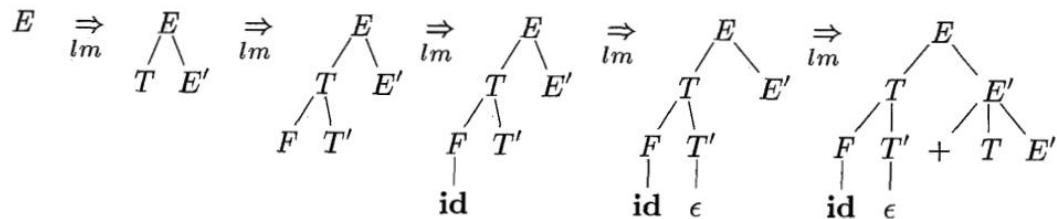
$$E \rightarrow T E'$$

$$E' \rightarrow + T E' \mid \epsilon$$

$$T \rightarrow F T'$$

$$T' \rightarrow * F T' \mid \epsilon$$

$$F \rightarrow (E) \mid \text{id}$$



Recursive Descent parser

- A top down parser that can be implemented by writing a set of recursive procedures to process the input.
- 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.
- Procedures will take care of the left most derivations for each non-terminal while processing the input.

Recursive Descent parser

```
void A() {  
    1) Choose an A-production, A → X1 X2 . . . Xk;  
    2) for (i=1 to k) {  
        3)     if ( Xi is a nonterminal )  
        4)         call procedure Xi() ;  
        5)     else if ( Xi equals the current input symbol a )  
        6)         advance the input to the next symbol;  
    7)     else /* an error has occurred */;  
            }  
}
```

A typical procedure for a non-terminal in a top down parser

Note that this pseudocode is nondeterministic, since it begins by choosing the A-production to apply in a manner that is not specified.

- General recursive-descent may require backtracking; that is, it may require repeated scans over the input.
- However, backtracking is rarely needed to parse programming language constructs, so backtracking parsers are not seen frequently.

- . To allow backtracking, the above code needs to be modified.
 - First, we cannot choose a unique A-production at line (1), so we must try each of several productions in some order.
 - Then, failure at line (7) is not ultimate failure, but suggests only that we need to return to line (1) and try another A-production.
 - Only if there are no more A-productions to try do we declare that an input error has been found.
 - In order to try another A-production, we need to be able to reset the input pointer to where it was when we first reached line (1). Thus, a local variable is needed to store this input pointer for future use.

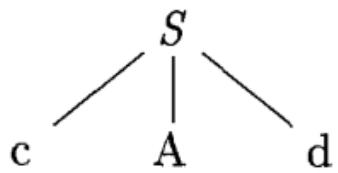
Example

Consider the grammar

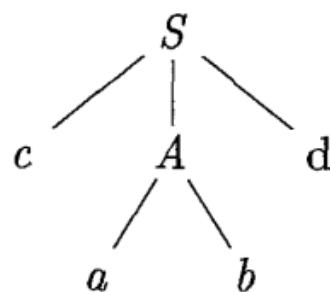
$$S \rightarrow cAd$$

$$A \rightarrow ab \mid a$$

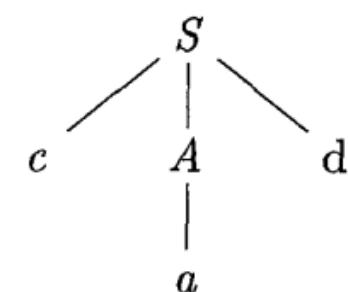
Construct a parse tree top-down for the input string $w = cad$



(a)



(b)



(c)

Predictive Parser or LL(1) parser

- 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.

FIRST and FOLLOW

- During top-down parsing, FIRST and FOLLOW allow us to choose which production to apply, based on the next input symbol.
- $\text{FIRST}(\alpha)$, where α is any string of grammar symbols, to be the set of terminals that begin strings derived from α .
 $\alpha^* \Rightarrow \epsilon$, then ϵ is also in $\text{FIRST}(\alpha)$.
- $\text{FOLLOW}(A)$, for nonterminal A , to be the set of terminals ‘a’ that can appear immediately to the right of A in some sentential form; that is, the set of terminals a such that there exists a derivation of the form
 $S^* \Rightarrow \alpha A a \beta$ for some α and β

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

Compute FIRST sets

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. If X is a terminal symbol then $\text{First}(X) = \{X\}$
2. If $X \rightarrow \epsilon$ is a production then ϵ is in $\text{First}(X)$
3. If X is a non terminal and $X \rightarrow Y_1 Y_2 \dots Y_k$ is a production for some $k \geq 1$ then place a in $\text{First}(X)$ if for some i, a is in $\text{FIRST}(Y_i)$ and ϵ is in all of $\text{First}(Y_1), \dots, \text{First}(Y_{i-1})$ i.e. $Y_1 Y_2 \dots Y_{i-1} \xrightarrow{*} \epsilon$.
if ϵ is in $\text{FIRST}(Y_j)$ for all $j=1 \dots k$ add ϵ to $\text{FIRST}(X)$.

For the expression grammar, Compute FIRST

$$E \rightarrow TE'$$

$$E' \rightarrow +TE' \mid \epsilon$$

$$T \rightarrow FT'$$

$$T' \rightarrow *FT' \mid \epsilon$$

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

$$\text{First}(E) = \text{First}(T) = \text{First}(F) = \{ (, \text{id} \} \}$$

$$\text{First}(E') = \{ +, \epsilon \}$$

$$\text{First}(T') = \{ *, \epsilon \}$$

Compute FOLLOW sets

To compute $\text{FOLLOW}(X)$ for all nonterminals X , apply the following rules until nothing can be added to any FOLLOW set.

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

For the expression grammar

$$E \rightarrow T E'$$

$$E' \rightarrow + T E' \mid \epsilon$$

$$T \rightarrow F T'$$

$$T' \rightarrow * F T' \mid \epsilon$$

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

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

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

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

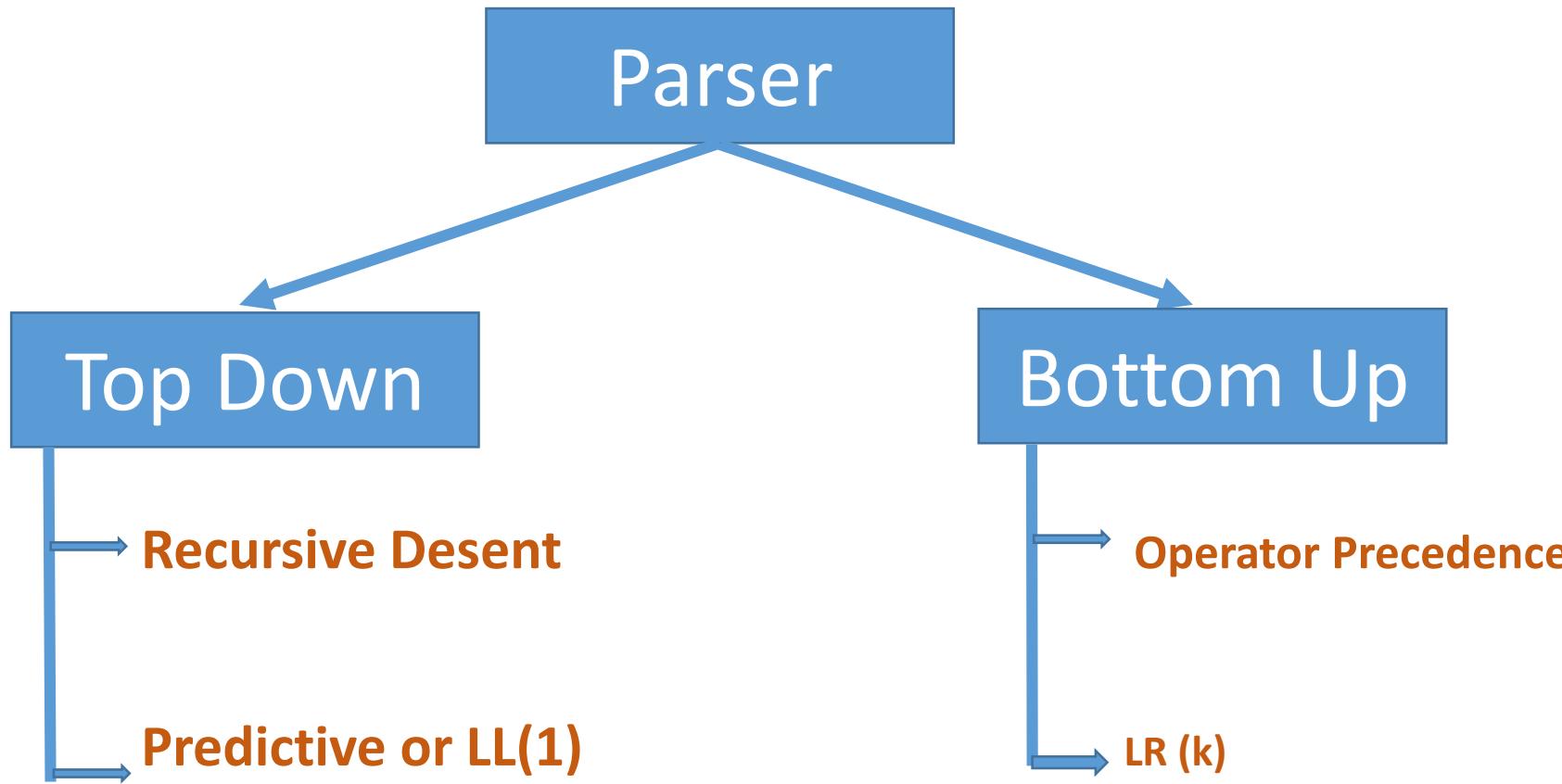
Suggested Book

Compilers: Principles, Techniques, and Tools

by Aho, Sethi, Ullman and Lam



Compiler Design



Note: except operator precedence parser all parsers work on unambiguous grammar

Top Down Parser

- Recursive Descent parser
- Predictive parser or LL(1) parser

Predictive Parser or LL(1) parser

- 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.

FIRST and FOLLOW

- During top-down parsing, FIRST and FOLLOW allow us to choose which production to apply, based on the next input symbol.
- $\text{FIRST}(\alpha)$, where α is any string of grammar symbols, to be the set of terminals that begin strings derived from α .
 $\alpha^* \Rightarrow \epsilon$, then ϵ is also in $\text{FIRST}(\alpha)$.
- $\text{FOLLOW}(A)$, for nonterminal A , to be the set of terminals ‘a’ that can appear immediately to the right of A in some sentential form; that is, the set of terminals a such that there exists a derivation of the form
 $S^* \Rightarrow \alpha A a \beta$ for some α and β

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

Compute FIRST sets

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. If X is a terminal symbol then $\text{First}(X) = \{X\}$
2. If $X \rightarrow \epsilon$ is a production then ϵ is in $\text{First}(X)$
3. If X is a non terminal and $X \rightarrow Y_1 Y_2 \dots Y_k$ is a production for some $k \geq 1$ then place a in $\text{First}(X)$ if for some i, a is in $\text{FIRST}(Y_i)$ and ϵ is in all of $\text{First}(Y_1), \dots, \text{First}(Y_{i-1})$ i.e. $Y_1 Y_2 \dots Y_{i-1} \xrightarrow{*} \epsilon$.
if ϵ is in $\text{FIRST}(Y_j)$ for all $j=1 \dots k$ add ϵ to $\text{FIRST}(X)$.

For the expression grammar, Compute FIRST

$$E \rightarrow TE'$$

$$E' \rightarrow +TE' \mid \epsilon$$

$$T \rightarrow FT'$$

$$T' \rightarrow *FT' \mid \epsilon$$

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

$$\text{First}(E) = \text{First}(T) = \text{First}(F) = \{ (, \text{id} \} \}$$

$$\text{First}(E') = \{ +, \epsilon \}$$

$$\text{First}(T') = \{ *, \epsilon \}$$

Compute FOLLOW sets

To compute $\text{FOLLOW}(X)$ for all nonterminals X , apply the following rules until nothing can be added to any FOLLOW set.

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

For the expression grammar

$$E \rightarrow T E'$$

$$E' \rightarrow + T E' \mid \epsilon$$

$$T \rightarrow F T'$$

$$T' \rightarrow * F T' \mid \epsilon$$

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

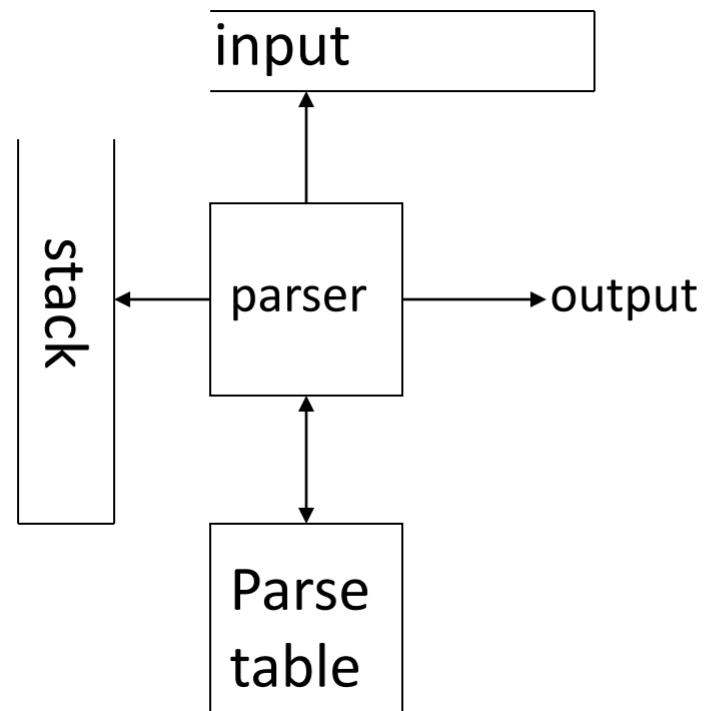
$$\text{follow}(E) = \text{follow}(E') = \{ \$,) \}$$

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

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

Predictive parsing

- Predictive parser can be implemented by maintaining an external stack



Parse table is a
two dimensional array
 $M[X,a]$ where “X” is a
non terminal and “a” is
a terminal of the grammar

Algorithm: Construction of Predictive parsing table

Input: Grammar G

Output: Parsing table M

Method: for each production $A \rightarrow \alpha$ do the following:

1. For each terminal 'a' in $\text{first}(\alpha)$, add

$$M[A,a] = A \rightarrow \alpha$$

2. If ϵ is in $\text{First}(\alpha)$

$$M[A,b] = A \rightarrow \alpha$$

for each terminal b in $\text{follow}(A)$

If ϵ is in $\text{First}(\alpha)$ and $\$$ is in $\text{follow}(A)$

$$M[A,\$] = A \rightarrow \alpha$$

3. 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 represented by an empty entry in the table).

Note: A grammar whose parse table has no multiple entries is called LL(1)

Example

- Consider the grammar

$$E \rightarrow T E'$$

$$E' \rightarrow +T E' \mid \epsilon$$

$$T \rightarrow F T'$$

$$T' \rightarrow * F T' \mid \epsilon$$

$$F \rightarrow (E) \mid \text{id}$$

	FIRST	FOLLOW
E	{ (, id }	{), \$ }
E'	{ +, ε }	{), \$ }
T	{ (, id }	{ +,), \$ }
T'	{ *, ε }	{ +,), \$ }
F	{ (, id }	{ *, +,), \$ }

Example

- Consider the grammar

$$E \rightarrow T E'$$

$$E' \rightarrow +T E' \mid \epsilon$$

$$T \rightarrow F T'$$

$$T' \rightarrow * F T' \mid \epsilon$$

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

	FIRST	FOLLOW
E	{ (, id }	{), \$ }
E'	{ +, ε }	{), \$ }
T	{ (, id }	{ +,), \$ }
T'	{ *, ε }	{ +,), \$ }
F	{ (, id }	{ *, +,), \$ }

	id	+	*	()	\$
E	E → TE'			E → TE'		
E'		E' → +TE'			E' → ε	E' → ε
T	T → FT'			T → FT'		
T'		T' → ε	T' → *FT'		T' → ε	T' → ε
F	F → id			F → (E)		

Blank entries are error states. For example
E cannot derive a string starting with ‘+’

Parse table for the grammar

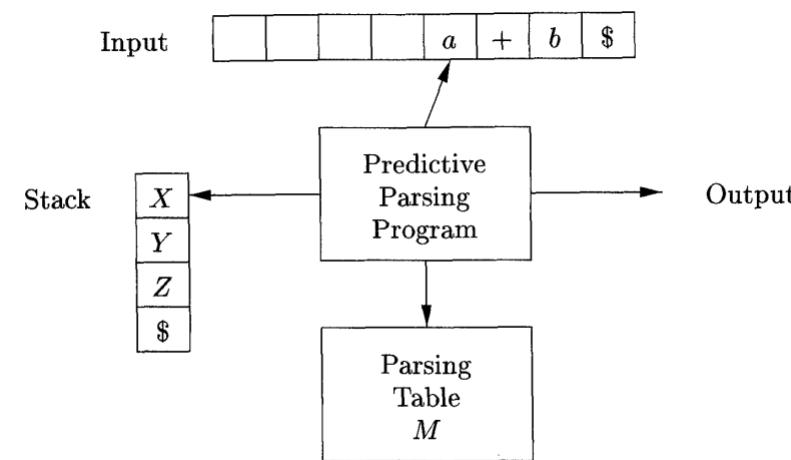
	id	+	*	()	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow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

Blank entries are error states. For example
E cannot derive a string starting with '+'

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



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 $\$$.

The predictive parsing table M is used to parse a input string.

let 'a' be the first symbol of w ;

let X be the top stack symbol;

while ($X \neq \$$) { /* stack is not empty */

 if (X is a) pop the stack and let a be the next symbol;

 elseif (X is a terminal) error();

 else if ($M[X, a]$ is an error entry) error();

 else if ($M[X, a] = X \rightarrow Y_1 Y_2 \dots Y_k$)

 {

 output the production $X \rightarrow Y_1 Y_2 \dots Y_k$

 pop the stack

 push Y_k, Y_{k-1}, \dots, Y_1 onto the stack, with Y_1 on top;

 }

 Let X be the top stack symbol;

}

Parse the string “id+id*id” using LL(1) parser

Matched string	Stack	Input	Action
	E \$	id + id * id \$	expand by $E \rightarrow TE'$
	TE'\$	id + id * id \$	expand by $T \rightarrow FT'$
	FT'E'\$	id + id * id \$	expand by $F \rightarrow id$
	id T'E'\$	id + id * id \$	Pop id and + be the next symbol
id	T'E'\$	+ id * id \$	expand by $T' \rightarrow \epsilon$
id	E'\$	+ id * id \$	expand by $E' \rightarrow +TE'$
id	+TE'\$	+ id * id \$	Pop + and id be the next symbol
id+	TE'\$	id * id \$	expand by $T \rightarrow FT'$
id+	FT'E'\$	id * id \$	expand by $F \rightarrow id$
id+	idT'E'\$	id * id \$	pop id and * be the next symbol
id+id	T'E'\$	* id \$	expand by $T' \rightarrow *FT'$
id+id	*FT'E'\$	* id \$	Pop * and id be the next symbol
id+id*	FT'E'\$	id \$	expand by $F \rightarrow id$
id+id*	idT'E'\$	id \$	pop id and \$ be the next symbol
id+id*id	T'E'\$	\$	expand by $T' \rightarrow \epsilon$
id+id*id	E'\$	\$	expand by $E' \rightarrow \epsilon$
id+id*id	\$	\$	halt

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

LL(1) Grammar

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

1. For no terminal a do both α and β derive strings beginning with a .

i.e. $\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \emptyset$

2. At most one of α and β can derive the empty string.

3. If $\beta \xrightarrow{*} \epsilon$, then α does not derive any string beginning with a terminal in $\text{FOLLOW}(A)$.
i.e. $\text{FIRST}(\alpha) \cap \text{FOLLOW}(A) = \emptyset$

Likewise, if $\alpha \xrightarrow{*} \epsilon$, then β does not derive any string beginning with a terminal in $\text{FOLLOW}(A)$.

i.e. $\text{FIRST}(\beta) \cap \text{FOLLOW}(A) = \emptyset$

Check whether the following grammars are LL(1)?

$$G_1 : S \rightarrow aSa \mid bS \mid c$$

$$G_2 : S \rightarrow iEtSS' \mid a, \quad S' \rightarrow eS \mid \epsilon$$

For Grammar $G_1 : S \rightarrow aSa \mid bS \mid c$

$$\text{FIRST}(aSa) \cap \text{FIRST}(bS) \cap \text{FIRST}(c)$$

$$a \cap b \cap c$$

$$=\emptyset, G_1 \text{ is LL(1)}$$

For Grammar $G_2 : S \rightarrow iEtSS' \mid a, \quad S' \rightarrow eS \mid \epsilon$

$$\text{FIRST}(iEtSS') \cap \text{FIRST}(a)$$

$$i \cap a$$

$$=\emptyset$$

For production rule, $S' \rightarrow eS \mid \epsilon$

$$\text{FIRST}(eS) \cap \text{FOLLOW}(S')$$

$$e \cap \text{FOLLOW}(S)$$

$$e \cap \{\text{FIRST}(S') \cup \$\}$$

$$e \cap \{e, \$\} \neq \emptyset$$

Hence, G_2 is not LL(1)

$$S \rightarrow cAd; \quad A \rightarrow bc \mid a$$
$$\text{FIRST}(A) = \{ b, a \}$$

Derive cad?

$$S \rightarrow cAd \rightarrow cad$$

Since $\text{first}(A)$ contains a so we can directly use the respective production.

$A \rightarrow aBb; B \rightarrow c \mid \epsilon$

Derive ab

$A \rightarrow aBb$

B does not start with b, but $B \rightarrow \epsilon$ can be used to vanish B and generate ab.

Note: Parser can vanish B only if it knows that the character that follows B in the production is same as the current character in the input FOLLOW (B) = {b}

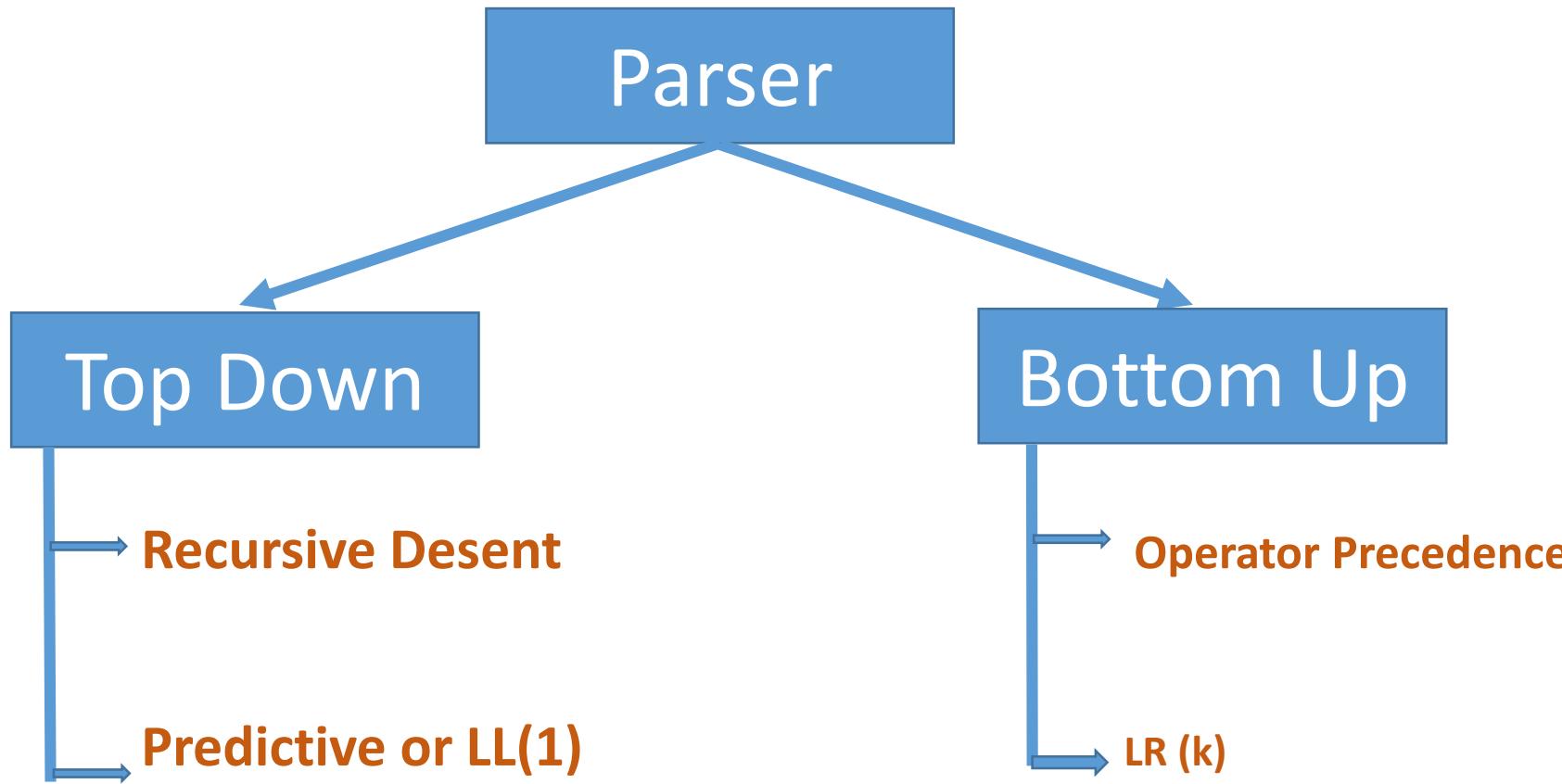
Suggested Book

Compilers: Principles, Techniques, and Tools

by Aho, Sethi, Ullman and Lam



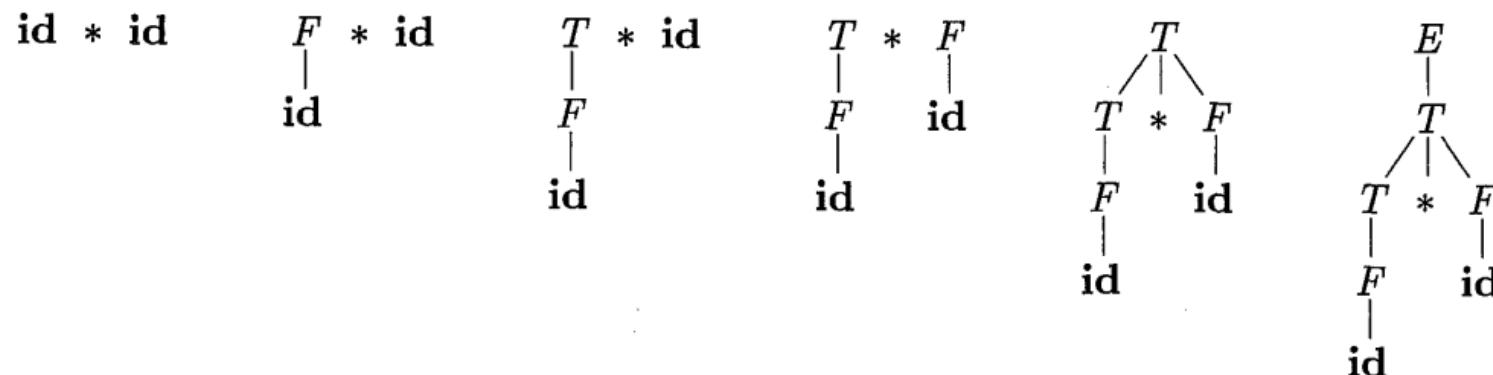
Compiler Design



Note: except operator precedence parser all parsers work on unambiguous grammar

Bottom Up Parser

- A bottom-up parse 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)**.
- Consider the grammar $E \rightarrow E+T \mid T, T \rightarrow T^*F \mid F, F \rightarrow id$
- Parse the string **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 key decisions during bottom-up parsing are about when to reduce and about what production to apply, as the parse proceeds.

$\text{id} * \text{id}$, $F * \text{id}$, $T * \text{id}$, $T * F$, T , E

Note: Bottom Up parsing may be viewed as one of finding and reducing handles.

Handles

- A handle of a right sentential form is a production $A \rightarrow \beta$ & a position γ where the string β may be found and replaced by A to produce the previous right sentential form in a right most derivation of γ .

i.e. if

$S \xrightarrow{*} \alpha A w \xrightarrow{*} \alpha \beta W$ then $A \rightarrow \beta$ in the position following α is handle of $\alpha \beta w$

- Note: the string w to the right of the handle contains only terminal symbols.
If a grammar is unambiguous then every right sentential form of the grammar has exactly one handle.

For the expression grammar,

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

Derive $id+ id*id$

$$E \Rightarrow E+E \Rightarrow E+E*E \Rightarrow E+E*id \Rightarrow E+id*id \Rightarrow id+id*id$$

Rightmost derivation

Sentential form	Handle
$id+id*id \$$	$E \rightarrow id$ at position preceding +
$E+id*id \$$	$E \rightarrow id$ at position following +
$E+E*id \$$	$E \rightarrow id$ at position following *
$E+E*E \$$	$E \rightarrow E*E$ at position following +
$E+E \$$	$E \rightarrow E+E$ at position preceding end marker

Bottom-up parsing is only an attempt to detect the handle of a right sentential form and whenever a handle is detected, the reduction is performed.

This is equivalent to rightmost derivation in reverse, called handle pruning.

Shift-Reduce parsing

Shift-reduce parsing is a form of bottom-up parsing in which a stack holds grammar symbols and an input buffer holds the rest of the string to be parsed.

As we shall see, the handle always appears at the top of the stack just before it is identified as the handle.

We use \$ to mark the bottom of the stack and also the right end of the input string

STACK
\$

INPUT
 $w \$$

Stack implementation of Shift-Reduce parsing

In this method, there is a **stack** and an **input buffer**, 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**.

The parser repeats this cycle until it has detected an error or stack contains the start symbol and the input buffer is empty.

Shift-Reduce parsing

There are actually four possible actions a shift reduce parser can make:
(1) shift, (2) reduce, (3) accept, and (4) error.

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.

Consider the grammar

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

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

$$F \rightarrow id$$

parse the input string “id*id”

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

Now try to parse “id+id*id”

Practice problem

- Consider the grammar

$$E \rightarrow E + E$$
$$E \rightarrow E * E$$
$$E \rightarrow id$$

and parse the string “id+id*id”

Problems with stack implementation

- How to locate a handle in a right sentential form
- What production to choose in case there is more than one production with same right side.
- In this parsing the precedence of operator is not considered, for ambiguous grammar, hence may produce incorrect result. This problem is resolved by operator precedence parser.

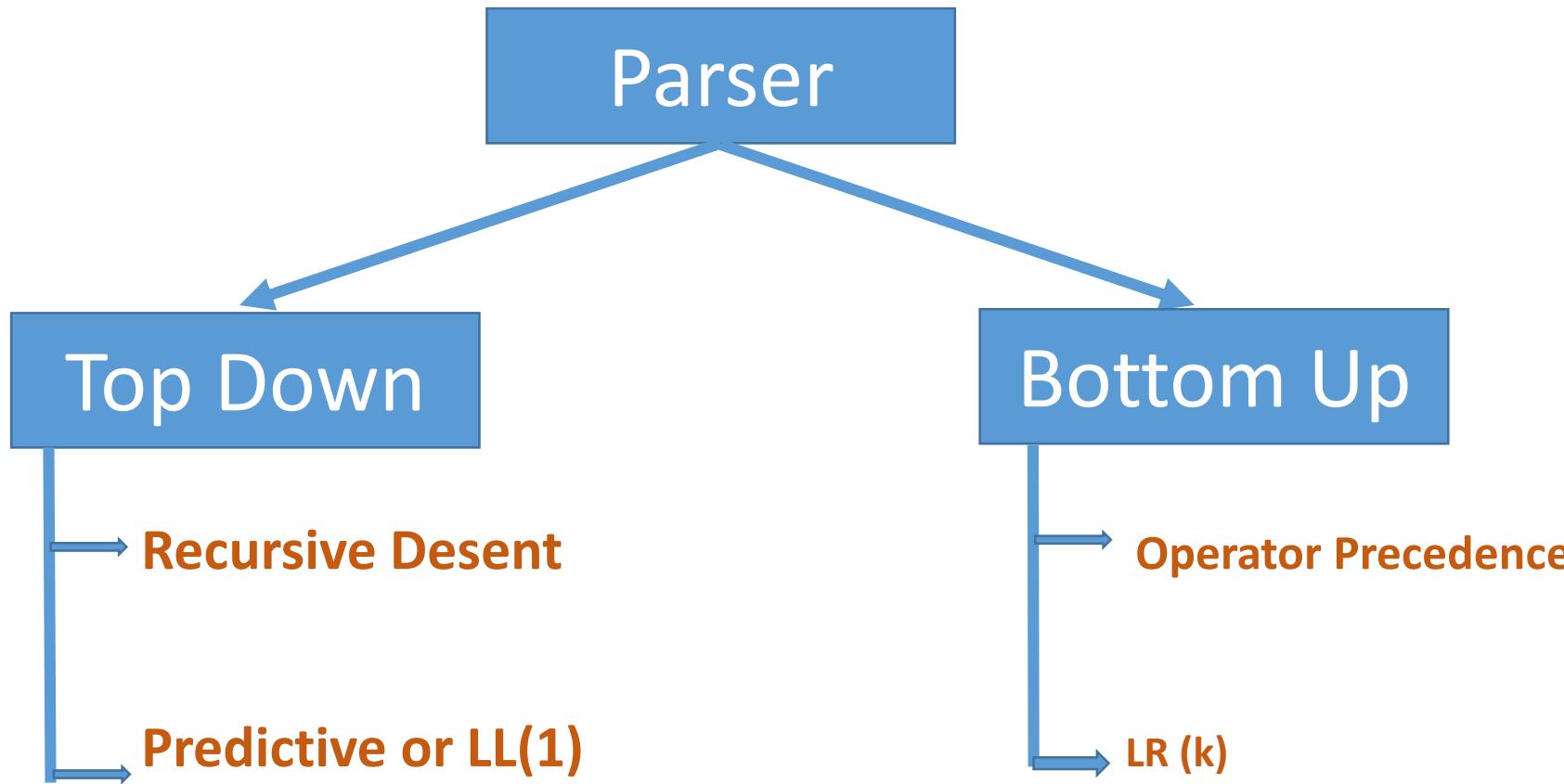
Suggested Book

Compilers: Principles, Techniques, and Tools

by Aho, Sethi, Ullman and Lam



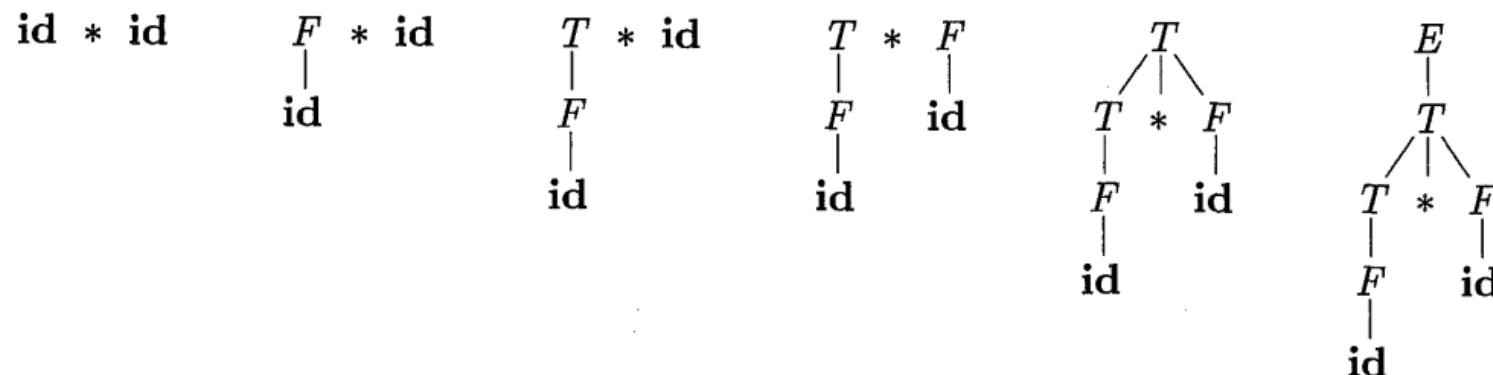
Compiler Design



Note: except operator precedence parser all parsers work on unambiguous grammar

Bottom Up Parser

- A bottom-up parse 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)**.
- Consider the grammar $E \rightarrow E+T \mid T, T \rightarrow T^*F \mid F, F \rightarrow id$
- Parse the string **id *id**



Consider the grammar

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

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

$$F \rightarrow id$$

parse the input string “id*id”

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

Now try to parse “id+id*id”

Practice problem

- Consider the grammar

$$E \rightarrow E + E$$
$$E \rightarrow E * E$$
$$E \rightarrow id$$

and parse the string “id+id*id”

Shift Reduce Parsers

- The general shift-reduce technique is:
 - if there is no handle on the stack then shift
 - If there is a handle then reduce
- Bottom up parsing is essentially the process of detecting handles and reducing them.
- Different bottom-up parsers differ in the way they detect handles.

Conflicts

- What happens when there is a choice
 - What action to take in case both shift and reduce are valid? (**shift-reduce conflict**)
 - Which rule to use for reduction if ? (**reduce-reduce conflict**)
- Conflicts come either because of ambiguous grammars or parsing method is not powerful enough.

Shift reduce conflict

Consider the grammar $E \rightarrow E+E \mid E^*E \mid id$

and the input $id+id^*id$

stack	input	action
$E+E$	$*id$	reduce by $E \rightarrow E+E$
E	$*id$	shift
E^*	id	shift
E^*id		reduce by $E \rightarrow id$
E^*E		reduce by $E \rightarrow E^*E$
E		

stack	input	action
$E+E$	$*id$	shift
$E+E^*$	id	shift
$E+E^*id$		reduce by $E \rightarrow id$
$E+E^*E$		reduce by $E \rightarrow E^*E$
$E+E$		reduce by $E \rightarrow E+E$
E		

Reduce reduce conflict

Consider the grammar $M \rightarrow R+R \mid R+c \mid R$

$R \rightarrow c$

and the input

$c+c$

Stack	input	action
	$c+c$	shift
c	$+c$	reduce by $R \rightarrow c$
R	$+c$	shift
$R+$	c	shift
$R+c$		reduce by $R \rightarrow c$
$R+R$		reduce by $M \rightarrow R+R$
M		

Stack	input	action
	$c+c$	shift
c	$+c$	reduce by $R \rightarrow c$
R	$+c$	shift
$R+$	c	shift
$R+c$		reduce by $M \rightarrow R+R$
M		

Problems with stack implementation

- How to locate a handle in a right sentential form
- What production to choose in case there is more than one production with same right side.

Note: In this parsing the precedence of operator is not considered, for ambiguous grammar, hence may produce incorrect result.

This problem is resolved by operator precedence parser.

Operator Precedence parser

Operator precedence parser can be constructed for grammar which has following properties:

1. No production on the side contain ϵ
2. Grammar should have to be operator grammar.

Operator Grammar (OG)

No production on the right side have adjacent non-terminal.

Example:

$$E \rightarrow EAE \mid (E) \mid -E \mid id$$

$$A \rightarrow + \mid - \mid * \mid / \mid ^$$

The grammar is not OG.

Example:

$$E \rightarrow E+E \mid E-E \mid E^*E \mid E/E \mid E^E$$

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

The grammar is OG.

In OPP, three disjoint precedence relations \lessdot , \gtrdot and \doteq are used between certain pairs of terminals, which guides in selection of handles.

- $a \gtrdot b$ means that terminal "a" has the higher precedence than terminal "b".
- $a \lessdot b$ means that terminal "a" has the lower precedence than terminal "b".
- $a \doteq b$ means that the terminal "a" and "b" both have same precedence.

LEADING and TRAILING SYMBOLS

LEADING(A): is a set of first terminals of non-terminal A.

Algorithm:

1. a is in LEADING(A) if $A \rightarrow \gamma a \delta$. γ is ϵ or a single non terminal
2. If a is in LEADING(B) and $A \rightarrow B \alpha$, then a is in LEADING(A).

Example:

$E \rightarrow E+E \mid E^*E \mid (E) \mid \text{id}$, compute LEADING(E)

$\text{LEADING}(E)=\{\text{+, *, (, id}\}$

TRAILING

TRAILING(A): is a set of last terminals of non-terminal A.

Algorithm:

1. a is in TRAILING(A) if $A \rightarrow \gamma a \delta$, δ is ϵ or a single non terminal
2. If a is in TRAILING(B) and $A \rightarrow \alpha B$, then a is in TRAILING (A).

Example:

$E \rightarrow E+E \mid E^*E \mid (E) \mid \text{id}$, compute TRAILING(E)

LEADING(E)= {+, *,), id}

Algorithm to compute precedence relation

1. Compute LEADING(A) and TRAILING(A) for each non-terminal.
2. For each production $A \rightarrow X_1 X_2 \dots X_n$ do
 for i=1 to n-1 do
 Begin
 a) X_i and X_{i+1} are both terminal then
 Set $X_i \doteq X_{i+1}$
 b) If X_i and X_{i+2} are terminals and X_{i+1} is non-terminal
 Set $X_i \doteq X_{i+2}$
 c) If X_i is terminal and X_{i+1} is non-terminal then
 for each a in LEADING(X_{i+1}) do
 set $X_i < a$
 d) If X_i is non-terminal and X_{i+1} is terminal then
 for each a in TRAILING(X_i) do
 set $a > X_{i+1}$
 END
 3. Set $\$ < a$ for every a in LEADING(S) and set $b > \$$ for every b in TRAILING (S), where S is start symbol of G.

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

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

$F \rightarrow (E) \mid id$

$LEADING(E) = \{+, *, (, id\}$

$TRAILING(E) = \{+, *,), id\}$

$LEADING(T) = \{*, (, id\}$

$TRAILING(T) = \{*,), id\}$

$LEADING(F) = \{(, id\}$

$TRAILING(F) = \{), id\}$

Consider the grammar with productions

$$1. E \rightarrow E + T$$

$$2. E \rightarrow T$$

$$3. T \rightarrow T^* F$$

$$4. T \rightarrow F$$

$$5. F \rightarrow (E)$$

$$6. F \rightarrow id$$

$$\text{LEADING}(E) = \{+, *, (, \text{id}\}$$

$$\text{LEADING}(T) = \{*, (, \text{id}\}$$

$$\text{LEADING}(F) = \{(), \text{id}\}$$

$$\text{TRAILING}(E) = \{+, *,), \text{id}\}$$

$$\text{TRAILING}(T) = \{*,), \text{id}\}$$

$$\text{TRAILING}(F) = \{(), \text{id}\}$$

1. $E \rightarrow E + T$

Trailing (E) $\geq +$

$$+ \geq +$$

$$* \geq +$$

$$) \geq +$$

$$\text{id} \geq +$$

$E \rightarrow E + T$

$+ \triangleleft \text{ Leading } (T)$

$$+ \triangleleft *$$

$$+ \triangleleft ($$

$$+ \triangleleft \text{id}$$

	+	*	()	id	\$
+	\triangleright	\triangleleft	\triangleleft		\triangleleft	\triangleright
*		\triangleright				\triangleright
(
)		\triangleright				\triangleright
id		\triangleright				\triangleright
\$	\triangleleft	\triangleleft	\triangleleft	\triangleleft	\triangleleft	

Consider the grammar with productions

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

$$\text{LEADING}(E) = \{+, *, (, \text{id}\}$$
$$\text{TRAILING}(E) = \{+, *,), \text{id}\}$$

2. $E \rightarrow T$ (no terminal)
3. $T \rightarrow T^* F$

$$\text{Trailing}(T) > *$$

i.e.

$$* \succ *$$
$$) \succ *$$
$$\text{id} \succ *$$

$$\text{LEADING}(T) = \{*, (, \text{id}\}$$
$$\text{TRAILING}(T) = \{*,), \text{id}\}$$

$$T \rightarrow T^* F$$

$$* \triangleleft \text{Leading}(F)$$

i.e.

$$* \triangleleft ($$
$$* \triangleleft \text{id}$$

$$\text{LEADING}(F) = \{(, \text{id}\}$$
$$\text{TRAILING}(F) = \{), \text{id}\}$$

	+	*	()	id	\$
+	>	<	<		<	>
*	>	>	<		<	>
(
)	>	>				>
id	>	>				>
\$	&	&	&		&	

Consider the grammar with productions

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

$$\text{LEADING}(E) = \{+, *, (, \text{id}\}$$

$$\text{TRAILING}(E) = \{+, *,), \text{id}\}$$

$$\text{LEADING}(T) = \{*, (, \text{id}\}$$

$$\text{TRAILING}(T) = \{*,), \text{id}\}$$

$$\text{LEADING}(F) = \{(, \text{id}\}$$

$$\text{TRAILING}(F) = \{), \text{id}\}$$

4. $T \rightarrow F$ (no terminal)

$$5. F \rightarrow (E)$$

$\forall a$ in $\text{LEADING}(E)$

i.e.

(<+)

(<*)

(<()

(<id)

$$F \rightarrow (E)$$

$\forall a$ in $\text{Trailing}(E)$

+ >)

* >)

) >)

id >)

$$F \rightarrow (E)$$

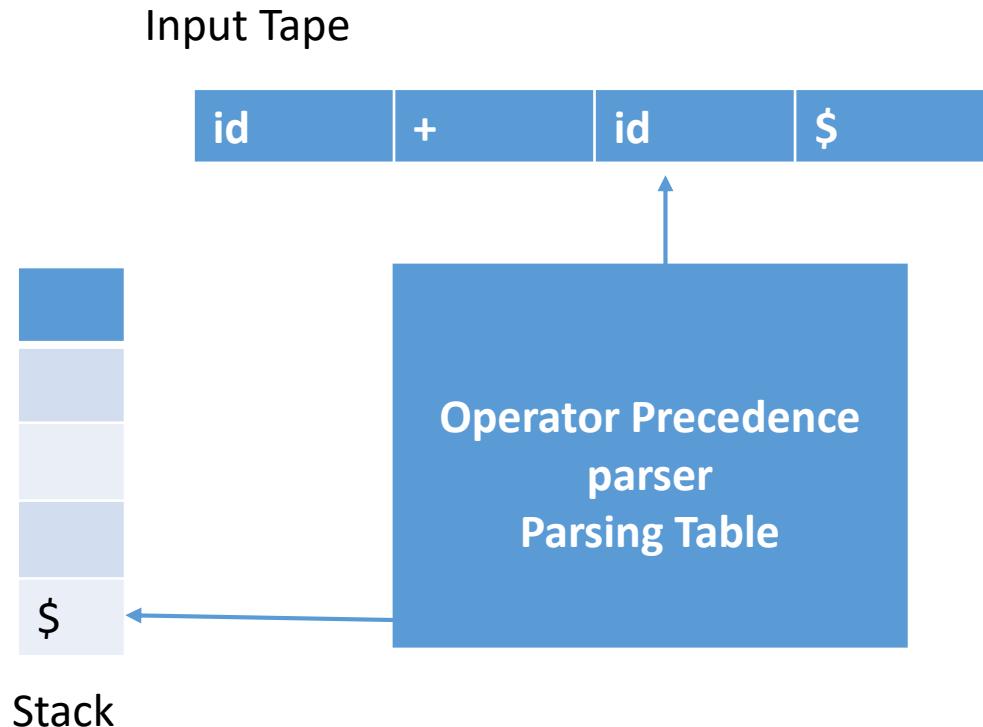
(≈)

	+	*	()	id	\$
+	∨	↖	↖	↘	↖	∨
*	∨	↘	↘	↖	↖	↘
(↖	↖	↖	↖	↖	↖
)	∨	↘	↘	↘	↘	↘
id	∨	↘	↘	↘	↘	↘
\$	↖	↖	↖	↖	↖	↖

Operator Precedence Parsing table

	+	*	()	id	\$
+	∨	↖	↖	∨	↖	∨
*	∨	↘	↘	∨	↘	∨
(↖	↖	↖	↙	↖	
)	∨	↘	↘	∨	↘	∨
id	∨	↘		∨		∨
\$	↖	↖	↖		↖	

Operator Precedence Parsing



Rules for parsing:

1. if \$ is on the top of the stack and \$ is on the input tape, "Accept" the string.
2. if 'a' is on the top of the stack and 'b' is on the input tape:
 - i. If($a \neq b$ or $a < b$), shift b on the stack.
 - ii. If ($a > b$), pop the stack, until the symbol on the top of stack has less precedence than input tape.

Operator Precedence Parsing

Stack	Relation	Input	Action
\$	\triangleleft	id+id*id\$	shift
\$id	\triangleright	+id*id\$	Pop
\$	\triangleleft	+id*id\$	shift
\$+	\triangleleft	id*id\$	shift
\$+id	\triangleright	*id\$	pop
\$+	\triangleleft	*id\$	shift
\$+*	\triangleleft	id\$	shift
\$+*id	\triangleright	\$	pop
\$+*	\triangleright	\$	pop
\$+	\triangleright	\$	pop
\$		\$	

	+	*	()	id	\$
+	\triangleright	\triangleleft	\triangleleft	\triangleright	\triangleleft	\triangleright
*	\triangleright	\triangleright	\triangleleft	\triangleleft	\triangleright	\triangleleft
(\triangleleft	\triangleleft	\triangleleft	\triangleleft	\triangleleft	\triangleleft
)	\triangleright	\triangleright	\triangleright	\triangleright	\triangleright	\triangleright
id	\triangleright	\triangleright			\triangleright	\triangleright
\$	\triangleleft	\triangleleft	\triangleleft		\triangleleft	

Operator Precedence parse table

Suggested Book

Compilers: Principles, Techniques, and Tools

by Aho, Sethi, Ullman and Lam



Compiler Design

Types of LR Parsers

1. Simple LR Parser (SLR)
2. Canonical LR Parser (CLR)
3. Look-ahead LR Parser (LALR)

Handles

- A handle of a right sentential form is a production $A \rightarrow \beta$ & a position γ where the string β may be found and replaced by A to produce the previous right sentential form in a right most derivation of γ .

i.e. if

$S \xrightarrow{*} \alpha A w \xrightarrow{*} \alpha \beta W$ then $A \rightarrow \beta$ in the position following α is handle of $\alpha \beta w$

- Note: the string w to the right of the handle contains only terminal symbols.
If a grammar is unambiguous then every right sentential form of the grammar has exactly one handle.

For the expression grammar,

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

Derive $id+ id*id$

$$E \Rightarrow E+E \Rightarrow E+E*E \Rightarrow E+E*id \Rightarrow E+id*id \Rightarrow id+id*id$$

Rightmost derivation

Sentential form	Handle
$id+id*id \$$	$E \rightarrow id$ at position preceding +
$E+id*id \$$	$E \rightarrow id$ at position following +
$E+E*id \$$	$E \rightarrow id$ at position following *
$E+E*E \$$	$E \rightarrow E*E$ at position following +
$E+E \$$	$E \rightarrow E+E$ at position preceding end marker

Bottom-up parsing is only an attempt to detect the handle of a right sentential form and whenever a handle is detected, the reduction is performed.

This is equivalent to rightmost derivation in reverse, called handle pruning.

Viable Prefixes

- The LR(0) automaton for a grammar characterizes the strings of grammar symbols that can appear on the stack of a shift-reduce parser for the grammar. The stack contents must be a prefix of a right-sentential form.
- If the **stack holds α** and the **rest of the input is x** , then a sequence of reductions will take αx to S .

$$S \xrightarrow[\text{. } rm]{}^* \alpha x$$

- Not all prefixes of right-sentential forms can appear on the stack, however, since the parser must not shift past the handle. For example, suppose

$$E \xrightarrow[\text{rm}]{*} F * \mathbf{id} \Rightarrow (E) * \mathbf{id}$$

- Then, at various times during the parse, the stack will hold $($, $(E$, and (E) , but it must not hold $(E)^*$, since (E) is a handle, which the parser must reduce to F before shifting $*$.
- The prefixes of right sentential forms that can appear on the stack of a shift-reduce parser are called **viable prefixes**.

Or a **viable prefix** is a prefix of a right-sentential form that does not continue past the right end of the rightmost handle of that sentential form.

Problems in SLR parsing

- In SLR parsing method state i calls for reduction on symbol “ a ”, by rule $A \rightarrow \alpha$ if I_i contains $[A \rightarrow \alpha.]$ and “ a ” is in $\text{follow}(A)$.
- However, when state I appears on the top of the stack, the viable prefix $\beta\alpha$ on the stack may be such that βA can not be followed by symbol “ a ” in any right sentential form.
- Thus, the reduction by the rule $A \rightarrow \alpha$ on symbol “ a ” is invalid.

More Powerful LR Parsers

- 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.

Canonical LR(1) Items

- Carry extra information in the state so that wrong reductions by $A \rightarrow \alpha$ will be ruled out
- Redefine LR items to include a terminal symbol as a second component (look ahead symbol).
- The general form of the item becomes $[A \rightarrow \alpha.\beta, a]$ which is called LR(1) item.
- Item $[A \rightarrow \alpha., a]$ calls for reduction only if next input is a. The set of symbols “a”s will be a subset of $\text{Follow}(A)$.

Closure of items

```
SetOfItems CLOSURE( $I$ ) {
    repeat
        for ( each item  $[A \rightarrow \alpha \cdot B\beta, a]$  in  $I$  )
            for ( each production  $B \rightarrow \gamma$  in  $G'$  )
                for ( each terminal  $b$  in FIRST( $\beta a$ ) )
                    add  $[B \rightarrow \cdot \gamma, b]$  to set  $I$ ;
    until no more items are added to  $I$ ;
    return  $I$ ;
}
```

Example

Consider the following grammar

$$\begin{aligned} S' &\rightarrow S \\ S &\rightarrow CC \\ C &\rightarrow cC \mid d \end{aligned}$$

Compute closure(I) where I={[S' → .S, \$]}

$$\begin{array}{ll} S' \rightarrow .S, & \$ \\ S \rightarrow .CC, & \$ \\ C \rightarrow .cC, & c \\ C \rightarrow .cC, & d \\ C \rightarrow .d, & c \\ C \rightarrow .d, & d \end{array}$$

GOTO of Items

```
SetOfItems GOTO( $I, X$ ) {  
    initialize  $J$  to be the empty set;  
    for ( each item  $[A \rightarrow \alpha \cdot X \beta, a]$  in  $I$  )  
        add item  $[A \rightarrow \alpha X \cdot \beta, a]$  to set  $J$ ;  
    return CLOSURE( $J$ );  
}
```

Items

```
void items( $G'$ ) {
    initialize  $C$  to CLOSURE( $\{[S' \rightarrow \cdot S, \$]\}$ );
    repeat
        for ( each set of items  $I$  in  $C$  )
            for ( each grammar symbol  $X$  )
                if ( GOTO( $I, X$ ) is not empty and not in  $C$  )
                    add GOTO( $I, X$ ) to  $C$ ;
    until no new sets of items are added to  $C$ ;
}
```

Example

Consider the following grammar

$$\begin{aligned} S' &\rightarrow S \\ S &\rightarrow CC \\ C &\rightarrow cC \mid d \end{aligned}$$

Compute closure(I) where I={ $S' \rightarrow .S, \$$ }

$$\begin{array}{ll} S' \rightarrow .S, & \$ \\ S \rightarrow .CC, & \$ \\ C \rightarrow .cC, & c \\ C \rightarrow .cC, & d \\ C \rightarrow .d, & c \\ C \rightarrow .d, & d \end{array}$$

The set of LR(1) items

$I_0: S' \rightarrow .S,$
 $S \rightarrow .CC,$
 $C \rightarrow .cC,$
 $C \rightarrow .d,$

\$
\$
c/d
c/d

$I_4: \text{goto}(I_0, d)$
 $C \rightarrow d.,$

c/d

$I_1: \text{goto}(I_0, S)$
 $S' \rightarrow S.,$

\$

$I_5: \text{goto}(I_2, C)$
 $S \rightarrow CC.,$

\$

$I_2: \text{goto}(I_0, C)$
 $S \rightarrow C.C,$
 $C \rightarrow .cC,$
 $C \rightarrow .d,$

\$
\$
\$

$I_6: \text{goto}(I_2, c)$
 $C \rightarrow c.C,$
 $C \rightarrow .cC,$
 $C \rightarrow .d,$

\$
\$
\$

$I_3: \text{goto}(I_0, c)$
 $C \rightarrow c.C,$
 $C \rightarrow .cC,$
 $C \rightarrow .d,$

c/d
c/d
c/d

$I_7: \text{goto}(I_2, d)$
 $C \rightarrow d.,$

\$

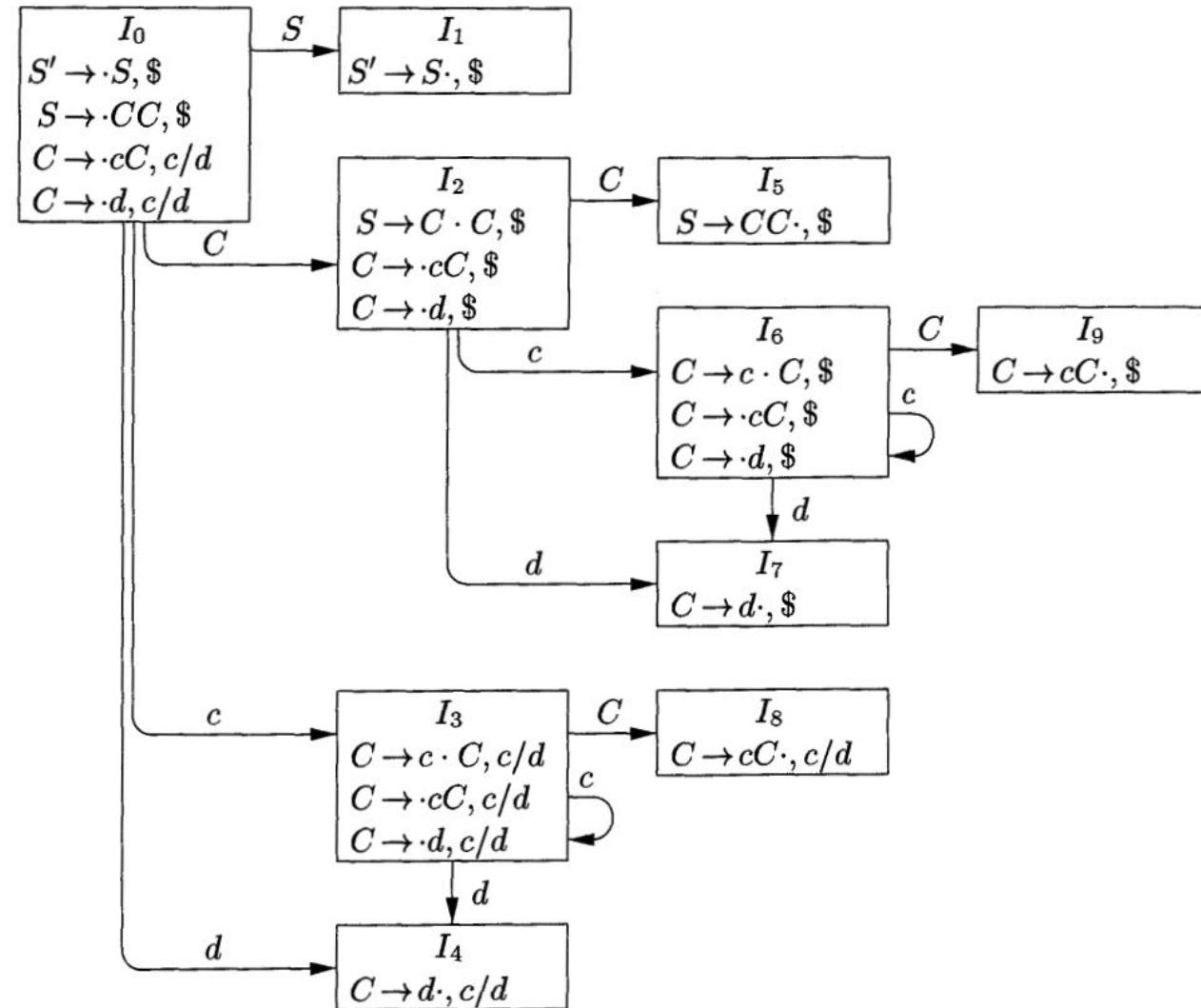
$I_8: \text{goto}(I_3, C)$
 $C \rightarrow cC.,$

c/d

$I_9: \text{goto}(I_6, C)$
 $C \rightarrow cC.,$

\$

GOTO



Construction of Canonical LR parse table

INPUT: An augmented grammar G'

OUTPUT: The canonical-LR parsing table functions ACTION and GOT0 for G' .

METHOD: Construct $C = \{I_0, \dots, I_n\}$ the sets of LR(1) items.

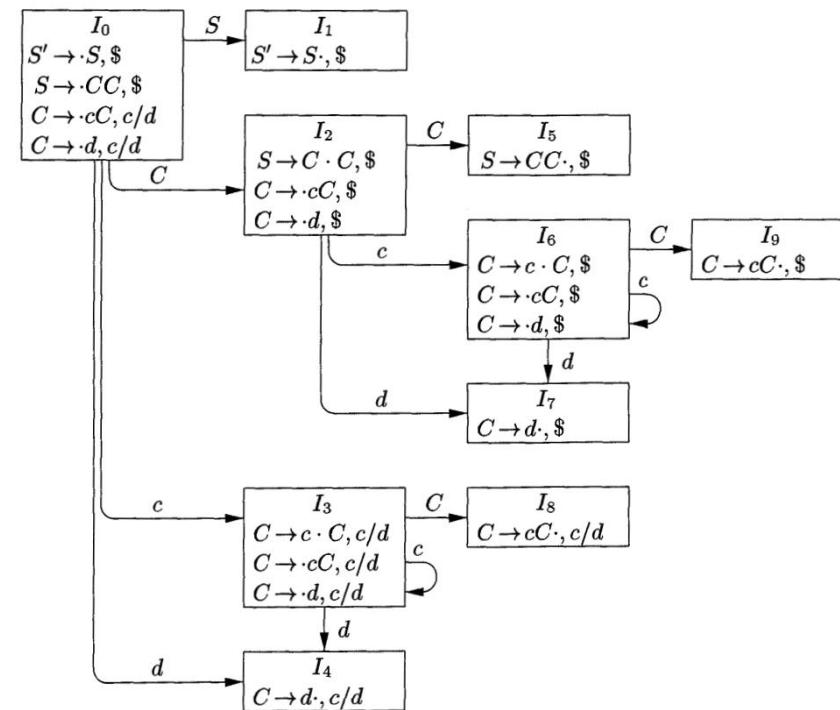
1. Construct $C' = \{I_0, I_1, \dots, I_n\}$, the collection of sets of LR(1) items for G' .
2. State i of the parser is constructed from I_i . The parsing action for state i is determined as follows.
 - (a) If $[A \rightarrow \alpha \cdot a\beta, b]$ is in I_i and $\text{GOTO}(I_i, a) = I_j$, then set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

If any conflicting actions result from the above rules, we say the grammar is not LR(1). The algorithm fails to produce a parser in this case.

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

1. $S \rightarrow CC$
2. $C \rightarrow cC$
3. $C \rightarrow d$

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



1. $S \rightarrow CC$
2. $C \rightarrow cC$
3. $C \rightarrow d$

Stack	Input buffer	Action	Goto	Parsing action
\$0	cdd\$	[0, c]→s3	-	Shift c with state no. 3
\$0c3	dd\$	[3, d]→s4	-	Shift d with state no. 4
\$0c3d4	d\$	[4, d]→r3	-	Reduce $C \rightarrow d$
\$0c3C	d\$	-	[3, C]→8	-
\$0c3C8	d\$	[8, d]→r2	-	Reduce $C \rightarrow cC$
\$0C	d\$		[0, C]→2	Shift d with state no. 4
\$0C2	d\$	[2,d]→s7	-	Shift d with state no. 7
\$0C2d7	\$	[7, \$]→r3	-	Reduce $C \rightarrow d$
\$0C2C	\$	-	[2, C]→5	-
\$0C2C5	\$	[5, \$]→r1	-	Reduce $S \rightarrow CC$
\$0S	\$	-	[0,S]→1	-
\$0S1	\$	[1, \$]→accept	-	-

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

Note: Stack top should always retain state no.

Suggested Book

Compilers: Principles, Techniques, and Tools

by Aho, Sethi, Ullman and Lam



Compiler Design

Types of LR Parsers

1. Simple LR Parser (SLR)
2. Canonical LR Parser (CLR)
3. Look-ahead LR Parser (LALR)

Canonical LR(1) Items

- Carry extra information in the state so that wrong reductions by $A \rightarrow \alpha$ will be ruled out
- Redefine LR items to include a terminal symbol as a second component (look ahead symbol).
- The general form of the item becomes $[A \rightarrow \alpha.\beta, a]$ which is called LR(1) item.
- Item $[A \rightarrow \alpha., a]$ calls for reduction only if next input is a. The set of symbols “a”s will be a subset of $\text{Follow}(A)$.

Closure of items

```
SetOfItems CLOSURE( $I$ ) {
    repeat
        for ( each item  $[A \rightarrow \alpha \cdot B\beta, a]$  in  $I$  )
            for ( each production  $B \rightarrow \gamma$  in  $G'$  )
                for ( each terminal  $b$  in FIRST( $\beta a$ ) )
                    add  $[B \rightarrow \cdot \gamma, b]$  to set  $I$ ;
    until no more items are added to  $I$ ;
    return  $I$ ;
}
```

Example

Consider the following grammar

$$\begin{aligned} S' &\rightarrow S \\ S &\rightarrow CC \\ C &\rightarrow cC \quad | \quad d \end{aligned}$$

Compute closure(I) where I={ $S' \rightarrow .S, \$$ }

$$\begin{array}{ll} S' \rightarrow .S, & \$ \\ S \rightarrow .CC, & \$ \\ C \rightarrow .cC, & c \\ C \rightarrow .cC, & d \\ C \rightarrow .d, & c \\ C \rightarrow .d, & d \end{array}$$

GOTO of Items

```
SetOfItems GOTO( $I, X$ ) {  
    initialize  $J$  to be the empty set;  
    for ( each item  $[A \rightarrow \alpha \cdot X \beta, a]$  in  $I$  )  
        add item  $[A \rightarrow \alpha X \cdot \beta, a]$  to set  $J$ ;  
    return CLOSURE( $J$ );  
}
```

Items

```
void items( $G'$ ) {
    initialize  $C$  to CLOSURE( $\{[S' \rightarrow \cdot S, \$]\}$ );
    repeat
        for ( each set of items  $I$  in  $C$  )
            for ( each grammar symbol  $X$  )
                if ( GOTO( $I, X$ ) is not empty and not in  $C$  )
                    add GOTO( $I, X$ ) to  $C$ ;
    until no new sets of items are added to  $C$ ;
}
```

Example

Consider the following grammar

$$\begin{aligned} S' &\rightarrow S \\ S &\rightarrow CC \\ C &\rightarrow cC \mid d \end{aligned}$$

Compute closure(I) where I={ $S' \rightarrow .S, \$$ }

$$\begin{array}{ll} S' \rightarrow .S, & \$ \\ S \rightarrow .CC, & \$ \\ C \rightarrow .cC, & c \\ C \rightarrow .cC, & d \\ C \rightarrow .d, & c \\ C \rightarrow .d, & d \end{array}$$

The set of LR(1) items

$I_0: S' \rightarrow .S,$
 $S \rightarrow .CC,$
 $C \rightarrow .cC,$
 $C \rightarrow .d,$

\$
\$
c/d
c/d

$I_4: \text{goto}(I_0, d)$
 $C \rightarrow d.,$

c/d

$I_1: \text{goto}(I_0, S)$
 $S' \rightarrow S.,$

\$

$I_5: \text{goto}(I_2, C)$
 $S \rightarrow CC.,$

\$

$I_2: \text{goto}(I_0, C)$
 $S \rightarrow C.C,$
 $C \rightarrow .cC,$
 $C \rightarrow .d,$

\$
\$
\$

$I_6: \text{goto}(I_2, c)$
 $C \rightarrow c.C,$
 $C \rightarrow .cC,$
 $C \rightarrow .d,$

\$
\$
\$

$I_3: \text{goto}(I_0, c)$
 $C \rightarrow c.C,$
 $C \rightarrow .cC,$
 $C \rightarrow .d,$

c/d
c/d
c/d

$I_7: \text{goto}(I_2, d)$
 $C \rightarrow d.,$

\$

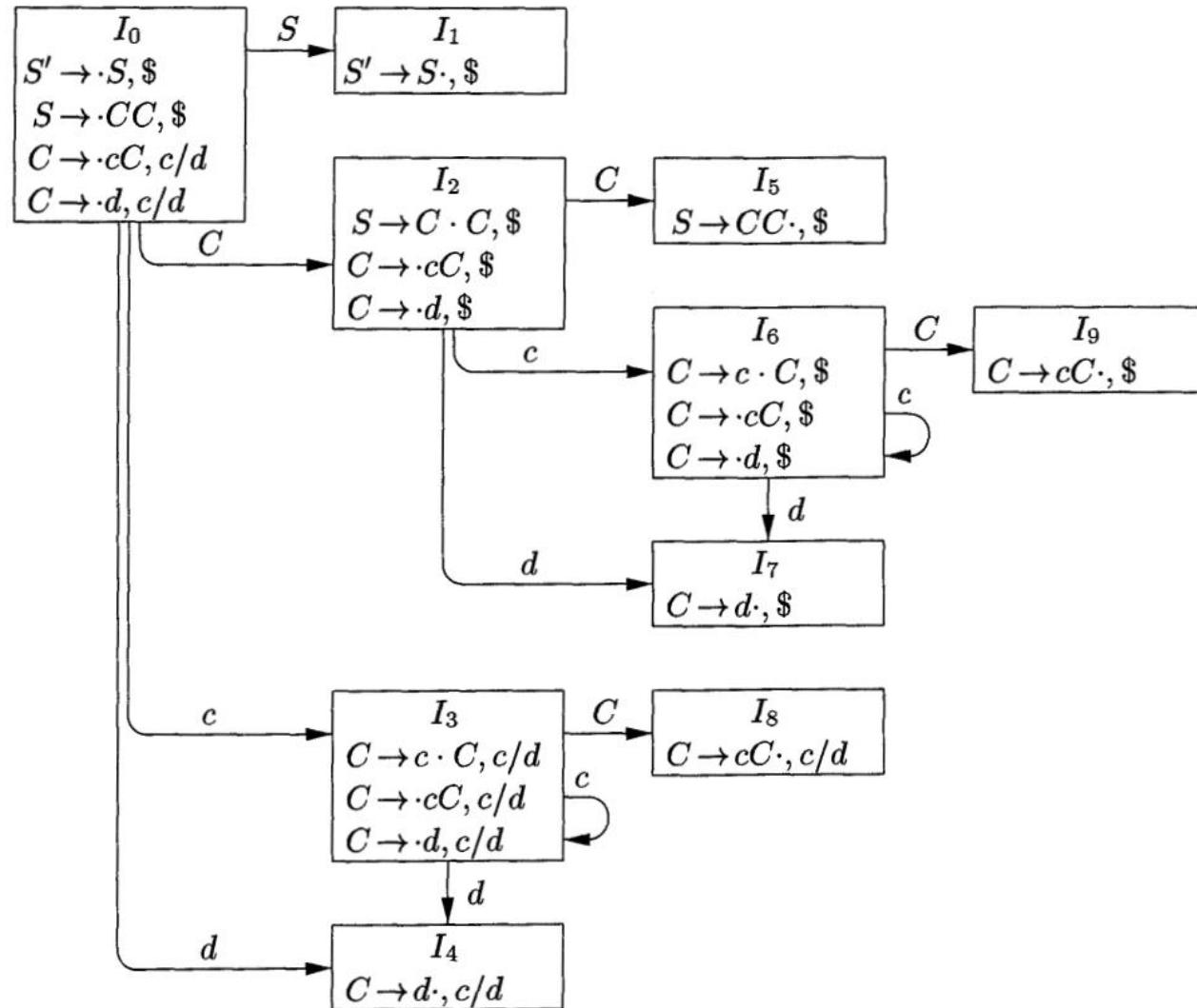
$I_8: \text{goto}(I_3, C)$
 $C \rightarrow cC.,$

c/d

$I_9: \text{goto}(I_6, C)$
 $C \rightarrow cC.,$

\$

GOTO



Construction of Canonical LR parse table

INPUT: An augmented grammar G'

OUTPUT: The canonical-LR parsing table functions ACTION and GOT0 for G' .

METHOD: Construct $C = \{I_0, \dots, I_n\}$ the sets of LR(1) items.

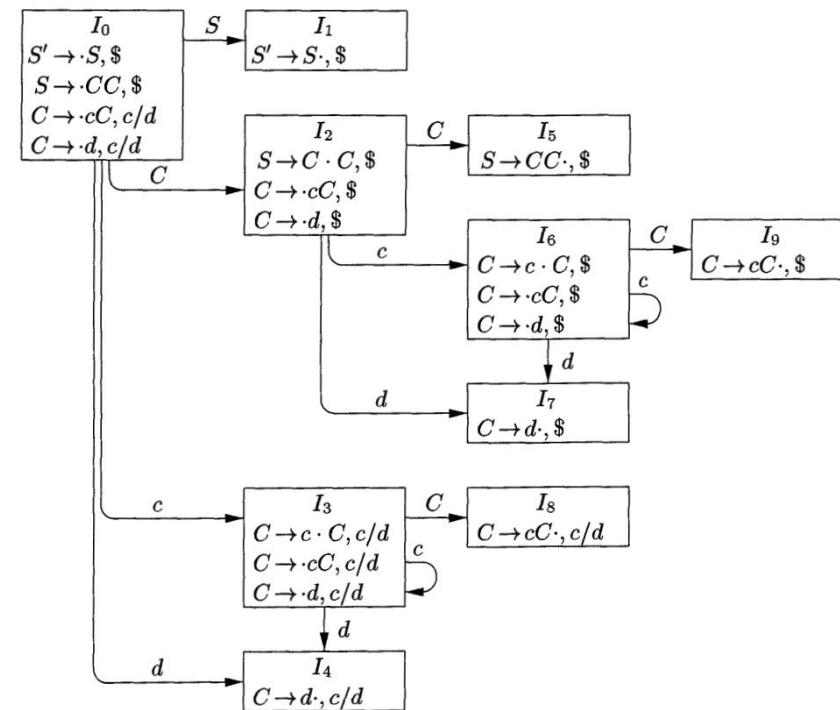
1. Construct $C' = \{I_0, I_1, \dots, I_n\}$, the collection of sets of LR(1) items for G' .
2. State i of the parser is constructed from I_i . The parsing action for state i is determined as follows.
 - (a) If $[A \rightarrow \alpha \cdot a\beta, b]$ is in I_i and $\text{GOTO}(I_i, a) = I_j$, then set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

If any conflicting actions result from the above rules, we say the grammar is not LR(1). The algorithm fails to produce a parser in this case.

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

1. $S \rightarrow CC$
2. $C \rightarrow cC$
3. $C \rightarrow d$

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



1. $S \rightarrow CC$
2. $C \rightarrow cC$
3. $C \rightarrow d$

Stack	Input buffer	Action	Goto	Parsing action
\$0	cdd\$	[0, c]→s3	-	Shift c with state no. 3
\$0c3	dd\$	[3, d]→s4	-	Shift d with state no. 4
\$0c3d4	d\$	[4, d]→r3	-	Reduce $C \rightarrow d$
\$0c3C	d\$	-	[3, C]→8	-
\$0c3C8	d\$	[8, d]→r2	-	Reduce $C \rightarrow cC$
\$0C	d\$		[0, C]→2	Shift d with state no. 4
\$0C2	d\$	[2,d]→s7	-	Shift d with state no. 7
\$0C2d7	\$	[7, \$]→r3	-	Reduce $C \rightarrow d$
\$0C2C	\$	-	[2, C]→5	-
\$0C2C5	\$	[5, \$]→r1	-	Reduce $S \rightarrow CC$
\$0S	\$	-	[0,S]→1	-
\$0S1	\$	[1, \$]→accept	-	-

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

Note: Stack top should always retain state no.

Notes on Canonical LR Parser

- Consider the grammar discussed in the previous two slides. The language specified by the grammar is c^*dc^*d .
- When reading input cc...dcc...d the parser shifts c's into stack and then goes into state 4 after reading d. It then calls for reduction by $C \rightarrow d$ if following symbol is c or d.
- IF \$ follows the first d then input string is c^*d which is not in the language; parser declares an error.
- On an error canonical LR parser never makes a wrong shift/reduce move. It immediately declares an error.
- **Problem:** Canonical LR parse table has a large number of states

LALR Parser

LALR Parse table

- Look Ahead LR parsers.
- Consider a pair of similar looking states (same kernel and different lookaheads) in the set of LR(1) items
 - $I_4 : C \rightarrow d., c/d$ $I_7 : C \rightarrow d., \$$
- Replace I_4 and I_7 by a new state I_{47} consisting of $(C \rightarrow d., c/d/\$)$.
- Similarly $I_3 & I_6$ and $I_8 & I_9$ form pairs .
- Merge LR(1) items having the same core.

The set of LR(1) items

$I_0: S' \rightarrow .S,$
 $S \rightarrow .CC,$
 $C \rightarrow .cC,$
 $C \rightarrow .d,$

\$
\$
c/d
c/d

$I_4: \text{goto}(I_0, d)$
 $C \rightarrow d.,$

c/d

$I_1: \text{goto}(I_0, S)$
 $S' \rightarrow S.,$

\$

$I_5: \text{goto}(I_2, C)$
 $S \rightarrow CC.,$

\$

$I_2: \text{goto}(I_0, C)$
 $S \rightarrow C.C,$
 $C \rightarrow .cC,$
 $C \rightarrow .d,$

\$
\$
\$

$I_6: \text{goto}(I_2, c)$
 $C \rightarrow c.C,$
 $C \rightarrow .cC,$
 $C \rightarrow .d,$

\$
\$
\$

$I_3: \text{goto}(I_0, c)$
 $C \rightarrow c.C,$
 $C \rightarrow .cC,$
 $C \rightarrow .d,$

c/d
c/d
c/d

$I_7: \text{goto}(I_2, d)$
 $C \rightarrow d.,$

\$

$I_8: \text{goto}(I_3, C)$
 $C \rightarrow cC.,$

c/d

$I_9: \text{goto}(I_6, C)$
 $C \rightarrow cC.,$

\$

Algorithm: LALR table construction.

- Construct $C = \{I_0, \dots, I_n\}$ set of LR(1) items.
- For each core present in LR(1) items find all sets having the same core and replace these sets by their union.
- Let $C' = \{J_0, \dots, J_m\}$ be the resulting set of items.
- Construct action table as was done earlier.
- Let $J = I_1 \cup I_2 \cup \dots \cup I_k$ since I_1, I_2, \dots, I_k have same core, $\text{goto}(J, X)$ will have the same core.
- Let $K = \text{goto}(I_1, X) \cup \text{goto}(I_2, X) \cup \dots \cup \text{goto}(I_k, X)$ then $\text{goto}(J, X) = K$

STATE	ACTION			GOTO	
	c	d	\$	S	C
0	s36	s47		1	2
1			acc		
2	s36	s47			5
36	s36	s47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

SLR(1)

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

CLR(1)

Notes on LALR parse table

- Modified parser behaves as original except that it will reduce $C \rightarrow d$ on inputs like ccd. The error will eventually be caught before any more symbols are shifted.
- In general core is a set of LR(0) items and LR(1) grammar may produce more than one set of items with the same core.
- SLR and LALR parse tables have same number of states.
- LALR parser can have new reduce-reduce conflicts
 - Assume states: $\{[X \rightarrow \alpha., a], [Y \rightarrow \beta., b]\}$ and $\{[X \rightarrow \alpha., b], [Y \rightarrow \beta., a]\}$
 - Merging the two states produces: $\{[X \rightarrow \alpha., a/b], [Y \rightarrow \beta., a/b]\}$
- Relative power of various classes –

$$\text{SLR}(1) \leq \text{LALR}(1) \leq \text{LR}(1)$$

$$\text{SLR}(k) \leq \text{LALR}(k) \leq \text{LR}(k)$$

$$\text{LL}(k) \leq \text{LR}(k)$$

Suggested Book

Compilers: Principles, Techniques, and Tools

by Aho, Sethi, Ullman and Lam



Compiler Design

Semantic Analyzer

- Syntax-directed definition
- Annotated parse tree
- Dependency Graphs

Syntax-Directed Definitions

- 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.

CFG+Semantic rules= SDD

Note: attribute may be of value, types, string, etc.

EXAMPLE

```
int a= "sum";
```

Lexical and syntax analysis phase does not generate any error because the line of code is lexically and structurally correct.

It should generate a semantic error as the type of the assignment differs.

Attributes

- We shall deal with two kinds of attributes for nonterminals:
 - Inherited
 - Synthesized

Inherited Attributes

- 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.
- A variable can take values from parents and/or siblings

Eg. $S \rightarrow ABC$

A can take values from S, B and C and so on...

$A.i = S.i$

Synthesized Attributes

- 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.

Eg. $S \rightarrow ABC$

S can take values from A, B and C.

$S.s = A.s$

- Terminals can have synthesized attributes, but not inherited attributes.
- Attributes for terminals have lexical values that are supplied by the lexical analyzer; there are no semantic rules in the SDD itself for computing the value of an attribute for a terminal.

Types of Syntax Directed Translation (SDT)

- S-Attributed SDT
- L-Attributed SDT

S-Attributed SDT

- SDD that only uses **synthesized attributes** is called as S-Attributed SDT.
- The attributes in S-Attributed SDT's are evaluated in bottom-up parsing as the values of parent node depends on the child node.

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

- It evaluates expressions terminated by an endmarker n.
- In the SDD, each of the nonterminals has a single synthesized attribute, called val.
- We also suppose that the terminal digit has a synthesized attribute lexval, which is an integer value returned by the lexical analyzer.
- The rule for **production 1**, $L \rightarrow E \ n$, sets L.val to E.val, which we shall see is the numerical value of the entire expression.
- **Production 2**, $E \rightarrow E_1 + T$, also has one rule, which computes the val attribute for the head E as the sum of the values at E_1 and T.
- **Production 3**, $E \rightarrow T$, has a single rule that defines the value of val for E to be the same as the value of val at the child for T.

L-Attributed SDT

This form of SDT's uses **both synthesized and inherited attributes** with restriction of not taking values from right siblings.

$X \rightarrow ABC$

X can take values from A, B and C.

A can take value from X only.

B can take value from X and A.

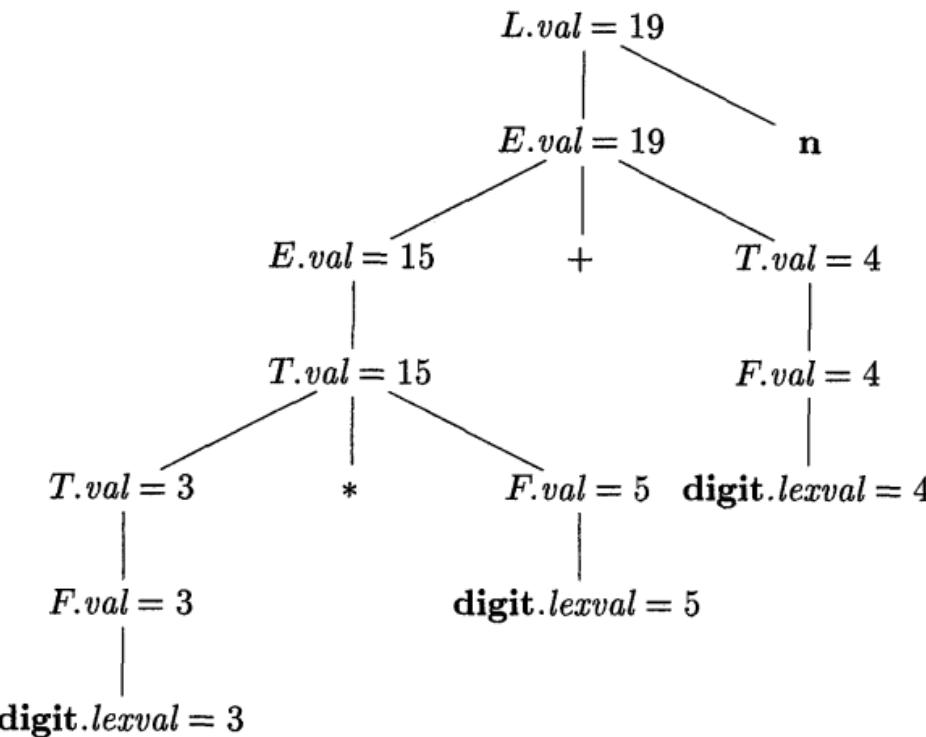
C can take value from X, A and B.

Evaluating SDD at the Nodes of a Parse Tree

- To visualize the translation specified by an SDD, it helps to work with parse trees, even though a translator need not actually build a parse tree.
- A parse tree, showing the value(s) of its attribute(s) is called an **annotated parse tree**.

Annotated parse tree for the input string $3 * 5 + 4 \text{ n}$, constructed using the grammar and rules:

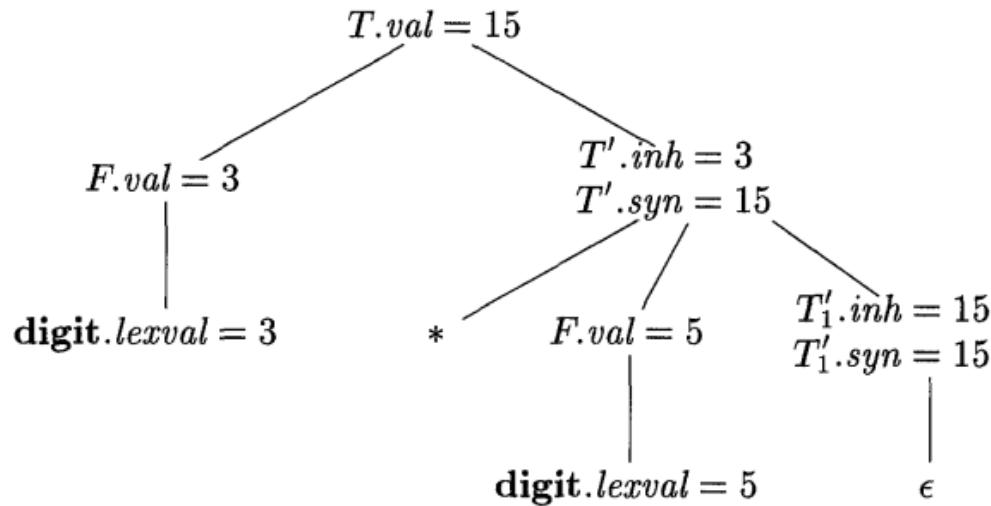
PRODUCTION	SEMANTIC RULES
1) $L \rightarrow E \text{ n}$	$L.val = E.val$
2) $E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
3) $E \rightarrow T$	$E.val = T.val$
4) $T \rightarrow T_1 * F$	$T.val = T_1.val \times F.val$
5) $T \rightarrow F$	$T.val = F.val$
6) $F \rightarrow (E)$	$F.val = E.val$
7) $F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



Annotated parse tree

Computes $3 * 5$ using grammar and rules:

PRODUCTION	SEMANTIC RULES
1) $T \rightarrow F T'$	$T'.inh = F.val$ $T.val = T'.syn$
2) $T' \rightarrow * F T'_1$	$T'_1.inh = T'.inh \times F.val$ $T'.syn = T'_1.syn$
3) $T' \rightarrow \epsilon$	$T'.syn = T'.inh$
4) $F \rightarrow \text{digit}$	$F.val = \text{digit}.lexval$

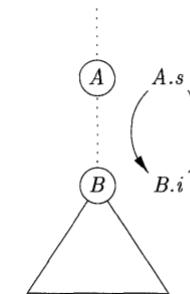


Annotated parse tree for $3 * 5$

- For SDD's with both inherited and synthesized attributes, there is no guarantee that **there is even one order in which to evaluate attributes at nodes**.
- For instance, consider nonterminals A and B, with synthesized and inherited attributes $A.s$ and $B.i$, respectively, along with the production and rules:

PRODUCTION
 $A \rightarrow B$

SEMANTIC RULES
 $A.s = B.i;$
 $B.i = A.s + 1$



- It is computationally difficult to determine **whether or not there exist any circularities** in any of the parse trees that a given SDD could have to translate.

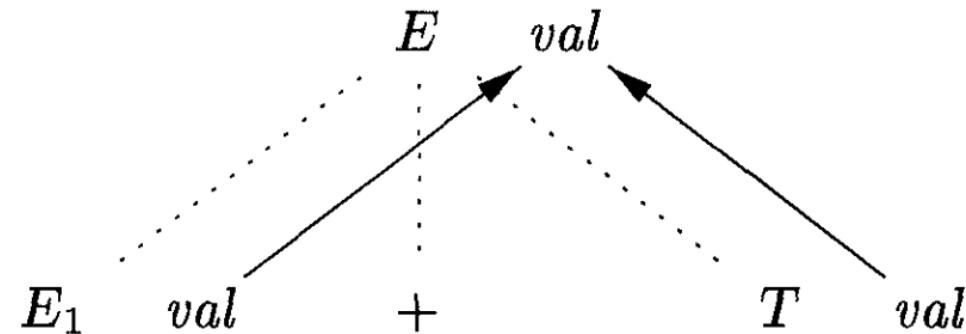
Dependency Graphs

- A dependency graph depicts the **flow of information** among the attribute instances in a particular parse tree; **an edge from one attribute instance to another means that the value of the first is needed to compute the second.**
 - For each parse-tree node, say a node labeled by grammar symbol X, the dependency graph has a node for each attribute associated with X.
 - A semantic rule associated with a production p defines the value of **synthesized attribute A.b** in terms of the value of X.c. . Then, the dependency graph has **an edge from X.c to A.b**
 - A semantic rule associated with a production p defines the value of inherited attribute B.c in terms of the value of X.a. Then, **the dependency graph has an edge from X.a to B.c**

Example: Consider the following production and rule:

PRODUCTION
 $E \rightarrow E_1 + T$

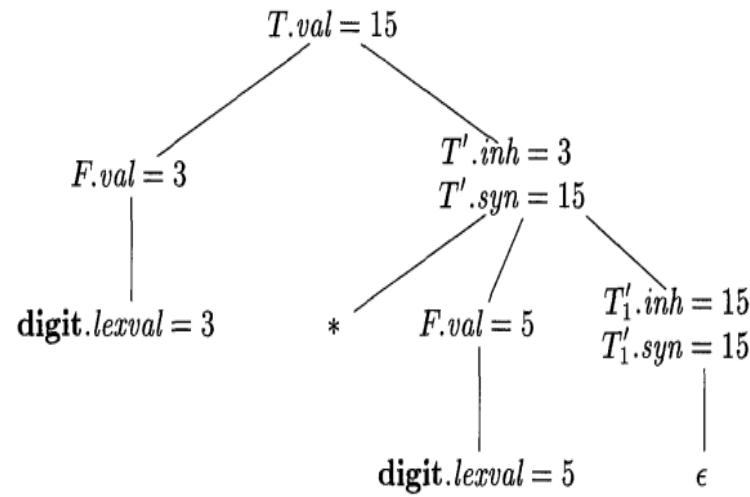
SEMANTIC RULE
 $E.val = E_1.val + T.val$



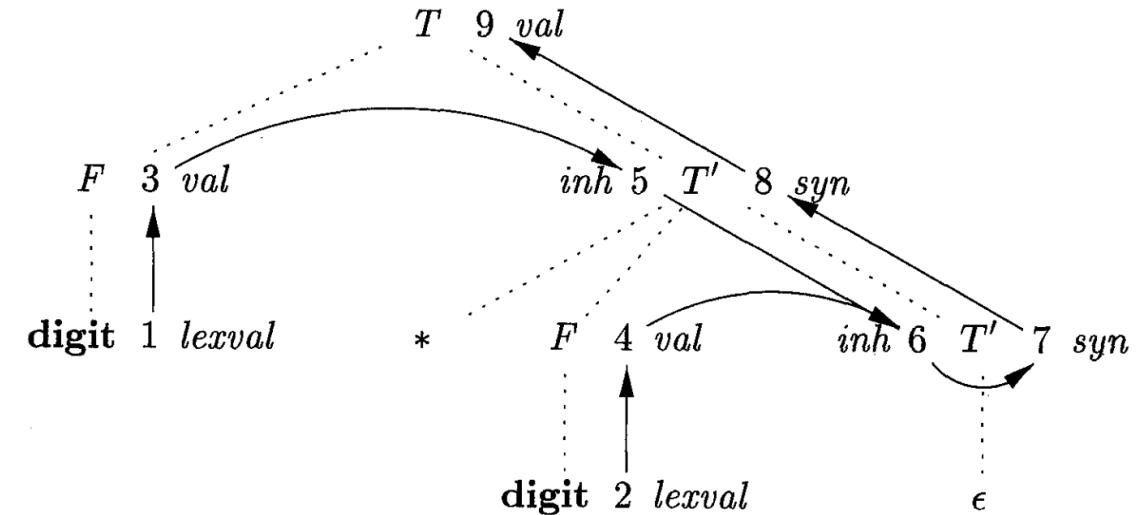
$E.val$ is synthesized from $E_1.val$ and $E_2.val$

An example of a complete dependency graph

PRODUCTION	SEMANTIC RULES
1) $T \rightarrow F T'$	$T'.inh = F.val$ $T.val = T'.syn$
2) $T' \rightarrow * F T'_1$	$T'_1.inh = T'.inh \times F.val$ $T'.syn = T'_1.syn$
3) $T' \rightarrow \epsilon$	$T'.syn = T'.inh$
4) $F \rightarrow \text{digit}$	$F.val = \text{digit}.lexval$



i. Annotated parse tree for $3 * 5$



ii. Dependency Graph

Note: The nodes of the dependency graph, represented by the numbers 1 through 9, correspond to the attributes in the annotated parse tree (i)

Suggested Book

Compilers: Principles, Techniques, and Tools

by Aho, Sethi, Ullman and Lam



Compiler Design

Intermediate-Code Generation

- Variants of Syntax Trees
- Three address code

Intermediate-Code Generation

- In the analysis-synthesis model of a compiler, the front end analyzes a source program and creates an intermediate representation, from which the back end generates target code.
- With a suitably defined intermediate representation, a compiler for language i and machine j can then be built by combining the front end for language i with the back end for machine j.
- This approach to creating suite of compilers can save a considerable amount of effort: $m \times n$ compilers can be built by writing just m front ends and n back ends.

In the process of translating a program in a given source language into code for a given target machine, a compiler may construct a **sequence of intermediate representations**.



High-level representations are close to the source language and **low-level representations** are close to the target machine.

Syntax trees are high level; they depict the natural hierarchical structure of the source program and are well suited to tasks like static type checking.

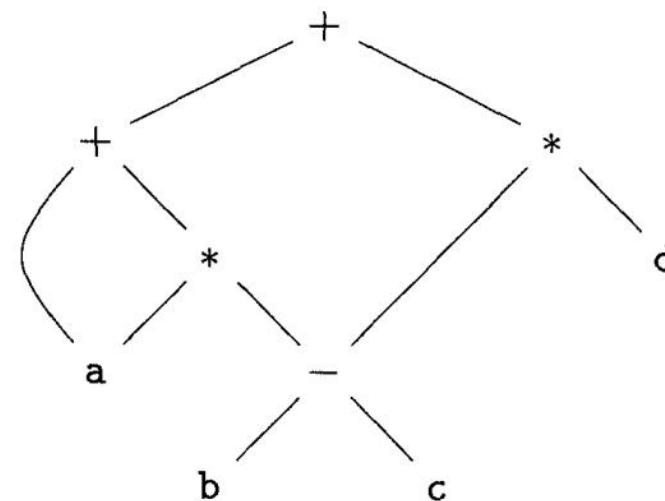
A low-level representation is suitable for machine-dependent tasks like register allocation and instruction selection.

Directed Acyclic Graphs for Expressions

- Nodes in a syntax tree represent constructs in the source program; the children of a node represent the meaningful components of a construct.
- A **directed acyclic graph** (hereafter called a DAG) for an expression identifies the **common subexpressions** (subexpressions that occur more than once) of the expression.

- Like the syntax tree for an expression, a DAG has leaves corresponding to atomic operands and interior codes corresponding to operators.
- The difference is that a node N in a DAG has **more than one parent** if N represents a common subexpression; in a syntax tree, the tree for the common subexpression would be replicated.
- A DAG not only represents expressions more succinctly, it gives the compiler important clues regarding the generation of **efficient code** to evaluate the expressions.

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



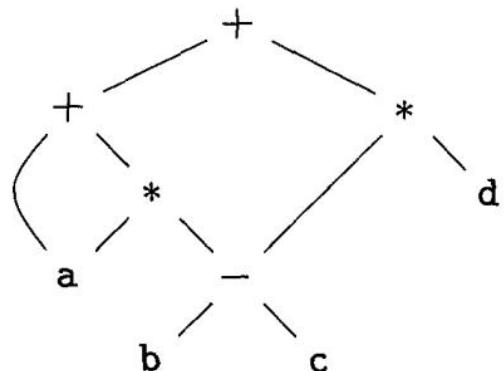
Three-Address Code

- In three-address code, there is **at most one operator on the right side** of an instruction; that is, no built-up arithmetic expressions are permitted.
- Thus a source-language expression like $x+y*z$ might be translated into the sequence of three-address instructions

$$t_1 = y * z$$
$$t_2 = x + t_1$$

Example:

- Three-address code is a linearized representation of a syntax tree or a DAG in which explicit names correspond to the interior nodes of the graph



(a) DAG

$$\begin{aligned}t_1 &= b - c \\t_2 &= a * t_1 \\t_3 &= a + t_2 \\t_4 &= t_1 * d \\t_5 &= t_3 + t_4\end{aligned}$$

(b) Three-address code

A DAG and its corresponding three-address code

Three-Address Code

- The description of three-address instructions specifies the components of each type of instruction, but it **does not specify** the representation of these instructions in **a data structure**.
- In a compiler, these instructions can be implemented as objects or as records with **fields for the operator and the operands**. Three such representations are called "quadruples," "triples" and "indirect triples."
 - Quadruples
 - Triples
 - Indirect triples

Quadruples

- A quadruple (or just "quad") has four fields, which we call **op**, **arg1**, **arg2**, and **result**. The op field contains an internal code for the operator.
- For instance, the three-address instruction $x = y + z$ is represented by placing **+** in op, **y** in arg1, **z** in arg2, and **x** in result.

Example

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

(a) Three-address code

	<i>op</i>	<i>arg₁</i>	<i>arg₂</i>	<i>result</i>
0	minus	c		t ₁
1	*	b	t ₁	t ₂
2	minus	c		t ₃
3	*	b	t ₃	t ₄
4	+	t ₂	t ₄	t ₅
5	=	t ₅		a
				...

(b) Quadruples

Triple

- A triple has only three fields, which we call **op**, **arg1**, and **arg2**.
- Using triples, we refer to the result of an operation $x \text{ op } y$ by its position, rather than by an explicit temporary name.

Example:

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

(a) Three-address code

	<i>op</i>	<i>arg₁</i>	<i>arg₂</i>	<i>result</i>
0	minus	c		t ₁
1	*	b	t ₁	t ₂
2	minus	c		t ₃
3	*	b	t ₃	t ₄
4	+	t ₂	t ₄	t ₅
5	=	t ₅		a
		...		

(b) Quadruples

	<i>op</i>	<i>arg₁</i>	<i>arg₂</i>
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
		...	

(c) Triples

Indirect triples

- Indirect triples consist of a listing of pointers to triples, rather than a listing of triples themselves

instruction

35	(0)
36	(1)
37	(2)
38	(3)
39	(4)
40	(5)
	...

op arg₁ arg₂

0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
			...

- A benefit of quadruples over triples can be seen in an optimizing compiler, where instructions are often moved around. With quadruples, if we move an instruction that computes a temporary t , then the instructions that use t require no change.
- With triples, the result of an operation is referred to by its position, so moving an instruction may require us to change all references to that result. This problem does not occur with indirect triples.
- Indirect triples consist of a listing of pointers to triples, rather than a listing of triples themselves.

Suggested Book

Compilers: Principles, Techniques, and Tools

by Aho, Sethi, Ullman and Lam



Compiler Design

Code Generation

- The final phase in our compiler model is the code generator.
- It takes as input the **intermediate representation** (IR) produced by the front end of the compiler, along with relevant symbol table information, and produces as output a **semantically equivalent target program**.

Basic Blocks

- Our first job is to partition a **sequence of three-address instructions into basic blocks**.
- We begin a new basic block with the first instruction and keep adding instructions **until we meet either a jump, a conditional jump, or a label** on the following instruction.
- In the absence of jumps and labels, control proceeds sequentially from one instruction to the next.

Partitioning three-address instructions into basic blocks.

INPUT: A sequence of three-address instructions.

OUTPUT: A list of the basic blocks for that sequence in which each instruction is assigned to exactly one basic block.

METHOD: First, we determine those instructions in the intermediate code that are leaders, that is, the first instructions in some basic block. The instruction just past the end of the intermediate program is not included as a leader. The rules for finding leaders are:

1. The first three-address instruction in the intermediate code is a leader.
2. Any instruction that is the target of a conditional or unconditional jump is a leader.
3. Any instruction that immediately follows a conditional or unconditional jump is a leader

Then, for each leader, its basic block consists of itself and all instructions up to but not including the next leader or the end of the intermediate program

Basic Blocks

- 1) $i = 1$
- 2) $j = 1$
- 3) $t_1 = 10 * i$
- 4) $t_2 = t_1 + j$
- 5) $t_3 = 8 * t_2$
- 6) $t_4 = t_3 - 88$
- 7) $a[t_4] = 0.0$
- 8) $j = j + 1$
- 9) $\text{if } j \leq 10 \text{ goto (3)}$
- 10) $i = i + 1$
- 11) $\text{if } i \leq 10 \text{ goto (2)}$
- 12) $i = 1$
- 13) $t_5 = i - 1$
- 14) $t_6 = 88 * t_5$
- 15) $a[t_6] = 1.0$
- 16) $i = i + 1$
- 17) $\text{if } i \leq 10 \text{ goto (13)}$

B_1 $i = 1$

B_2 $j = 1$

B_3

```
t1 = 10 * i
t2 = t1 + j
t3 = 8 * t2
t4 = t3 - 88
j = j + 1
if j <= 10 goto B3
```

B_4

```
i = i + 1
if i <= 10 goto B2
```

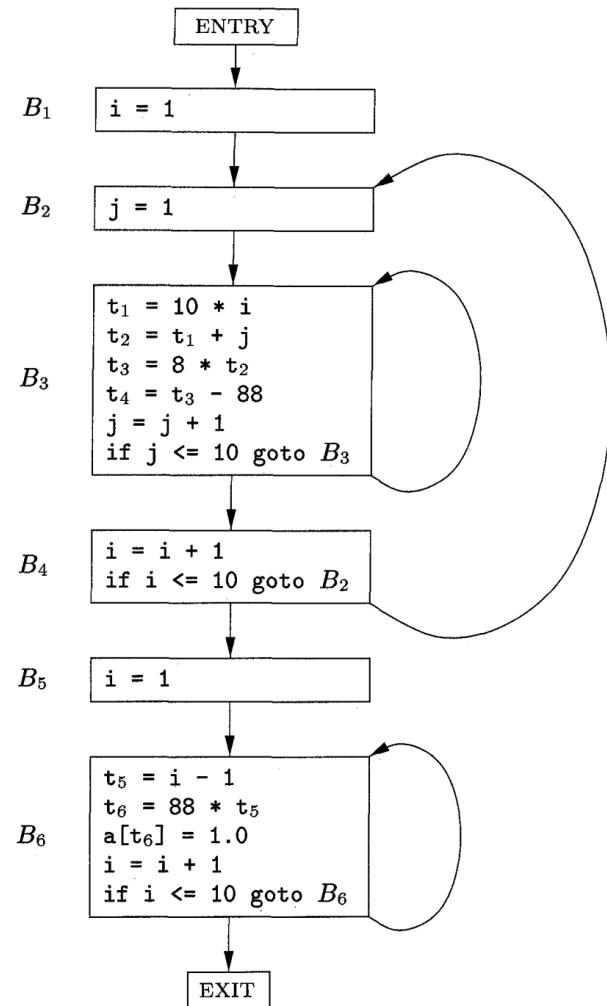
B_5 $i = 1$

B_6

```
t5 = i - 1
t6 = 88 * t5
a[t6] = 1.0
i = i + 1
if i <= 10 goto B6
```

Flow Graphs

- Once an intermediate-code program is partitioned into basic blocks, we represent the flow of control between them by a flow graph.
- The nodes of the flow graph are the basic blocks.
- There is an **edge from block B to block C** if and only if it is possible for the **first instruction in block C to immediately follow the last instruction in block B**.
- There are two ways that such an edge could be justified:
 - There is a conditional or unconditional jump from the end of B to the beginning of C.
 - C immediately follows B in the original order of the three-address instructions, and B does not end in an unconditional jump.



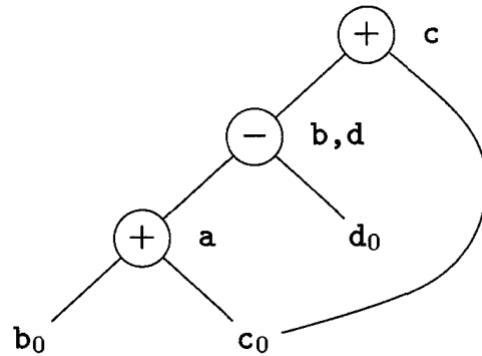
Optimization of Basic Blocks

- Finding Local Common Subexpressions
- Dead Code Elimination
- The Use of Algebraic Identities

Example: A DAG for the block

```
a = b + c  
b = a - d  
c = b + c  
d = a - d
```

- When we construct the node for the third statement $c = b + c$, we know that the use of b in $b + c$ refers to the node labeled at $-$, because that is the most recent definition of b .
- Thus, we do not confuse the values computed at statements one and three.



- The block then becomes

```
a = b + c  
d = a - d  
c = d + c
```

Dead Code Elimination

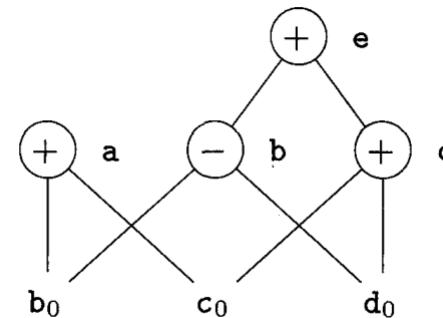
- The operation on DAG's that corresponds to dead-code elimination can be implemented as follows:
 - We delete from a DAG any root (node with no ancestors) that has no live variables attached.
 - Repeated application of this transformation will remove all nodes from the DAG that correspond to dead code.

- The DAG for this sequence is shown in Fig. , but does not exhibit any common subexpressions. However, algebraic identities applied to the DAG, may expose the equivalence.

```

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

```



- even though b and c both change between the first and last statements, their sum remains the same, because $e = b + c = (b - d) + (c + d)$.

The Use of Algebraic Identities

- Algebraic identities represent another important class of optimizations on basic blocks.
- For example, we may apply arithmetic identities, such as

$$x + 0 = 0 + x = x$$

$$x \times 1 = 1 \times x = x$$

$$x - 0 = x$$

$$x/1 = x$$

- to eliminate computations from a basic block

Local Reduction In Strength

- Another class of algebraic optimizations includes local reduction in strength, that is, replacing a more expensive operator by a cheaper one as in:

EXPENSIVE	=	CHEAPER
x^2	=	$x \times x$
$2 \times x$	=	$x + x$
$x/2$	=	$x \times 0.5$

Constant Folding

- A third class of related optimizations is constant folding.
- Here we evaluate constant expressions at compile time and replace the constant expressions by their value.
- Thus the expression $2 * 3.14$ would be replaced by 6.28.
- Many constant expressions arise in practice because of the frequent use of symbolic constants in programs.

- The DAG-construction process can help us apply these and other more general algebraic transformations such as **commutativity** and **associativity**.
- For example, the **condition** $x > y$ can also be tested by **subtracting the arguments** and performing a test on the **condition** code set by the subtraction.
- Thus, only one node of the DAG may need to be generated for $x - y$ and $x > y$.

- Associative laws might also be applicable to expose common subexpressions. For example, if the source code has the assignments

$$\begin{aligned}a &= b + c; \\e &= c + d + b;\end{aligned}$$

- the following intermediate code might be generated:

$$\begin{aligned}a &= b + c \\t &= c + d \\e &= t + b\end{aligned}$$

- If t is not needed outside this block, we can change this sequence to

$$\begin{aligned}a &= b + c \\e &= a + d\end{aligned}$$

Assignment

- Type checking and type conversion
- storage organization, activation tree, activation record, symbol table: hashing, linked list, tree structures.
- Optimization: Representation of Array References
- Optimization: Peephole Optimization
- Suggested book: A. V. Aho, J. D. Ullman, *Principles of Compiler Design*, Narosa Publishing House.

Suggested Book

Compilers: Principles, Techniques, and Tools

by Aho, Sethi, Ullman and Lam