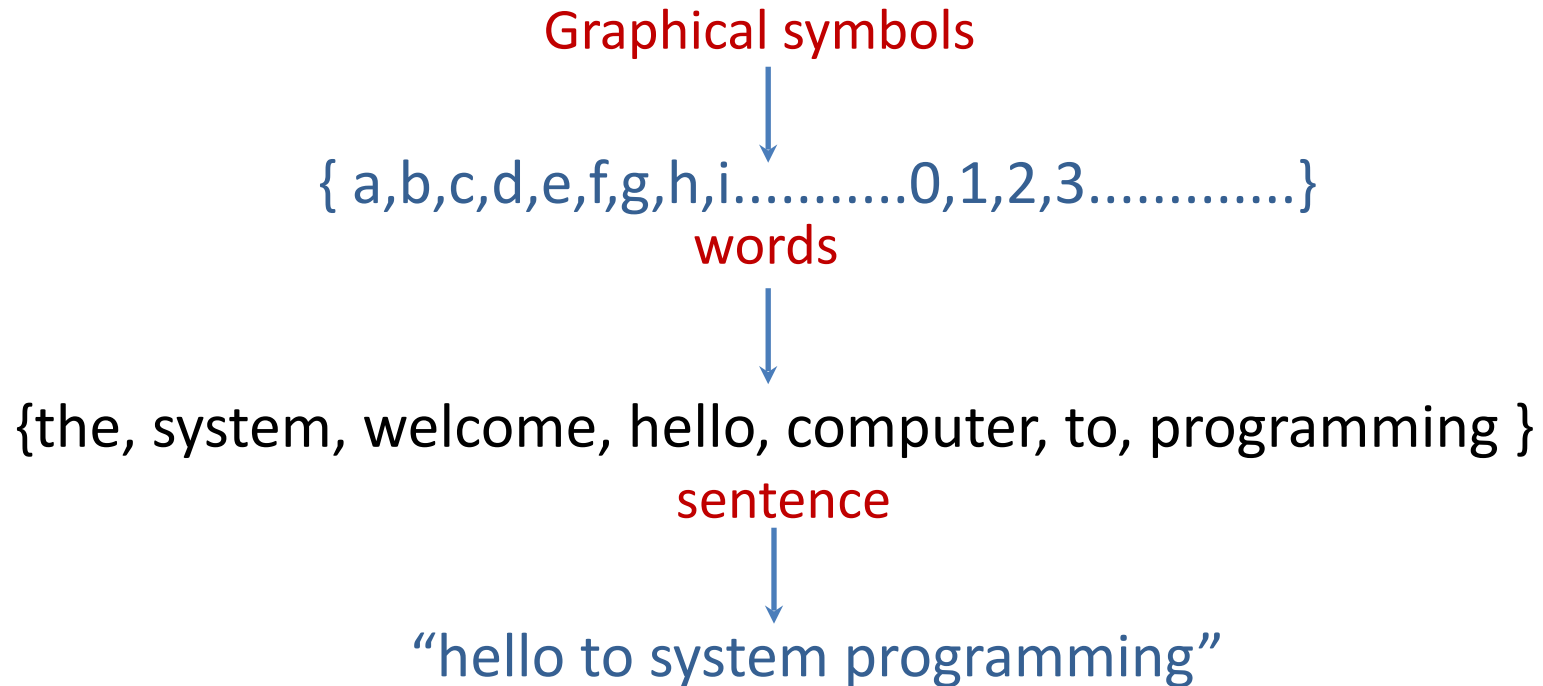


Compilers - Analysis Phase

Programming language grammar

Formal language

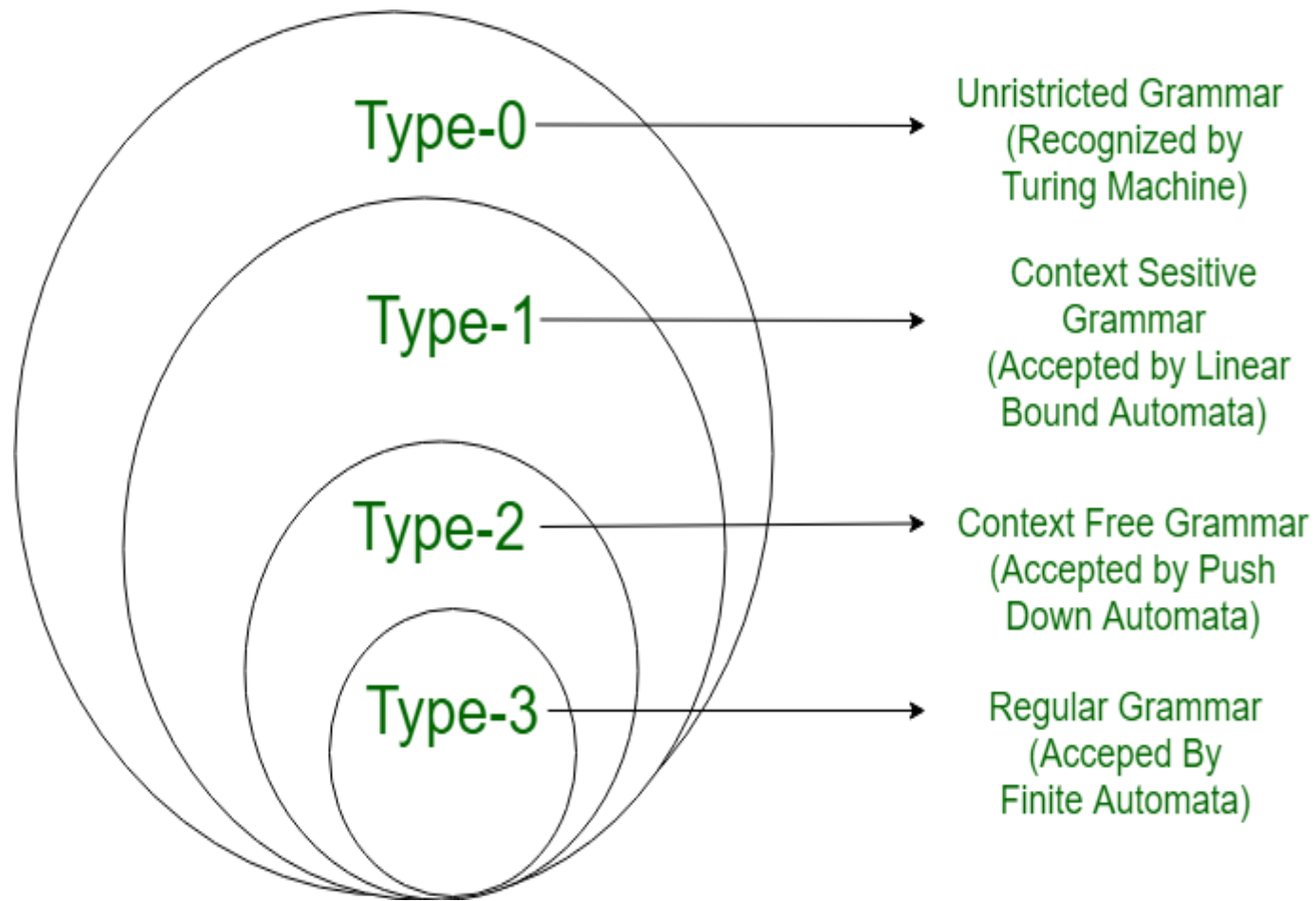


- A **formal language** is a collection of valid sentences, where each **sentence** is a sequence of **words**, and each word is a sequence of **graphic symbols** acceptable in a language.
- Set of **rules** that specify the construction of words and sentences is called **formal language grammar**.

Grammar

- A grammar G of a language L_G is a quadruple (Σ, SNT, S, P) where
 - Σ is the alphabet of L_G , i.e. the set of terminal symbols
 - SNT is the set of nonterminal symbols
 - S is the start symbol
 - P is the set of productions

Classification of grammar (Chomsky hierarchy)



Derivation

Grammar : $\langle \text{Noun Phrase} \rangle \rightarrow \langle \text{Article} \rangle \langle \text{Noun} \rangle$

$\langle \text{Article} \rangle \rightarrow a \mid an \mid the$

$\langle \text{Noun} \rangle \rightarrow boy \mid apple$

$\langle \text{Noun Phrase} \rangle \rightarrow \underline{\langle \text{Article} \rangle} \langle \text{Noun} \rangle$

$\rightarrow the \underline{\langle \text{Noun} \rangle}$

$\rightarrow the \text{ boy}$

- Let production P_1 of grammar G be of the form $P_1: A \rightarrow \alpha$ and let β be a string such that $\beta = \gamma A \theta$, then replacement of A by α in string β constitutes a derivation according to production P_1 .
- There are two types of derivation:
 1. Leftmost derivation
 2. Rightmost derivation

Reduction

Grammar : $\langle \text{Noun Phrase} \rangle \rightarrow \langle \text{Article} \rangle \langle \text{Noun} \rangle$

$\langle \text{Article} \rangle \rightarrow a \mid an \mid the$

$\langle \text{Noun} \rangle \rightarrow boy \mid apple$

$\rightarrow \underline{the} boy$

$\rightarrow \langle \text{Article} \rangle \underline{boy}$

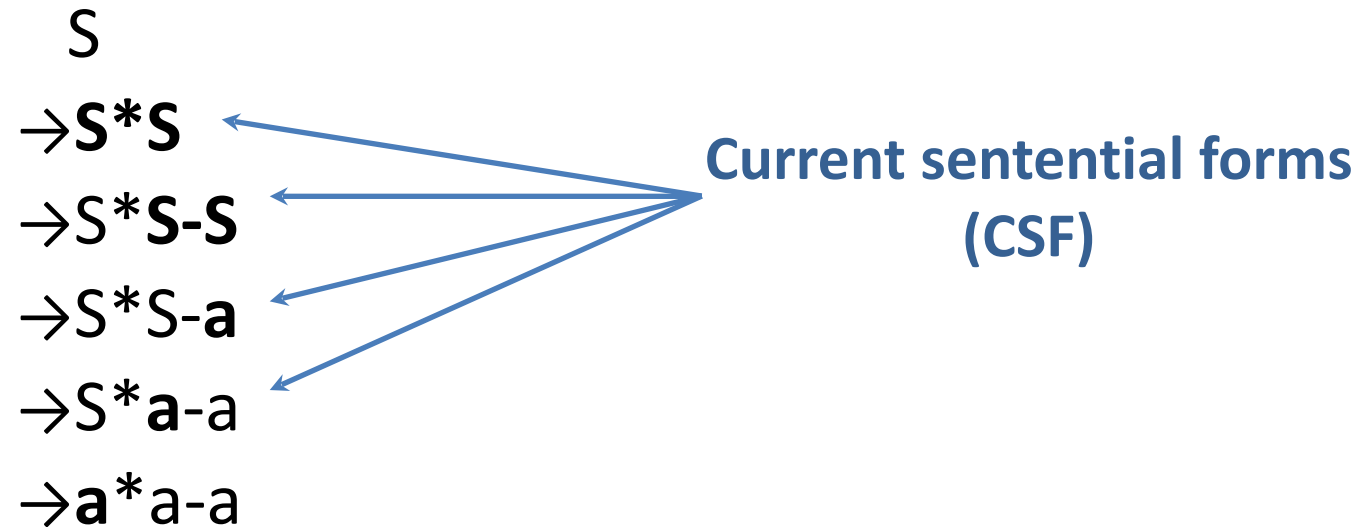
$\rightarrow \underline{\langle \text{Article} \rangle \langle \text{Noun} \rangle}$

$\langle \text{Noun Phrase} \rangle$

- Let production P_1 of grammar G be of the form $P_1: A \rightarrow \alpha$ and let σ be a string such that $\sigma \rightarrow \gamma\alpha\theta$, then replacement of α by A in string σ constitutes a reduction according to production P_1 .

Current sentential form

- Current sentential form is any string derivable from start symbol.
- Grammar: $S \rightarrow S+S \mid S-S \mid S*S \mid S/S \mid a$ Output string: $a*a-a$



Ambiguity

- In formal language grammar, ambiguity would arise if identical string can occur on the RHS of two or more productions.

- Grammar:

$$N_1 \rightarrow \alpha$$

$$N_2 \rightarrow \alpha$$



- α can be derived from either N_1 or N_2

Regular expression

- A regular expression is a sequence of characters that **define a pattern**.
- **Notational shorthand's**
 1. One or more occurrences: $+$
 2. Zero or more occurrences: $*$
 3. Alphabets: Σ
- Regular Expression is mainly for use in pattern matching with strings, or string matching, i.e. "find and replace"-like operations.

Finite State Automata

Finite state automata

- A finite state automata is a triple (S, Σ, T) where
 - S : is a finite set of states
 - one of which is the initial state s_{init}
 - one or more of which are the final states.
 - Σ : is the input symbols.
 - T : is a finite set of state transitions defining transitions out of states in S on encountering symbols in Σ .

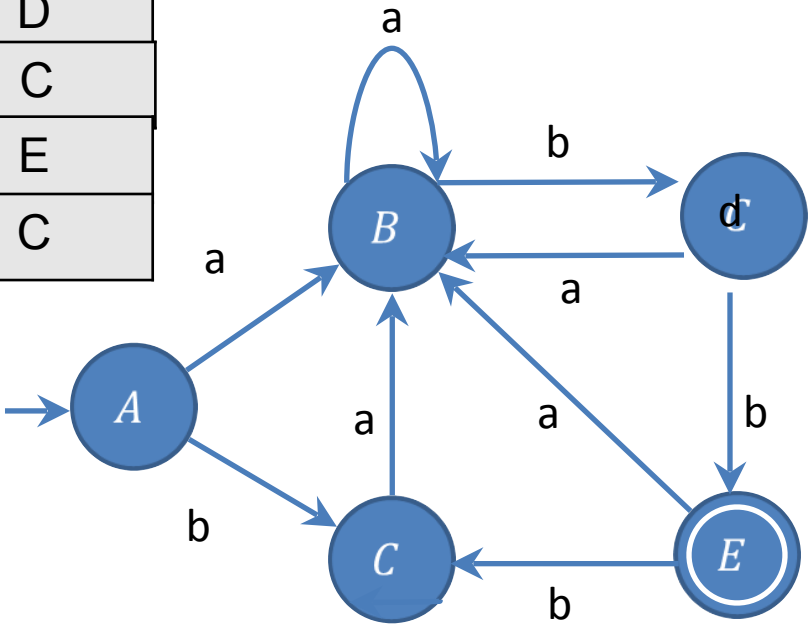
Finite state automata

- Deterministic finite state automaton:
 - It is a finite state automaton none of whose states has two or more transitions for the same source symbol.
 - The DFA has the property that it reaches a unique state for every source string input to it.
- Non Deterministic finite state automaton:
 - No restriction on edges leaving states.
 - There can be several with the same symbol as label and some edges can be labeled as ϵ .

DFA

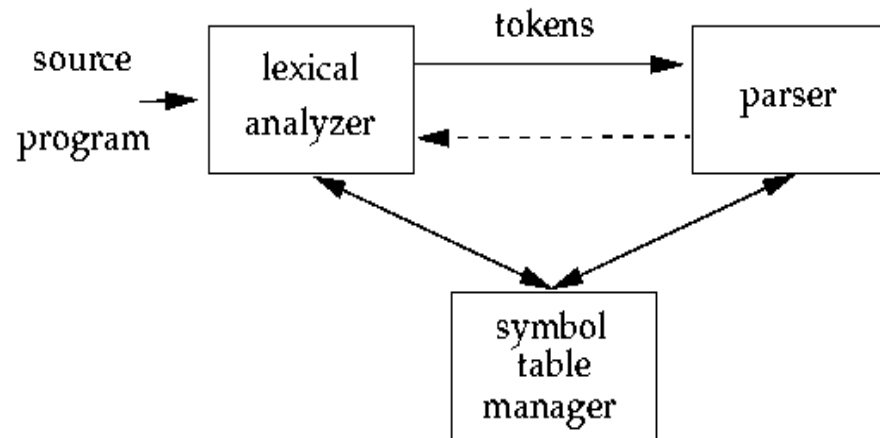
States	a	b
A	B	C
B	B	D
C	B	C
D	B	E
E	B	C

Transition Table



DFA

Lexical Analysis



- Main task: to read input characters and group them into “*tokens*” – (Constants, identifiers, keywords etc.)
- Secondary tasks:
 - Skip comments and white space;
 - Correlate error messages with source program (e.g., line number of error).

Lexical Analysis: Terminology

- token: a name for a set of input strings with related structure.

Example: “identifier,” “integer constant”

- pattern: a rule describing the set of strings associated with a token.

Example: “a letter followed by zero or more letters, digits, or underscores.”

- lexeme: the actual input string that matches a pattern.

Example: count

- Attribute of Tokens

Example: count = 123

Specifying Tokens: regular expressions

- Terminology:

alphabet : a finite set of symbols

string : a finite sequence of alphabet symbols

language : a (finite or infinite) set of strings.

- Regular Operations on languages:

Union: $R \cup S = \{ x \mid x \in R \text{ or } x \in S \}$

Concatenation: $RS = \{ xy \mid x \in R \text{ and } y \in S \}$

Kleene closure: $R^* = R$ concatenated with itself 0 or more times

$= \{\epsilon\} \cup R \cup RR \cup RRR \cup$

$=$ strings obtained by concatenating a finite number of strings from the set R .

Regular Expressions

A pattern notation for describing certain kinds of sets over strings:

Given an alphabet Σ :

- ε is a regular exp. (denotes the language $\{\varepsilon\}$)
- for each $a \in \Sigma$, a is a regular exp. (denotes the language $\{a\}$)
- if \mathbf{r} and \mathbf{s} are regular exps. denoting $L(\mathbf{r})$ and $L(\mathbf{s})$ respectively, then so are:
 - $\mathbf{r} \mid \mathbf{s}$ (denotes the language $L(\mathbf{r}) \cup L(\mathbf{s})$)
 - $\mathbf{r}(\mathbf{s})$ (denotes the language $L(\mathbf{r})L(\mathbf{s})$)
 - \mathbf{r}^* (denotes the language $L(\mathbf{r})^*$)

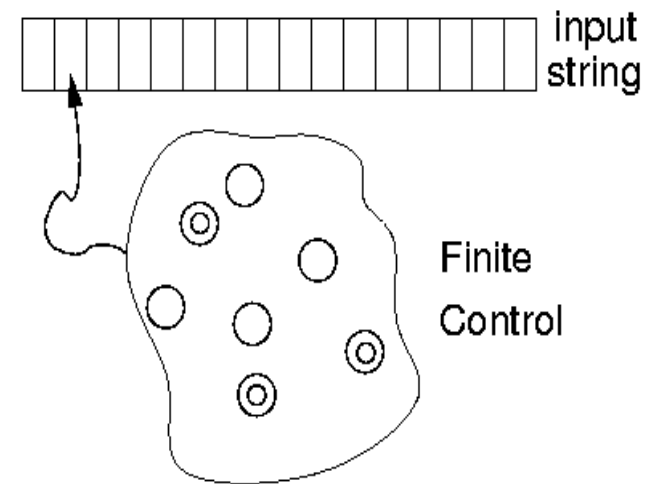
Common Extensions to r.e. Notation

- One or more repetitions of r : r^+
- A range of characters : $[a-zA-Z]$, $[0-9]$
- An optional expression: $r^?$
- Any single character: $.$
- Giving names to regular expressions, e.g.:
 - $\text{letter} = [a-zA-Z_]$
 - $\text{digit} = 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$
 - $\text{ident} = \text{letter} (\text{letter} \mid \text{digit})^*$
 - $\text{Integer_const} = \text{digit}^+$

Recognizing Tokens: Finite Automata

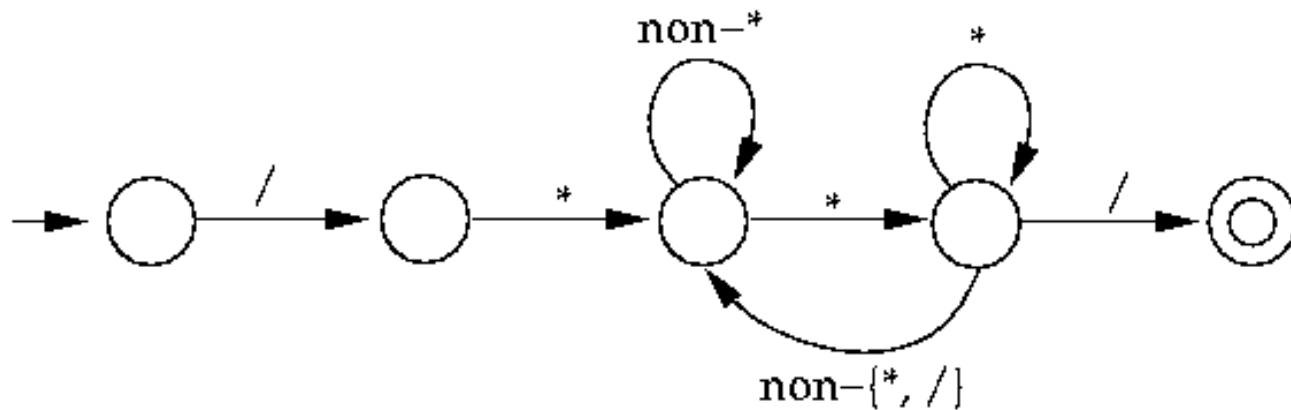
A finite automaton is a 5-tuple (Q, Σ, T, q_0, F) , where:

- Σ is a finite alphabet;
- Q is a finite set of states;
- $T: Q \times \Sigma \rightarrow Q$ is the transition function;
- $q_0 \in Q$ is the initial state; and
- $F \subseteq Q$ is a set of final states.

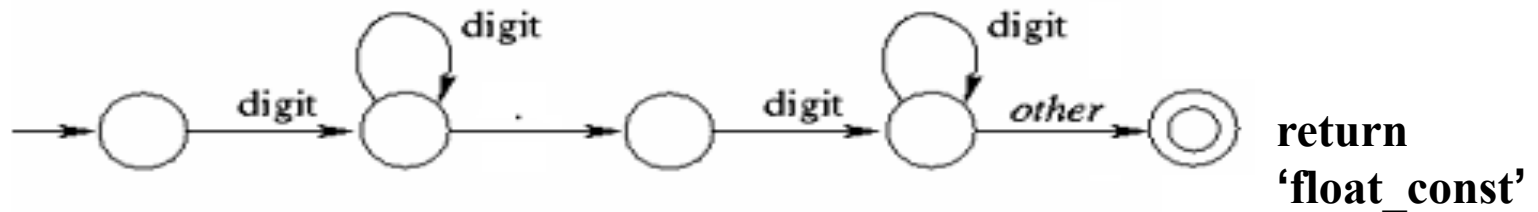
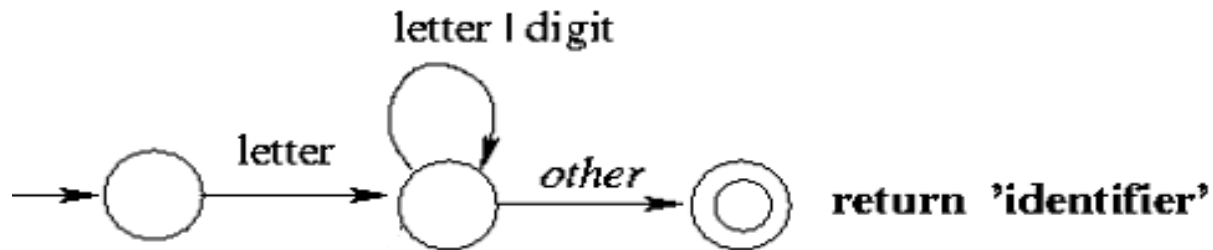
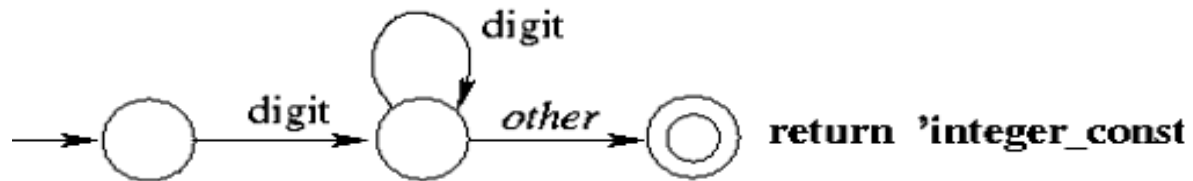


Finite Automata: An Example

A (deterministic) finite automaton (DFA) to match
C - style comments:



Identify Integer Constant, Real Constant, Identifier.



Finite Automata and Lexical Analysis

- The tokens of a language are specified using regular expressions.
- A scanner is a big DFA, essentially the “aggregate” of the automata for the individual tokens.
- Issues:
 - What does the scanner automaton look like?
 - How much should we match? (When do we stop?)
 - What do we do when a match is found?
 - Buffer management (for efficiency reasons).

How much should we match?

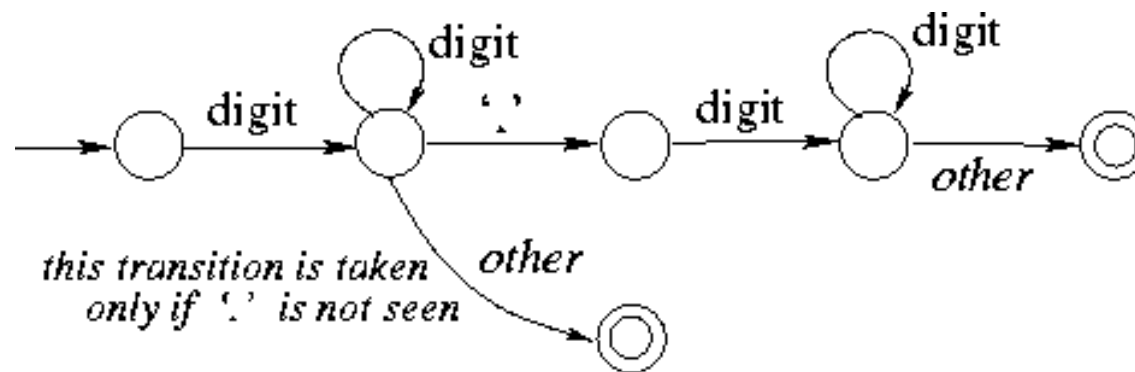
In general, find the longest match possible.

E.g., on input **123.45**, match this as

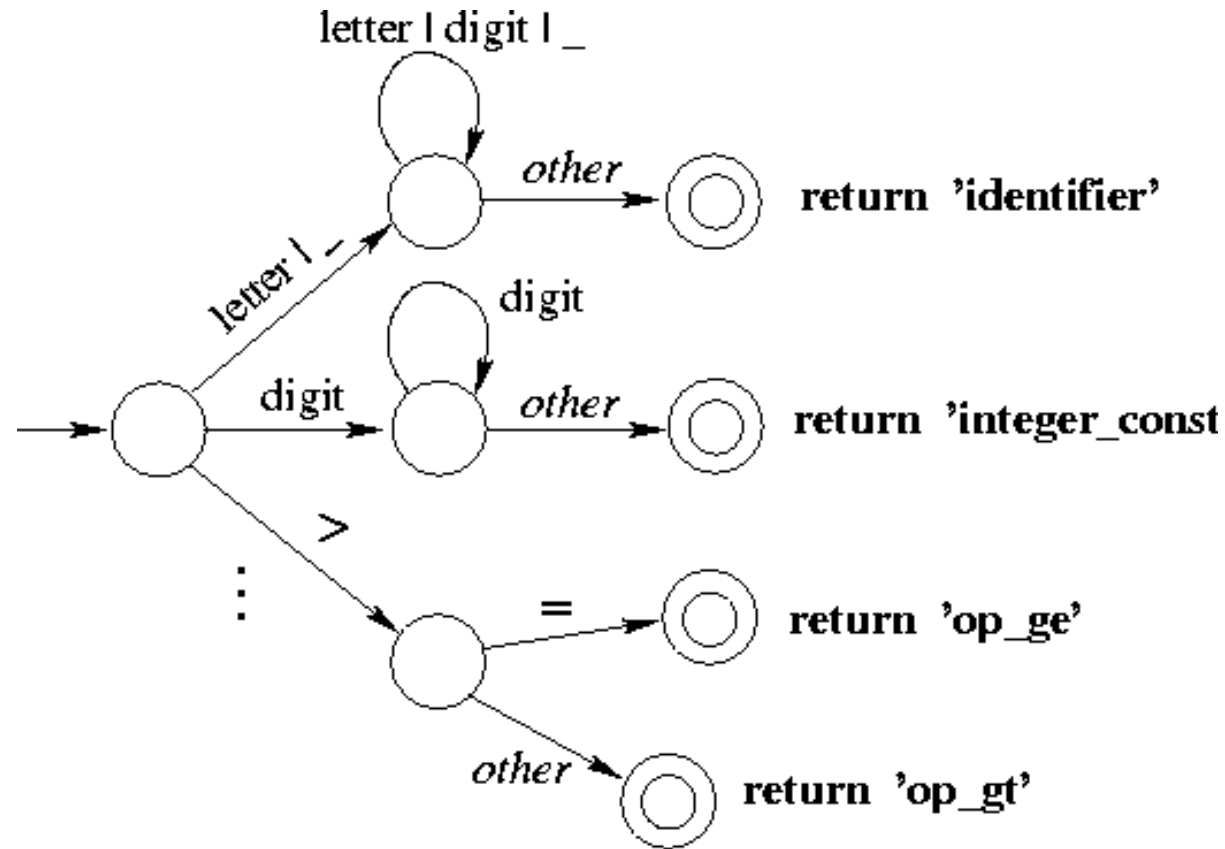
`num_const(123.45)`

rather than

`num_const(123), ".", num_const(45).`



Structure of a Scanner Automaton



Scanner Automaton

- Direct-Coded Automaton
- Table-driven automata
- Actions on finding a match.
- Incorporating Symbol table
- Handling reserved words (symbol table interface)

Implementing Lexical Analyzers

Different approaches:

- Using a scanner generator, e.g., **lex** or **flex**. This automatically generates a lexical analyzer from a high-level description of the tokens.
(easiest to implement; least efficient)
- Programming it in a language such as C, using the I/O facilities of the language.
(intermediate in ease, efficiency)

Input Buffering

- Buffer Pairs : Scanners use double-buffering to minimise the overheads associated with this.
- Buffer Pairs using sentinels : optimise to reduce number of tests.

Syntax Analysis / Parsing

Context-Free Grammars

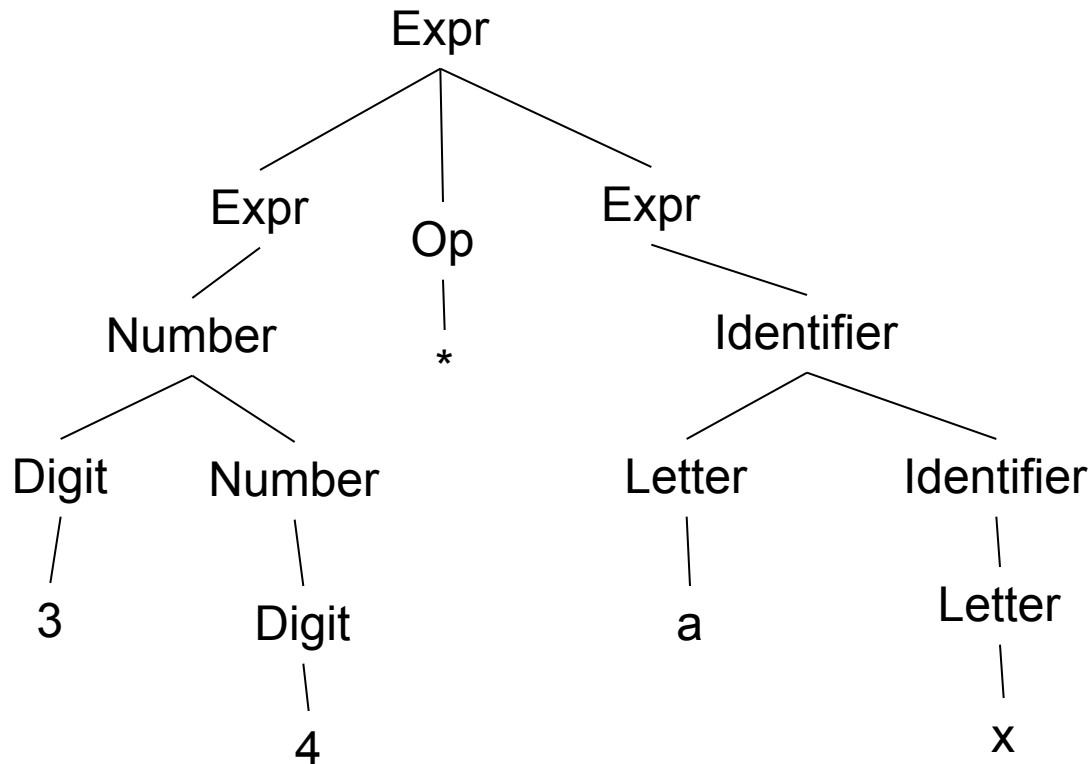
- A context-free grammar (CFG) consists of a set of **production rules**
- Each rule describes how a **non-terminal symbol** can be “replaced” or “expanded” by a string that consists of **non-terminal symbols** or by **terminal symbols**
 - Terminal symbols are really tokens
 - Rules are written with syntax like regular expressions
- Rules can then be applied recursively
- Eventually one reaches a string of only terminal symbols, or so one hopes
- This string is syntactically correct according to the grammatical rules.

What is Parsing?

- What we just saw is the process of, starting with the start symbol and, through a sequence of rule derivation obtain a string of terminal symbols
 - We could generate all correct programs
- When we say we can't parse a string, we mean that we can't find any legal way in which the string can be obtained from the start symbol through derivations
- Parser : a program that takes in a string of tokens (terminal symbols) and discovers a derivation sequence, thus validating that the input is a syntactically correct program

Derivations as Trees

- A convenient and natural way to represent a sequence of derivations is a **syntactic tree** or **parse tree**
- Example:
- $\text{Expr} \rightarrow \text{Expr Op Expr} \rightarrow \text{Number Op Expr} \rightarrow \text{Digit Number Op Expr} \rightarrow 3 \text{ Number Op Expr} \rightarrow 34 \text{ Op Expr} \rightarrow 34 * \text{Expr} \rightarrow 34 * \text{Identifier} \rightarrow 34 * \text{Letter Identifier} \rightarrow 34 * a \text{ Identifier} \rightarrow 34 * a \text{ Letter} \rightarrow 34 * ax$



Ambiguity

- We call a grammar **ambiguous** if a string of terminal symbols can be reached by two different derivation sequences
- In other terms, a string can have more than one parse tree
- It turns out that our expression grammar is ambiguous.

How do we build a Parser?

- There are two approaches for parsing:
 - **Top-Down**: Start with the start symbol and try to expand it using derivation rules until you get the input source code
 - **Bottom-Up**: Start with the input source code, consume symbols, and infer which rules could be used
- Both techniques does not work for all CFGs
 - CFGs must have some properties to be parsable with our parsing algorithms

Top-Down Parsing

- A simple recursive algorithm
 - Start with the start symbol
 - Pick one of the rules to expand and expand it
 - If the leftmost symbol is a non-terminal and matches the current token of the input source, great
 - If there is no match, then backtrack and try another rule
 - Repeat for all non-terminal symbols
 - Success if we get all terminals
 - Failure if we've tried all productions without getting all terminals.

Left-Recursion

- One problem for the Top-Down approach is **left-recursive rules**
- Example: $\text{Expr} \rightarrow \text{Expr} + \text{Number}$
 - The Parser will expand the leftmost Expr as $\text{Expr} + \text{Number}$ to get: “Expr + Number + Number”
 - And again: “Expr + Number + Number + Number”
 - And again: “Expr + Number + Number + Number + Number”
 - Ad infinitum. . .
 - Since the leftmost symbol is never a non-terminal symbol the parser will never check for a match with the source code and will be stuck in an infinite loop
- Remove left-recursion

Implementing top down parsing

- The following features are needed to implement top down parsing:
 1. Source string marker (SSM): SSM points to the first unmatched symbol in the source string.
 2. Prediction making mechanism: This mechanism systematically selects the RHS alternatives of a production during prediction making. It must ensure that any string in LG can be derived from S.
 3. Matching and backtracking mechanism: This mechanism matches every terminal symbol generated during a derivation with the source symbol pointed to by SSM. (This implements the incremental continuation check.) Backtracking is performed if the match fails. This involves resetting CSF and SSM to earlier values.

Top Down Parsing

- Two problems arise due to the possibility of backtracking.
 - First, semantic actions cannot be performed while making a prediction. The actions must be delayed until the prediction is known to be a part of a successful parse.
 - Second, precise error reporting is not possible. A mismatch merely triggers backtracking. A source string is known to be erroneous only after all predictions have failed. This makes it impossible to pinpoint the violations of PL specification.
- Grammars containing left recursion are not amenable to top down parsing. For example, consider parsing of the string $\langle \text{id} \rangle + \langle \text{id} \rangle * \langle \text{id} \rangle$ according to the grammar $E = E + T$.
- The first prediction would be $E = E + T$ which makes E the leftmost NT in CSF once again. Thus, the parser would enter an infinite loop of prediction making.

Top down parsing without backtracking (Predictive Parser)

- Elimination of backtracking in top down parsing would have several advantages—
 - Parsing would become more efficient.
 - It would be possible to perform semantic actions and precise error reporting during parsing.
 - Prediction making becomes very crucial when backtracking is eliminated.
 - The parser must use some contextual information from the source string to decide which prediction to make for the leftmost NT.

(Predictive Parser)

- The complete rewritten form of Grammar is
 - $E ::= TE''$
 - $E'' ::= +E \mid \varepsilon$
 - $T ::= VT''$
 - $T'' ::= *T \mid \varepsilon$
 - $V ::= \langle \text{id} \rangle$
- Note that grammar does not need left factoring since its RHS alternatives produce unique terminal symbols in the first position.

$E ::= T+E \mid T$
 $T ::= V*T \mid V$
 $V ::= \langle \text{id} \rangle$

Example Parsing of $\langle id \rangle + \langle id \rangle * \langle id \rangle$

- according to Grammar proceeds as shown in Table

<i>Sr. No.</i>	<i>CSF</i>	<i>Symbol</i>	<i>Prediction</i>
1.	E	$\langle id \rangle$	$E \Rightarrow T E''$
2.	$T E''$	$\langle id \rangle$	$T \Rightarrow V T''$
3.	$V T'' E''$	$\langle id \rangle$	$V \Rightarrow \langle id \rangle$
4.	$\langle id \rangle T'' E''$	$+$	$T'' \Rightarrow \epsilon$
5.	$\langle id \rangle E''$	$+$	$E'' \Rightarrow + E$
6.	$\langle id \rangle + E$	$\langle id \rangle$	$E \Rightarrow T E''$
7.	$\langle id \rangle + T E''$	$\langle id \rangle$	$T \Rightarrow V T''$
8.	$\langle id \rangle + V T'' E''$	id	$V \Rightarrow \langle id \rangle$
9.	$\langle id \rangle + \langle id \rangle T'' E''$	$*$	$T'' \Rightarrow * T$
10.	$\langle id \rangle + \langle id \rangle * T E''$	$\langle id \rangle$	$T \Rightarrow V T''$
11.	$\langle id \rangle + \langle id \rangle * V T'' E''$	$\langle id \rangle$	$V \Rightarrow \langle id \rangle$
12.	$\langle id \rangle + \langle id \rangle * \langle id \rangle T'' E''$	$-$	$T'' \Rightarrow \epsilon$
13.	$\langle id \rangle + \langle id \rangle * \langle id \rangle E''$	$-$	$E'' \Rightarrow \epsilon$
14.	$\langle id \rangle + \langle id \rangle * \langle id \rangle$	$-$	$-$

A recursive descent parser

- A recursive descend (RD) parser is a variant of top down parsing without backtracking.
- It uses a set of recursive procedures to perform parsing.
- Salient advantages of recursive descent parsing are its simplicity and generality. It can be implemented in any language supporting recursive procedures.
- To implement recursive descent parsing, a left-factored grammar is modified to make repeated occurrences of strings more explicit.
- Grammar is rewritten as
 - $E ::= T \{ + T \}^*$
 - $T ::= V \{ * V \}^*$
 - $V ::= \langle \text{id} \rangle$

Top-Down Parsing

- The parse tree is created top to bottom.
- Top-down parser
 - Recursive-Descent Parsing
 - Backtracking is needed (If a choice of a production rule does not work, we backtrack to try other alternatives.)
 - It is a general parsing technique, but not widely used.
 - Not efficient
 - Predictive Parsing
 - no backtracking
 - efficient
 - needs a special form of grammars (LL(1) grammars).
 - Recursive Predictive Parsing is a special form of Recursive Descent parsing without backtracking.
 - Non-Recursive (Table Driven) Predictive Parser is also known as LL(1) parser.

Abstract Syntax Tree

- The parser accepts *syntactically* correct programs and produces a full parse tree.
- Unfortunately, being syntactically correct is a necessary condition for the program to be correct (i.e., compilable), but it is not sufficient.
- Let's see this on a simple example
- Say we want to write a compiler for the Ada language
- The Ada language requires that procedures be written as:

```
procedure my_func
...
end my_func
```
- An incorrect program:

```
procedure my_func
...
end some_other_name
```
- Problem: There is no way to express the “both names should be the same” requirement in a CFG!
 - Both are seen as a `TOKEN_IDENT` token

Attributed Syntax Tree

- To perform such checks we need to associate attributes to nodes in the Syntax Tree and to define rules about these attributes
- You can really see this as adding tons of little pieces of code associated with grammar rules
- Example #1:
 - The Ada Example

```
ProcDecl → procedure Ident
          ProcBody
          end Ident
```
- Whenever this rule is used, run the piece of code:

```
if (Ident[1].lexeme != Ident[2].lexeme) {
    fprintf(stderr, "Syntax error: non-matched procedure names\n");
    exit(1); }
```

Attributed Syntax Tree

- **Example #2 : Type Checking**

Say we have a language in which the body of a function can declare variables

$\text{VarDecl} \rightarrow \text{Type Ident} ";"$

Each time we see this we execute the following code:

`Symbol_Table.insert(Ident.lexeme, type.lexeme);`

(updates some table that keeps track of variables and their types)

$\text{Sum} \rightarrow \text{Ident} "=" \text{Number} + \text{Number}$

Each time we see this we execute the following code:

```
if ((Number[1].type == "float") && (Number[1].type == "float")) {  
    if (Symbol_Table.lookupType(Ident.lexeme) != "float") {  
        fprintf(stderr, "Syntax error: float must be assigned to float\n");  
        exit(1);  
    }  
}
```

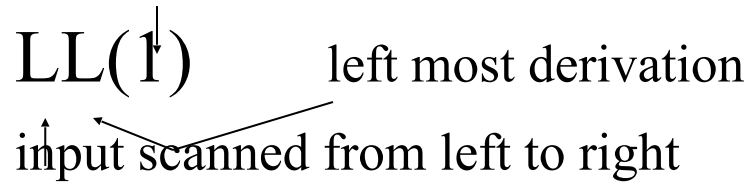
LL(1) Parsing

LL(1) Grammars

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

one input symbol used as a look-head symbol do determine parser action

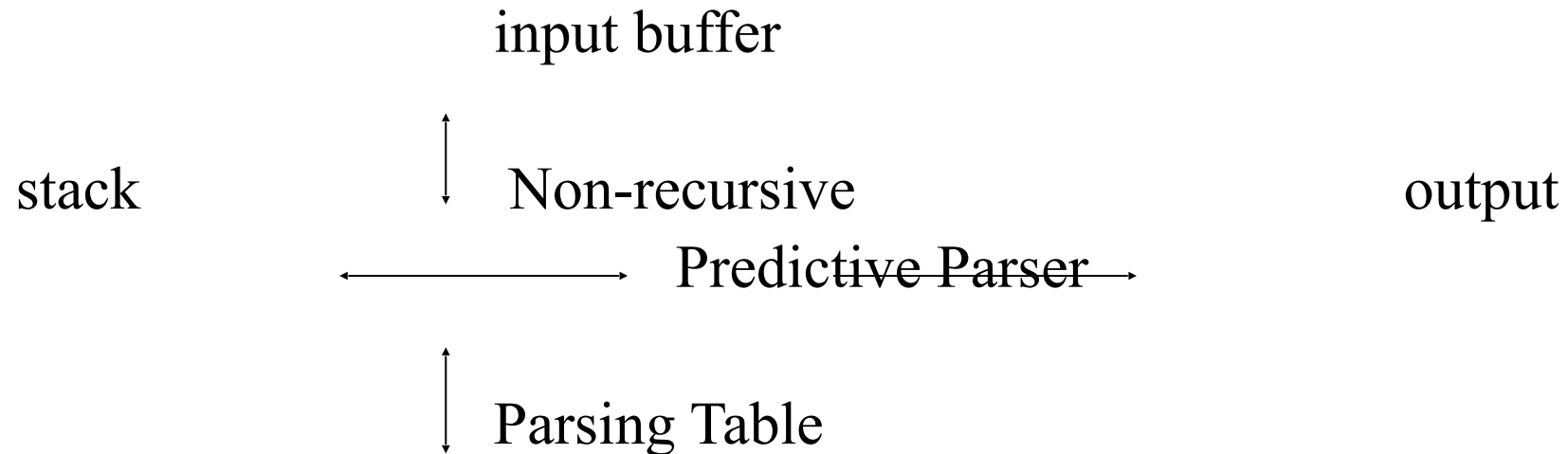
LL(\uparrow) left most derivation
input scanned from left to right



- The parsing table of a grammar may contain more than one production rule. In this case, we say that it is not a LL(1) grammar.

Non-Recursive Predictive Parsing -- LL(1) Parser

- It parses the input from **Left** to right, and constructs a **Leftmost** derivation of the sentence.
- Non-Recursive predictive parsing is a table-driven parser.
- It is a top-down parser.
- It is also known as LL(1) Parser.



LL(1) Parser

input buffer

- our string to be parsed. We will assume that its end is marked with a special symbol \$.

output

- a production rule representing a step of the derivation sequence (left-most derivation) of the string in the input buffer.

stack

- contains the grammar symbols
- at the bottom of the stack, there is a special end marker symbol \$.
- initially the stack contains only the symbol \$ and the starting symbol S.
- \$S is initial stack
- when the stack is emptied (ie. only \$ left in the stack), the parsing is completed.

parsing table

- a two-dimensional array $M[A,a]$
- each entry holds a production rule.

LL(1) Parser – Parser Actions

- The symbol at the top of the stack (say X) and the current symbol in the input string (say a) determine the parser action.
- There are four possible parser actions.
 1. If X and a are $\$$ \rightarrow parser halts (successful completion)
 1. If X and a are the same terminal symbol (different from $\$$)
 \rightarrow parser pops X from the stack, and moves the next symbol in the input buffer.
 3. If X is a non-terminal
 \rightarrow parser looks at the parsing table entry $M[X,a]$. If $M[X,a]$ holds a production rule $X \rightarrow Y_1 Y_2 \dots Y_k$, it pops X from the stack and pushes Y_k, Y_{k-1}, \dots, Y_1 into the stack. The parser also outputs the production rule $X \rightarrow Y_1 Y_2 \dots Y_k$ to represent a step of the derivation.
- none of the above \rightarrow error
 - all empty entries in the parsing table are errors.
 - If X is a terminal symbol different from a , this is also an error case.

Constructing LL(1) Parsing Tables

- Two functions are used in the construction of LL(1) parsing tables:
 - FIRST FOLLOW
- **FIRST(α)** is a set of the terminal symbols which occur as first symbols in strings derived from α where α is any string of grammar symbols.
- if α derives to ϵ , then ϵ is also in FIRST(α) .
- **FOLLOW(A)** is the set of the terminals which occur immediately after (follow) the *non-terminal* A in the strings derived from the starting symbol.
 - a terminal a is in FOLLOW(A) if $S \Rightarrow_* \alpha A a \beta$
 - \$ is in FOLLOW(A) if $S \Rightarrow_* \alpha A$

Compute FIRST for Any String X

1. If X is a terminal symbol
 1. $\rightarrow \text{FIRST}(X) = \{X\}$
2. If X is a non-terminal symbol and $X \rightarrow \epsilon$ is a production rule
 1. $\rightarrow \text{FIRST}(X) = \{\epsilon\}$.
3. If X is a non-terminal symbol and $X \rightarrow Y_1 Y_2 \dots Y_n$ is a production rule
 1. \rightarrow if a terminal **a** in $\text{FIRST}(Y_i)$ and ϵ is in all $\text{FIRST}(Y_j)$ for $j=1, \dots, i-1$ then **a** is in $\text{FIRST}(X)$.
 2. \rightarrow if ϵ is in all $\text{FIRST}(Y_j)$ for $j=1, \dots, n$ then ϵ is in $\text{FIRST}(X)$.

Compute FOLLOW (for non-terminals)

1. If S is the start symbol
 1. \rightarrow $\$$ is in $\text{FOLLOW}(S)$

1. if $A \rightarrow \alpha B \beta$ is a production rule
 1. \rightarrow Everything in $\text{FIRST}(\beta)$ is $\text{FOLLOW}(B)$ except ϵ

1. If ($A \rightarrow \alpha B$ is a production rule) or ($A \rightarrow \alpha B \beta$ is a production rule and ϵ is in $\text{FIRST}(\beta)$)
 1. \rightarrow everything in $\text{FOLLOW}(A)$ is in $\text{FOLLOW}(B)$.

We apply these rules until nothing more can be added to any follow set.

FIRST & FOLLOW Set

FIRST Set

Grammar Rule

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) \mid id$

$FIRST(F) = \{ (, id \} \text{ -- Rule 1 , 1}$

$FIRST(T') = \{ *, \epsilon \} \text{ -- Rule 1 , 2}$

$FIRST(T) = \{ (, id \} \text{ -- Rule 3}$

$FIRST(E') = \{ +, \epsilon \} \text{ -- Rule 1 , 2}$

$FIRST(E) = \{ (, id \} \text{ -- Rule 3}$

FOLLOW Set

$FOLLOW(E) = \{ \$,) \} \text{ -- Rule 1 , 2}$

$FOLLOW(E') = \{ \$,) \} \text{ -- Rule 3}$

$FOLLOW(T) = \{ +,), \$ \} \text{ -- Rule 2,3,3}$

$FOLLOW(T') = \{ +,), \$ \} \text{ -- Rule 3}$

$FOLLOW(F) = \{ +, *,), \$ \} \text{ -- Rule 3,2,3,3}$

$F \rightarrow (E)$

$E \rightarrow TE'$

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

$T \rightarrow FT'$

$T' \rightarrow *FT'$

Constructing LL(1) Parsing Table -- Algorithm

- for each production rule $A \rightarrow \alpha$ of a grammar G
 1. for each terminal a in $\text{FIRST}(A)$
 \rightarrow add $A \rightarrow \alpha$ to $M[A,a]$
 2. If ϵ in $\text{FIRST}(\alpha)$, then for each terminal b in $\text{FOLLOW}(A)$,
 \rightarrow add $A \rightarrow \alpha$ to $M[A,b]$
 3. If ϵ in $\text{FIRST}(\alpha)$ and $\$$ in $\text{FOLLOW}(A)$
add $A \rightarrow \alpha$ to $M[A,\$]$
- All other undefined entries of the parsing table are error entries.

Constructing LL(1) Parsing Table -- Example

$E \rightarrow TE'$	$\text{FIRST}(TE') = \{ (, \text{id} \}$	$\boxed{?} E \rightarrow TE' \text{ into } M[E, (] \text{ and } M[E, \text{id}]$
$E' \rightarrow +TE'$	$\text{FIRST}(+TE') = \{ + \}$	$\boxed{?} E' \rightarrow +TE' \text{ into } M[E', +]$
$E' \rightarrow \varepsilon$	$\text{FIRST}(\varepsilon) = \{ \varepsilon \}$ but since ε in $\text{FIRST}(E')$ and $\text{FOLLOW}(E') = \{ \$,) \}$	$\boxed{?}$ none $\boxed{?} E' \rightarrow \varepsilon \text{ into } M[E', \$] \text{ and } M[E',)]$
$T \rightarrow FT'$	$\text{FIRST}(FT') = \{ (, \text{id} \}$	$\boxed{?} T \rightarrow FT' \text{ into } M[T, (] \text{ and } M[T, \text{id}]$
$T' \rightarrow *FT'$	$\text{FIRST}(*FT') = \{ * \}$	$\boxed{?} T' \rightarrow *FT' \text{ into } M[T', *]$
$T' \rightarrow \varepsilon$	$\text{FIRST}(\varepsilon) = \{ \varepsilon \}$ but since ε in $\text{FIRST}(\varepsilon)$ and $\text{FOLLOW}(T') = \{ \$,), + \}$	$\boxed{?}$ none $\boxed{?} T' \rightarrow \varepsilon \text{ into } M[T', \$], M[T',)] \text{ and } M[T', +]$
$F \rightarrow (E)$	$\text{FIRST}((E)) = \{ (\}$	$\boxed{?} F \rightarrow (E) \text{ into } M[F, (]$
$F \rightarrow \text{id}$	$\text{FIRST}(\text{id}) = \{ \text{id} \}$	$\boxed{?} F \rightarrow \text{id} \text{ into } M[F, \text{id}]$

LL(1) Parser – Example

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

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

	id	+	*	()	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$F \rightarrow \text{id}$			$F \rightarrow (E)$		

LL(1) Parser – Example

<u>stack</u>	<u>input</u>	<u>output</u>
\$E	id+id\$	$E \rightarrow TE'$
\$E'T	id+id\$	$T \rightarrow FT'$
\$E'T'F	id+id\$	$F \rightarrow id$
\$E'T'id	id+id\$	
\$E'T'	+id\$	$T' \rightarrow \varepsilon$
\$E'	+id\$	$E' \rightarrow +TE'$
\$E'T+	+id\$	
\$E'T	id\$	$T \rightarrow FT'$
\$E'T'F	id\$	$F \rightarrow id$
\$E'T'id	id\$	
\$E'T'	\$	$T' \rightarrow \varepsilon$
\$E'	\$	$E' \rightarrow \varepsilon$
\$	\$	accept

LL(1) Grammer

- Which grammar is not LL(1)?
- Properties of LL(1) grammar.

Error Recovery in Predictive Parsing

- An error may occur in the predictive parsing (LL(1) parsing)
 - if the terminal symbol on the top of stack does not match with the current input symbol.
 - if the top of stack is a non-terminal A , the current input symbol is a , and the parsing table entry $M[A,a]$ is empty.
- What should the parser do in an error case?
 - The parser should be able to give an error message (as much as possible meaningful error message).
 - It should be able to recover from that error case, and it should be able to continue the parsing with the rest of the input.

Error Recovery Techniques

- **Panic-Mode Error Recovery**
 - Skipping the input symbols until a synchronising token is found.
- **Phrase-Level Error Recovery**
 - Each empty entry in the parsing table is filled with a pointer to a specific error routine to take care that error case.
- **Error-Productions**
 - If we have a good idea of the common errors that might be encountered, we can augment the grammar with productions that generate erroneous constructs.
 - When an error production is used by the parser, we can generate appropriate error diagnostics.
 - Since it is almost impossible to know all the errors that can be made by the programmers, this method is not practical.
- **Global-Correction**
 - Ideally, we would like a compiler to make as few change as possible in processing incorrect inputs.
 - We have to globally analyse the input to find the error.
 - This is an expensive method, and it is not in practice.

Example - Error Recovery

$S \rightarrow AbS \mid e \mid \varepsilon$

$A \rightarrow a \mid cAd$

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

$\text{FOLLOW}(A) = \{b, d\}$

	a	b	c	d	e	\$
S	$S \rightarrow AbS$	<i>sync</i>	$S \rightarrow AbS$	<i>sync</i>	$S \rightarrow e$	$S \rightarrow \varepsilon$
A	$A \rightarrow a$	<i>sync</i>	$A \rightarrow cAd$	<i>sync</i>	<i>sync</i>	<i>sync</i>

<u>stack</u>	<u>input</u>	<u>output</u>
\$S	aab\$	$S \rightarrow AbS$
\$SbA	aab\$	$A \rightarrow a$
\$Sba	aab\$	
\$Sb	ab\$	Error: missing b, inserted
\$S	ab\$	
\$SbA	ab\$	$A \rightarrow a$
\$Sba	ab\$	
\$Sb	b\$	
\$S	\$	
\$	\$	accept

<u>stack</u>	<u>input</u>	<u>output</u>
\$S		ceadb\$ $S \rightarrow AbS$
\$SbA	ceadb\$	$A \rightarrow cAd$
\$SbdAc	ceadb\$	
\$SbdA	eadb\$	Error: unexpected e (illegal A)
$S \rightarrow AbS$		(Remove all input tokens until first b or d, pop A)
\$Sbd	db\$	
\$Sb	b\$	
\$S	\$	$S \rightarrow \varepsilon$
$S \rightarrow \varepsilon$	\$	\$ accept

Bottom - Up Parsing

Bottom-Up Parsing

- A **bottom-up parser** creates the parse tree of the given input starting from leaves towards the root.
- A bottom-up parser tries to find the right-most derivation of the given input in the reverse order.

$S \Rightarrow \dots \Rightarrow \omega$ (the right-most derivation of ω)

\leftarrow (the bottom-up parser finds the right-most derivation in the reverse order)

- Bottom-up parsing is also known as **shift-reduce parsing** because its two main actions are shift and reduce.
 - At each shift action, the current symbol in the input string is pushed to a stack.
 - At each reduction step, the symbols at the top of the stack (this symbol sequence is the right side of a production) will be replaced by the non-terminal at the left side of that production.
 - There are also two more actions: accept and error.

Shift-Reduce Parsing

- A shift-reduce parser tries to reduce the given input string into the starting symbol.
a string \rightarrow the starting symbol
reduced to
- At each reduction step, a substring of the input matching to the right side of a production rule is replaced by the non-terminal at the left side of that production rule.
- If the substring is chosen correctly, the right most derivation of that string is created in the reverse order.

Rightmost Derivation:

$$S \xRightarrow[*]{\text{rm}} \omega$$

Shift-Reduce Parser finds:

$$\omega \xleftarrow{\text{rm}} \dots \xleftarrow{\text{rm}} S$$

Shift-Reduce Parsing -- Example

$S \rightarrow aABb$

$A \rightarrow aA \mid a$

$B \rightarrow bB \mid b$

input string: $aaabb$

$aabb$

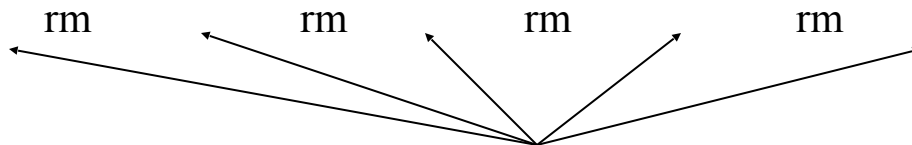
$aAbb$

↓ reduction

$aABb$

S

$S \Rightarrow aABb \Rightarrow aAbb \Rightarrow aaAbb \Rightarrow aaabb$



Right Sentential Forms

- How do we know which substring to be replaced at each reduction step?

Handle

- Informally, a **handle** of a string is a substring that matches the right side of a production rule.
 - But not every substring matches the right side of a production rule is handle
- A **handle** of a right sentential form $\gamma (\equiv \alpha\beta\omega)$ is a production rule $A \rightarrow \beta$ and a position of γ where the string β may be found and replaced by A to produce the previous right-sentential form in a rightmost derivation of γ .

$$\begin{array}{ccc} S & \Rightarrow & \alpha A \omega \Rightarrow \alpha \beta \omega \\ * & & \\ \text{rm} & & \text{rm} \end{array}$$

- If the grammar is unambiguous, then every right-sentential form of the grammar has exactly one handle.
- Here the ω is a string of terminals.

Handle Pruning

- A right-most derivation in reverse can be obtained by **handle-pruning**.

$$S = \gamma_0 \underset{\text{rm}}{\Rightarrow} \gamma_1 \underset{\text{rm}}{\Rightarrow} \gamma_2 \underset{\text{rm}}{\Rightarrow} \dots \underset{\text{rm}}{\Rightarrow} \gamma_{n-1} \underset{\text{rm}}{\Rightarrow} \gamma_n = \omega$$

↖ input string

- Start from γ_n , find a handle $A_n \rightarrow \beta_n$ in γ_n and replace β_n in by A_n to get γ_{n-1} .
- Then find a handle $A_{n-1} \rightarrow \beta_{n-1}$ in γ_{n-1} , and replace β_{n-1} in by A_{n-1} to get γ_{n-2} .
- Repeat this, until we reach S.

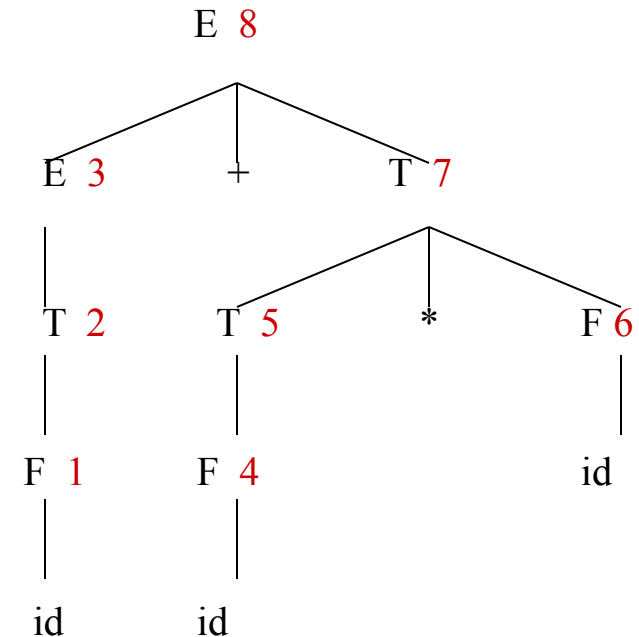
A Stack Implementation of A Shift-Reduce Parser

- There are four possible actions of a shift-parser action:
 1. **Shift** : The next input symbol is shifted onto the top of the stack.
 2. **Reduce**: Replace the handle on the top of the stack by the non-terminal.
 3. **Accept**: Successful completion of parsing.
 4. **Error**: Parser discovers a syntax error, and calls an error recovery routine.
- Initial stack just contains only the end-marker \$.
- The end of the input string is marked by the end-marker \$.

A Stack Implementation of A Shift-Reduce Parser

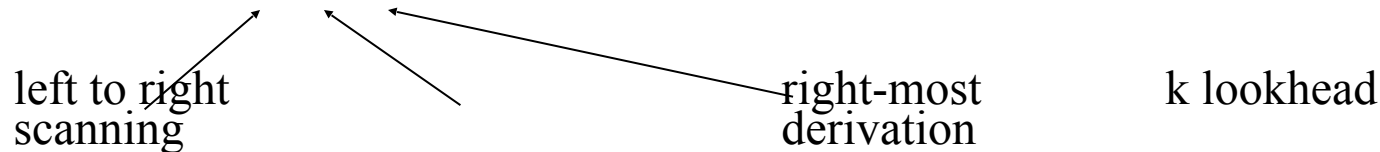
<u>Stack</u>	<u>Input</u>	<u>Action</u>
\$	id+id*id\$	shift
\$id	+id*id\$	reduce by $F \rightarrow id$
\$F	+id*id\$	reduce by $T \rightarrow F$
\$T	+id*id\$	reduce by $E \rightarrow T$
\$E	+id*id\$	shift
\$E+	id*id\$	shift
\$E+id	*id\$	reduce by $F \rightarrow id$
\$E+F	*id\$	reduce by $T \rightarrow F$
\$E+T	*id\$	shift
\$E+T*	id\$	shift
\$E+T*id	\$	reduce by $F \rightarrow id$
\$E+T*F	\$	reduce by $T \rightarrow T*F$
\$E+T	\$	reduce by $E \rightarrow E+T$
\$E	\$	accept

Parse Tree



Conflicts During Shift-Reduce Parsing

- There are context-free grammars for which shift-reduce parsers cannot be used.
- Stack contents and the next input symbol may not decide action:
 - **shift/reduce conflict**: Whether make a shift operation or a reduction.
 - **reduce/reduce conflict**: The parser cannot decide which of several reductions to make.
- If a shift-reduce parser cannot be used for a grammar, that grammar is called as non-LR(k) grammar.



- An ambiguous grammar can never be a LR grammar.

Shift-Reduce Parsers

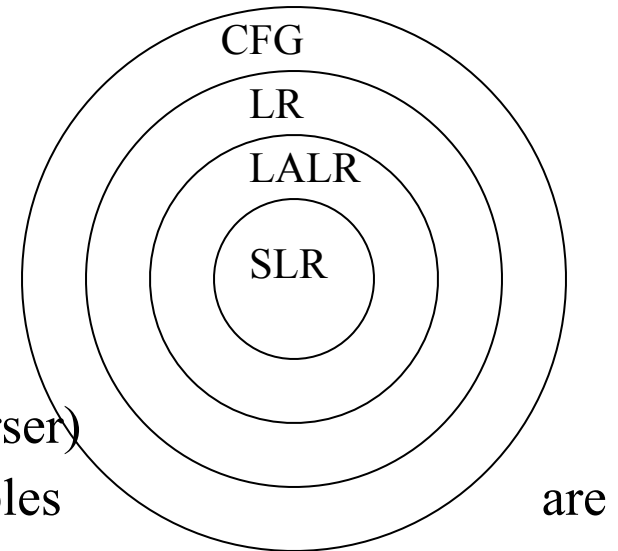
- There are two main categories of shift-reduce parsers

1. Operator-Precedence Parser

- simple, but only a small class of grammars.

2. LR-Parsers

- covers wide range of grammars.
 - SLR – simple LR parser
 - LR – most general LR parser
 - LALR – intermediate LR parser (lookahead LR parser)
- SLR, LR and LALR work same, only their parsing tables different.



Operator-Precedence Parser

- **Operator grammar**
 - small, but an important class of grammars
 - we may have an efficient operator precedence parser (a shift-reduce parser) for an operator grammar.
- In an *operator grammar*, no production rule can have:
 - ϵ at the right side
 - two adjacent non-terminals at the right side.

- Ex:

$E \rightarrow AB$

$A \rightarrow a$

|

$B \rightarrow b$

id

not operator grammar

$E \rightarrow \epsilon$

$E \rightarrow id$

$O \rightarrow + | * | /$

not operator grammar

$E \rightarrow E + E \mid$

$E * E$

$E / E \mid$

operator grammar

Precedence Relations

- In operator-precedence parsing, we define three disjoint precedence relations between certain pairs of terminals.

$a < \cdot b$ b has higher precedence than a

$a = \cdot b$ b has same precedence as a

$a \cdot > b$ b has lower precedence than a

- The determination of correct precedence relations between terminals are based on the traditional notions of associativity and precedence of operators.

Using Operator - Precedence Relations

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

The partial operator-precedence table for this grammar

	id	+	*	\$
id		.>	.>	.>
+	<.	.>	<.	.>
*	<.	.>	.>	.>
\$	<.	<.	<.	

- Then the input string $id+id*id$ with the precedence relations inserted will be:

$\$ < . id . > + < . id . > * < . id . > \$$

To Find The Handles

1. Scan the string from left end until the first \rightarrow is encountered.
2. Then scan backwards (to the left) over any $=$ until a $<$ is encountered.
3. The handle contains everything to left of the first \rightarrow and to the right of the $<$ is encountered.

\$ < . id .> + < . id .> * < . id .> \$

\$ < . + < . id .> * < . id .> \$

\$ < . + < . * < . id .> \$

\$ < . + < . * .> \$

\$ < . + .> \$

\$ \$

$E \rightarrow id$

$E \rightarrow id$

$E \rightarrow id$

$E \rightarrow E * E$

$E \rightarrow E + E$

\$ E \$

\$ id + id * id \$

\$ E + id * id \$

\$ E + E * id \$

\$ E + E * E \$

\$ E + E \$

Operator - Precedence Parsing Algorithm

- The input string is $w\$$, the initial stack is $\$$ and a table holds precedence relations between certain terminals

Algorithm:

set p to point to the first symbol of $w\$$;

repeat forever

if ($\$$ is on top of the stack **and** p points to $\$$) **then return**

else {

let a be the topmost terminal symbol on the stack and let b be the symbol pointed to by p ;

if ($a < \cdot b$ or $a = \cdot b$) **then**

{

/ SHIFT */*

push b onto the stack;

advance p to the next input symbol;

}

else if ($a > \cdot b$) **then**

/ REDUCE */*

repeat pop stack

until (the top of stack terminal is related by $< \cdot$ to the terminal most recently popped);

else error();

}

How to Create Operator - Precedence Relations

- We use associativity and precedence relations among operators.
1. If operator O_1 has higher precedence than operator O_2 ,
 $\rightarrow O_1 \cdot > O_2$ and $O_2 < \cdot O_1$

 1. If operator O_1 and operator O_2 have equal precedence,
they are left-associative $\rightarrow O_1 \cdot > O_2$ and $O_2 < \cdot O_1$
they are right-associative $\rightarrow O_1 < \cdot O_2$ and $O_2 \cdot > O_1$

 1. For all operators O ,
 $O < \cdot \text{id}$, $\text{id} \cdot > O$, $O < \cdot ($, $(< \cdot O$, $O \cdot >)$, $) \cdot > O$, $O \cdot > \$$, and $\$ < \cdot O$

 1. Also, let

$(= \cdot) \quad \$ < \cdot (\quad \text{id} \cdot >) \quad) \cdot > \$$
 $(< \cdot (\quad \$ < \cdot \text{id} \quad \text{id} \cdot > \$ \quad) \cdot >)$
 $(< \cdot \text{id}$

Operator-Precedence Relations

[illegible]

Disadvantages of Operator Precedence Parsing

- **Disadvantages:**
 - Small class of grammars.
 - Decide which language is recognized by the grammar.
- **Advantages:**
 - simple
 - powerful enough for expressions in programming languages

Error Recovery in Operator-Precedence Parsing

Error Cases:

1. No relation holds between the terminal on the top of stack and the next input symbol.
2. A handle is found (reduction step), but there is no production with this handle as a right side

Error Recovery:

1. Each empty entry is filled with a pointer to an error routine.
2. Decides the popped handle “looks like” which right hand side. And tries to recover from that situation.

Precedence Functions

- Compilers using operator precedence parsers do not need to store the table of precedence relations.
- The table can be encoded by two precedence functions f and g that map terminal symbols to integers.
- For symbols a and b .

$$f(a) < g(b) \quad \text{whenever } a < \cdot b$$

$$f(a) = g(b) \quad \text{whenever } a = \cdot b$$

$$f(a) > g(b) \quad \text{whenever } a \cdot > b$$

Thus the precedence relation between a and b can be determined by a numerical comparison between $f(a)$ and $g(b)$.

Algorithm : Constructing precedence function

Input An operator precedence matrix. $a < \cdot b$ $a = \cdot b$ $a \cdot > b$ fa gb

Output Precedence functions representing the input matrix, or an indication acne exist.

Method

1. Create symbols f and g for each a that is a terminal or $\$$.
2. Partition the created symbols into as many groups as possible, in such a way that if $a = \cdot b$, then fa and gb are in the same group. Note that we may have to put symbols in the same group even if they are not related by $= \cdot$. For example, if $a = \cdot b$ and $c = \cdot b$, then fa and fc must be in the same group, since they are both in the same group as gb . If in addition, $c = \cdot d$, then fa and gd , are in the same group even though $a = \cdot d$ may not hold.
3. Create a directed graph whose nodes are the groups found in (2). For any a and b , if $a < \cdot b$, place an edge from the group of gb , to the group Of fa . If $a \cdot > b$, place an edge from the group of fa to that of gb . Note that an edge or path from fa , to gb means that $f(a)$ must exceed $g(b)$; a path from gb to fa , means that $g(b)$ must exceed $f(a)$.
4. If the graph constructed in (3) has a cycle, then no precedence functions exist. If there are no cycles, let $f(a)$ be the length of the longest path beginning at the group of fa , let $g(a)$ be the length of the longest path from the group of ga .

Example : Graph representing precedence function

Consider the matrix of Fig.1 There are no = relationships, so each symbol is in a group by itself.

Figure 2 shows the graph constructed using Algorithm.

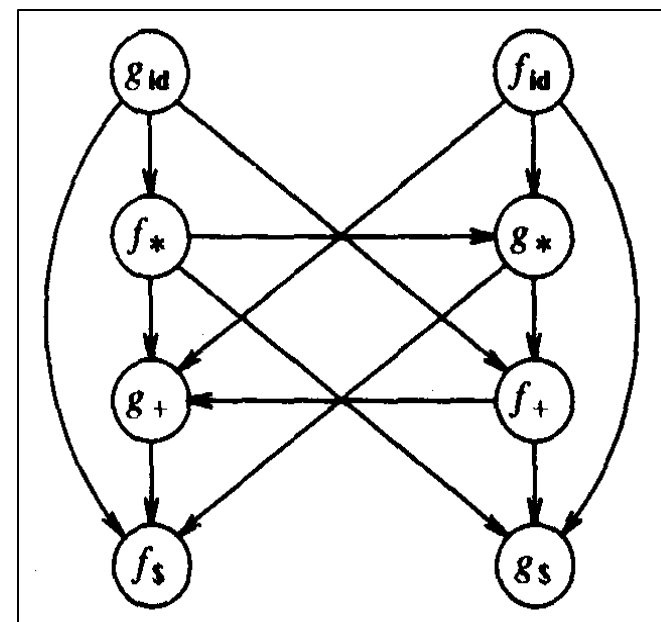
There are no cycles, so precedence functions exist.

As $f\$$ and $g\$$ have no out edges.

- $f(\$) = g(\$) = 0$. The longest path from $g+$ has length 1 so $g(+)=1$.
- There is a path from $g(id)$ to f^* to g^* to $f+$ to $g+$ to $f\$$ so $g(id)=5$.
- The resulting precedence functions are:

	+	*	id	\$
<i>f</i>	2	4	4	0
<i>g</i>	1	3	5	0

	id	+	*	\$
id		.>	.>	.>
+	<.	.>	<.	.>
*	<.	.>	.>	.>
\$	<.	<.	<.	



Parse using Precedence Function

\$ < . id . > + < . id . > * < . id . > \$

0 5

$E \rightarrow id$

\$ id + id * id \$

\$ < . + < . id . > * < . id . > \$

2 5

$E \rightarrow id$

\$ E + id * id \$

\$ < . + < . * < . id . > \$

4 5

$E \rightarrow id$

\$ E + E * id \$

\$ < . + < . * . > \$

0 1(2) 3(4) 0

$E \rightarrow E * E$

\$ E + E * . E \$

\$ < . + . > \$

0 1(2) 0

$E \rightarrow E + E$

\$ E + E \$

\$ \$

\$ E \$

	+	*	id	\$
<i>f</i>	2	4	4	0
<i>g</i>	1	3	5	0

	+	-	*	/	^	id	()	\$
+	.>	.>	<.	<.	<.	<.	<.	.>	.>
-	.>	.>	<.	<.	<.	<.	<.	.>	.>
*	.>	.>	.>	.>	<.	<.	<.	.>	.>
/	.>	.>	.>	.>	<.	<.	<.	.>	.>
^	.>	.>	.>	.>	<.	<.	<.	.>	.>
id	.>	.>	.>	.>	.>			.>	.>
(<.	<.	<.	<.	<.	<.	<.	.>	
)	.>	.>	.>	.>	.>			.>	.>
\$	<.	<.	<.	<.	<.	<.	<.		

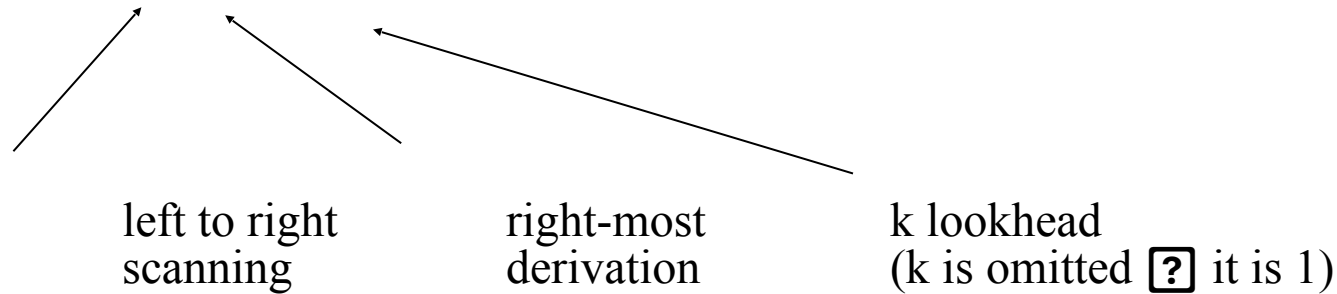
Precedence table and its equivalent function

	+	-	*	/	↑	()	id	\$
<i>f</i>	2	2	4	4	4	0	6	6	0
<i>g</i>	1	1	3	3	5	5	0	5	0

LR Parsers

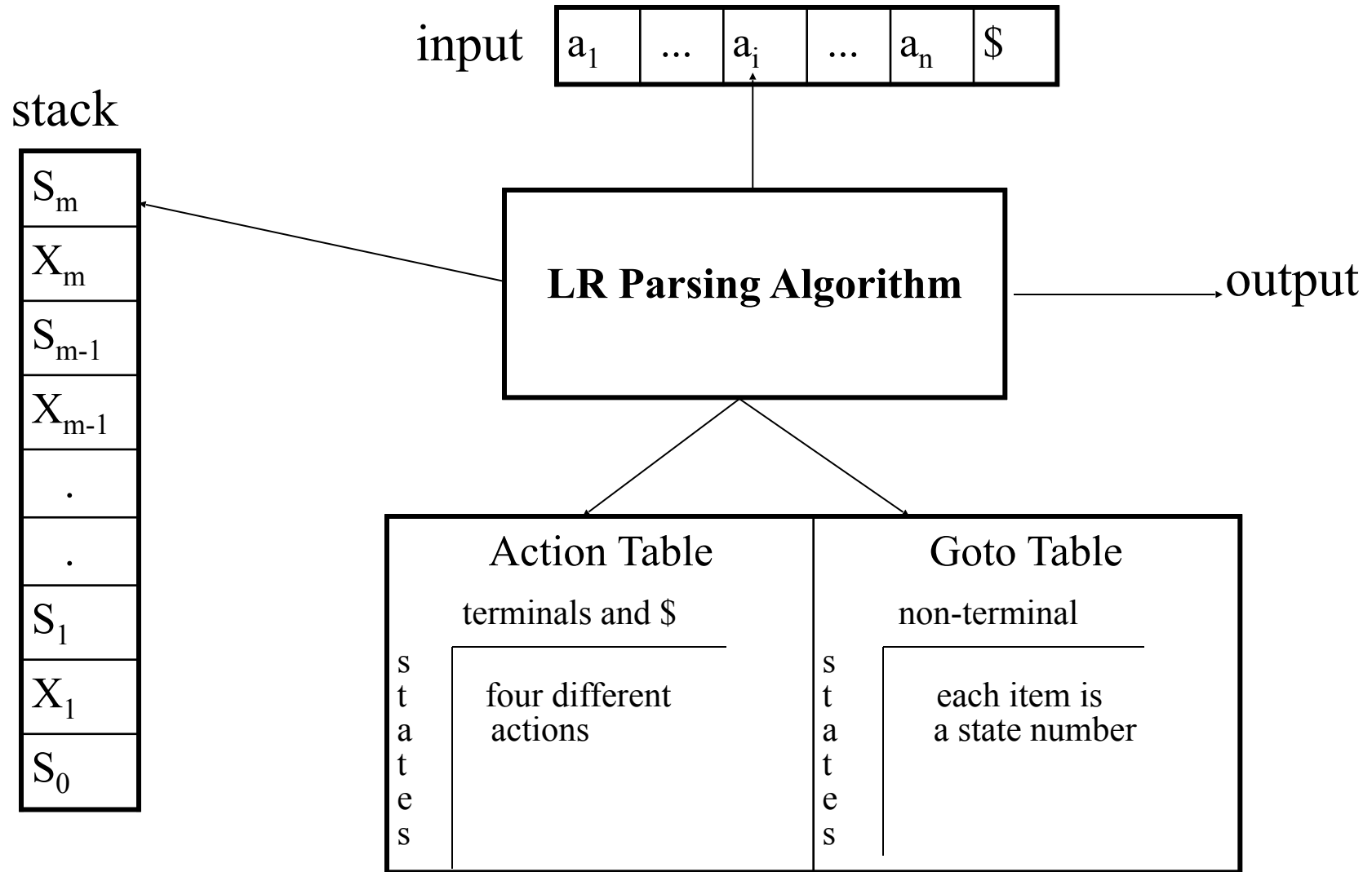
- The most powerful shift-reduce parsing (yet efficient) is:

LR(k) parsing.



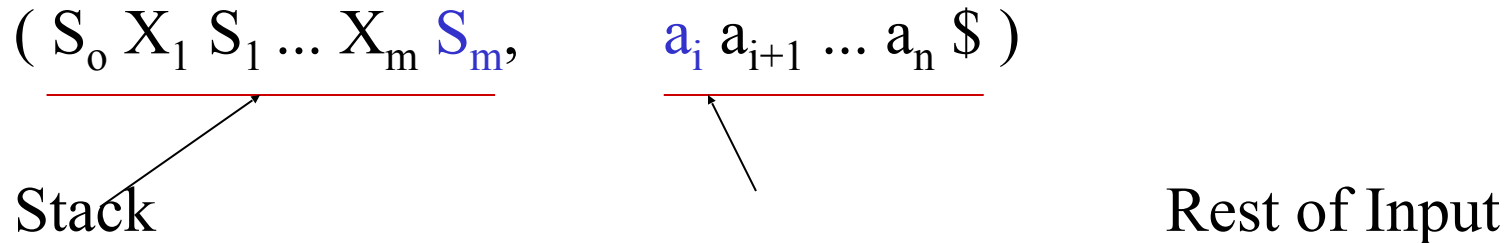
- LR parsing is attractive because:
 - LR parsing is most general non-backtracking shift-reduce parsing, yet it is still efficient.
 - The class of grammars that can be parsed using LR methods is a proper superset of the class of grammars that can be parsed with predictive parsers.
LL(1)-Grammars \subset LR(1)-Grammars
 - An LR-parser can detect a syntactic error as soon as it is possible to do so a left-to-right scan of the input.

LR Parsing Algorithm



A Configuration of LR Parsing Algorithm

- A configuration of a LR parsing is:



- S_m and a_i decides the parser action by consulting the parsing action table. (*Initial Stack* contains just S_0)
- A configuration of a LR parsing represents the right sentential form:

$$X_1 \dots X_m a_i a_{i+1} \dots a_n \$$$

Actions of A LR-Parser

1. **shift s** -- shifts the next input symbol and the state **s** onto the stack
 $(S_0 X_1 S_1 \dots X_m S_m, a_i a_{i+1} \dots a_n \$) \xrightarrow{?} (S_0 X_1 S_1 \dots X_m S_m \textcolor{red}{a_i} \textcolor{red}{s}, a_{i+1} \dots a_n \$)$
- **reduce $A \rightarrow \beta$**
 - pop $2|\beta|$ items from the stack;
 - 1. then push **A** and **s** where $\textbf{s} = \textbf{goto}[\textbf{s}_{m-r}, \textbf{A}]$
 $(S_0 X_1 S_1 \dots X_m S_m, a_i a_{i+1} \dots a_n \$) \xrightarrow{?} (S_0 X_1 S_1 \dots X_{m-r} \textcolor{red}{S_{m-r}} \textcolor{red}{A} \textcolor{red}{s}, a_i \dots a_n \$)$
 - Output is the reducing production reduce $A \rightarrow \beta$
- **Accept** – Parsing successfully completed
- **Error** -- Parser detected an error (an empty entry in the action table)

(SLR) Parsing Tables for Expression Grammar

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

Action Table

Goto Table

state	id	+	*	()	\$		E	T	F
0	s5			s4				1	2	3
1		s6				acc				
2		r2	s7		r2	r2				
3		r4	r4		r4	r4				
4	s5			s4				8	2	3
5		r6	r6		r6	r6				
6	s5			s4					9	3
7	s5			s4						10
8		s6			s11					
9		r1	s7		r1	r1				
10		r3	r3		r3	r3				
11		r5	r5		r5	r5				

Actions of A (S)LR-Parser -- Example

<u>stack</u>	<u>input</u>	<u>action</u>	<u>output</u>
0	id*id+id\$	shift 5	
0id5	*id+id\$	reduce by $F \rightarrow id$	$F \rightarrow id$
0F3	*id+id\$	reduce by $T \rightarrow F$	$T \rightarrow F$
0T2	*id+id\$	shift 7	
0T2*7	id+id\$	shift 5	
0T2*7id5	+id\$	reduce by $F \rightarrow id$	$F \rightarrow id$
0T2*7F10	+id\$	reduce by $T \rightarrow T * F$	$T \rightarrow T * F$
0T2	+id\$	reduce by $E \rightarrow T$	$E \rightarrow T$
0E1	+id\$	shift 6	
0E1+6	id\$	shift 5	
0E1+6id5	\$	reduce by $F \rightarrow id$	$F \rightarrow id$
0E1+6F3	\$	reduce by $T \rightarrow F$	$T \rightarrow F$
0E1+6T9	\$	reduce by $E \rightarrow E + T$	$E \rightarrow E + T$
0E1	\$	accept	

Panic Mode Error Recovery in LR Parsing

- Scan down the stack until a state **s** with a goto on a particular nonterminal **A** is found. (Get rid of everything from the stack before this state **s**).
- Discard zero or more input symbols until a symbol **a** is found that can legitimately follow **A**.
 - The symbol **a** is simply in FOLLOW(**A**), but this may not work for all situations.
- The parser stacks the nonterminal **A** and the state **goto[s,A]**, and it resumes the normal parsing.
- This nonterminal **A** is normally is a basic programming block (there can be more than one choice for **A**).
 - stmt, expr, block, ...

Phrase-Level Error Recovery in LR Parsing

- Each empty entry in the action table is marked with a specific error routine.
- An error routine reflects the error that the user most likely will make in that case.
- An error routine inserts the symbols into the stack or the input (or it deletes the symbols from the stack and the input, or it can do both insertion and deletion).
 - missing operand
 - unbalanced right parenthesis