Compiler Design

CS VI SEM

Prepared by
Arun Kumar Dewangan
(Lecturer Computer Sc.)

Unit - 2

Syntax Analysis & Parsing Techniques

Context Free Grammars

Assignment

Derivation

Assignment

Parse Tree

Assignment

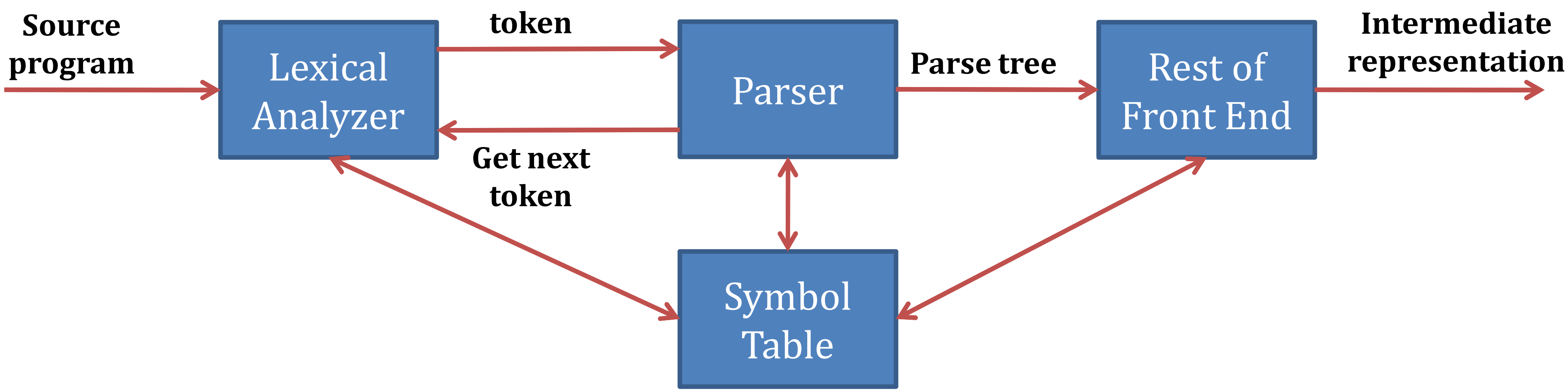
Ambiguity

Assignment

Parsers

- The parser obtains a string of tokens from lexical analyzer.
- Verifies that string of token names can be generated by the grammar for the source language.
- Parser report any syntax error in intelligible fashion and recover the commonly occurring errors to continue processing the remainder of the program.
- Parser construct the parse tree.
- It passes parse tree to the rest of the compiler for further processing.
- A parser for grammar G takes string as input w.
- And produces as output either a parse tree for w (if w is a sentence of G).
- Or an error message indicating that w is not a sentence of G.
- Two type of parser:
  1. Top down parser
  2. Bottom up parser

- Bottom up parsers build parse tree from bottom (leave) to top (root).
- Top down parser starts with root and work down to the leaves.
- The bottom up parsing method we discuss is called "shift reduce" parsing since it consist of shifting input symbols into a stack until right side of production appears on top of stack.
- The right side may then be replaced by (reduced to) symbol on left side of production and process repeated.
- In either case the input to parser is scanned from left to right, one symbol at a time.



Representation of Parse Tree

Assignment

Shift Reduce Parsing

- This uses bottom up style of parsing.
- Since it attempts to construct a parse tree for an input string beginning at the leaves (bottom) and working up towards the root (top).
- The process is reducing a string w to the start symbol of a grammar.
- At each step, string matching the right side of production is replaced by symbol on the left.

Example: Consider the grammar S → aAcBe, A → Ab | b, B → d
string is ***abbcde**.*
we want to reduce this string to S.

- Scan ***abbcde*** looking for substring that match the right side of some production.
- The substrings b and d qualify (as A → b and B → d).
- Let us choose leftmost b of string, replace it by A (from A → b).
- The string obtained is a**A**bcde.
- We can find that Ab, b and d matches the right side of some production, (A → Ab, A → b, B → d).
- Suppose we choose to replace the substring Ab by A (using A → Ab).
- The string obtained is a**A**cde.
- Replace d by B (using B → d). The string is a**A**c**B**e.
- Now we replace the entire string by S (as S → aAcBe).
- Each replacement of right side of production by left side symbol is called **reduction**.
- The process of bottom up parsing may be viewed as finding and reducing handles.

Handles

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

$$S \overset{*}{\underset{rm}{\Longrightarrow}} \alpha A w \underset{rm}{\Longrightarrow} \alpha \beta w$$

then $A \rightarrow \beta$ in the position following α is a handle of $\alpha\beta w$.

the string w to the right of handle contains only terminal symbols.

- In previous example **abbcde** is a right sentential form whose handle is $A \rightarrow b$ at position 2.
- Likewise **aAbcde** is a right sentential form whose handle is $A \rightarrow Ab$ at position 2.
- We can say that "substring β is a handle of $\alpha\beta w$".
- If grammar is unambiguous then every right sentential form has only one handle.

Example

- Consider following grammar
  1) E → E + E
  2) E → E * E
  3) E → (E)
  4) E → id
- And consider rightmost derivation

$$E \underset{rm}{\Longrightarrow} \underline{E + E}$$

$$\underset{rm}{\Longrightarrow} E + \underline{E * E}$$

$$\underset{rm}{\Longrightarrow} E + E * \underline{id}_3$$

$$\underset{rm}{\Longrightarrow} E + \underline{id}_2 * id_3$$

$$\underset{rm}{\Longrightarrow} \underline{id}_1 + id_2 * id_3$$

Description of Example

- Subscripted id's for convenience.
- Underlined denotes handle of each right sentential form.
- **$\mathbf{id_1}$** is a handle of right sentential form **$\mathbf{id_1 + id_2 * id_3}$** since **id** is right side of production E → id.
- Replacing **$\mathbf{id_1}$** by E produces previous right sentential form E **$\mathbf{+ id_2 * id_3}$**.
- String appearing to the right of handle contains only terminal symbol.

Handle Pruning

- Rightmost derivation in reverse, called a *canonical reduction sequence*, is obtained by "handle pruning".
- We start with string of terminals w which is to be parsed.
- If w is a sentence of grammar then w = γ_n, where γ_n is the nth right sentential form of some rightmost derivation.

$$S = \gamma_0 \underset{rm}{\Longrightarrow} \gamma_1 \underset{rm}{\Longrightarrow} \gamma_2 \underset{rm}{\Longrightarrow} \cdots \underset{rm}{\Longrightarrow} \gamma_{n-1} \underset{rm}{\Longrightarrow} \gamma_n = w$$

- To reconstruct this derivation in reverse order, locate handle β_n in $\gamma_{n.}$
- Replace β_n by left side of some production $A_n \rightarrow \beta_n$ to obtain the (n-1)st right sentential form Υ_{n-1}.
- Repeat this process, locate handle β_{n-1} in γ_{n-1} and reduce this handle to obtain right sentential form γ_{n-2}.
- In this way If we produce a right sentential form having only start symbol S, then we halt and this is successful completion of parsing.

Example: Consider input string **id$_1$ + id$_2$ * id$_3$** for previous grammar.

Right sentential form	Handle	Reduction production
id$_1$ + id$_2$ * id$_3$	**id$_1$**	E → id
E + **id$_2$ * id$_3$**	**id$_2$**	E → id
E + E * **id$_3$**	**id$_3$**	E → id
E + E * E	E * E	E → E * E
E + E	E + E	E → E + E
E		

Stack Implementation of Shift Reduce Parsing

- Two problem must be solved to automate parsing by handle pruning:
  1. How to locate a handle in right sentential form.
  2. What production to choose if there is more than one production with same right side.
- A convenient way to implement shift reduce parser is to use a stack & input buffer.
- Use $ to mark bottom of stack and right end of input.

Stack	**Input**
$	w $

- The parser operates by shifting zero or more input symbols onto stack until handle β is on top of stack.
- Parser then reduces β to left side of appropriate production.
- Parser repeats the cycle until it has detected an error or stack contains start symbol and input is empty.

Stack	**Input**
$ S	$

Example: Let input string is **id_1 + id_2 * id_3** for previous grammar.
Shift reduce parser might make parsing in following sequence

	Stack	Input	Action
1	$	**id_1 + id_2 * id_3 $**	Shift
2	$ **id_1**	**+ id_2 * id_3 $**	Reduce by E → id
3	$ E	**+ id_2 * id_3 $**	Shift
4	$ E +	**id_2 * id_3 $**	Shift
5	$ E + **id_2**	*** id_3 $**	Reduce by E → id
6	$ E + E	*** id_3 $**	Shift
7	$ E + E *	**id_3 $**	Shift
8	$ E + E * **id_3**	**$**	Reduce by E → id
9	$ E + E * E	$	Reduce by E → E * E
10	$ E + E	$	Reduce by E → E + E
11	$ E	$	Accept

- Four possible actions a shift reduce parser can make:
  1. Shift
  2. Reduce
  3. Accept
  4. Error
- In ***shift*** action, next input symbol is shifted to top of stack.
- In ***reduce*** action, parser knows the right end of handle is at top of stack. Locate left end of handle within stack and take decision of nonterminal to replace the handle.
- In ***accept*** action, parser announces successful completion of parsing.
- In ***error*** action, parser discovers that a syntax error is found and calls error recovery routine.

Constructing a Parse Tree

Assignment

Operator Grammar

- The grammar in which no production has two adjacent nonterminals at right side is called *operator grammar*.

Example: Consider following grammar for expressions

E → E A E | (E) | - E | id

A → + | - | * | / | ↑

is not an operator grammar, since right side E A E has two consecutive nonterminals.

If we substitute for A each its alternate, then we get following operator grammar

E → E + E | E - E | E * E | E / E | E ↑ E | (E) | - E | id

Operator Precedence Parsing

- The operator precedence technique was first described as a manipulation on tokens without any reference to underlying grammar.
- In this parsing, we use 3 disjoint precedence relations : $<\bullet$ $\doteq$ $\bullet>$ between certain pair of terminals.
- These relations guide selection of handles.
- If a $<\bullet$ b, means a "*yields precedence to*" b.
- If a $\doteq$ b, means a "*has same precedence as*" b.
- if a $\bullet>$ b, means a "*takes precedence over*" *b.*

Using Operator Precedence Relation

- Method is based on traditional notions of associativity and precedence of operators.
- Example: if * has higher precedence than +, we make + < • * and * • > +
- The intention of precedence relation is to delimit handle of right sentential form.
- With **< •** Marking left end, $\doteq$ appearing in interior of handle (if any), and **• >** marking the right end.
- Let we have right sentential form of an operator grammar.
- No adjacent nonterminals appear on right side of productions i.e. no right sentential form have two adjacent nonterminals.

- We write the right sentential form as

  $\boldsymbol{\beta_0}$ $\mathbf{a_1}$ $\boldsymbol{\beta_1}$ **.....** $\mathbf{a_n}$ $\boldsymbol{\beta_n}$ where

  Each $\boldsymbol{\beta_i}$ is either ε or a single nonterminal

  Each $\mathbf{a_i}$ is a single terminal
- Let between a_i and a_{i+1} exactly one relation holds.
- **$** marks each end of string and $ **<** • b and b • **>** $ for all terminals b.
- If
  - we remove nonterminals from string and
  - place correct relation between each pair of terminals and between endmost terminals and **$**'s marking ends of string.
- Then string with precedence relation inserted is (for previous right sentential form):

  $ <• id •> + <• id •> * <• id •> $

Example: Let input string is **$id_1 + id_2 * id_3$**. Operator precedence relation is given as

	id	**+**	*****	**$**
id		• >	• >	• >
+	< •	• >	< •	• >
*	< •	• >	• >	• >
$	< •	< •	< •	

Handle can be found in this way:

1. Scan string from left end until leftmost • > encountered.
2. Then scan backwards (to the left) over any $\doteq$ until < • is countered (in this example scan backwards to $).
3. The handle contains everything to the left of first • > and to the right of < • encountered in step 2 including any surrounding nonterminals.

Operator Precedence Relations from Associativity and Precedence

Assignment

Handling Unary Operators

Assignment

Operator Precedence Grammars

- It shows how to compute its precedence relation
- Explain the details of shift reduce parsing using precedence relations.
- Let G be an ε free operator grammar (no right side is ε and no right side has a pair of adjacent nonterminals).
- For each two terminal symbols a and b :
  1) a $\doteq$ b if there is a right side of a production of form **α a β b γ**, where β is either ε or a single nonterminal.

     that is a $\doteq$ b if a appears immediately to the left of b in a right side or if they appear separated by one nonterminal.

2) a <• b if for some nonterminal A there is a right side of the form $\alpha a A \beta$, and $A \xRightarrow{+} \gamma b \delta$ where γ is either ε or a single nonterminal.

that is a <• b if a nonterminal A appears immediately to the right of a and derives a string in which b is the first terminal symbol.

3) a •> b if for some nonterminal A there is a right side of the form $\alpha A b \beta$ and $A \xRightarrow{+} \gamma a \delta$ where δ is either ε or a single nonterminal.

that is a •> b if a nonterminal appearing immediately to the left of b derives a string whose last terminal is a.

Definition:

- An operator precedence grammar is an ε free operator grammar in which precedence relations construction are disjoint.
- That is, for any pair of terminals a and b, never more than one of relations a <• b, a $\doteq$ b, and a •> b is true.

Example: Let operator grammar is

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

in which there are only operators + and *.

- This is not an operator precedence grammar as two precedence relations hold between certain pair of terminals.
- By rule 3) (defining • **>** relation):
  - Let right side $\boldsymbol{\alpha\, A\, b\, \beta}$ be $\boldsymbol{E + E}$.
  - i.e. $\alpha = \varepsilon$, $A = E$, $b = +$, and $\beta = E$.
  - Derivation $\mathbf{A \overset{+}{\Longrightarrow} \gamma a \delta}$ could be $\boldsymbol{E \Rightarrow E + E}$,

    where $\gamma = E$, $a = +$, and $\delta = E$.
  - So relation is a • **>** b implies + • **>** +

- By rule 2) (defining **< •** relation):
  - Let right side $\boldsymbol{\alpha\ a\ A\ \beta}$ be $\boldsymbol{E + E}$.
  - i.e. a = +, A = E.
  - Derivation $\mathbf{E} \xRightarrow{+} \mathbf{E + E}$ shows that E can derive strings whose first terminal is +.
  - So relation is a **< •** b implies + **< •** +
- Both + **< •** + and + **• >** +, is not an operator precedence grammar.

Top Down Parsing

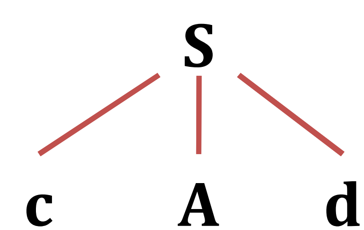
- We will discuss about:
  1. General form of top down parsing that may involve backtracking (making repeated scans of input).
  2. Recursive descent parsing, which eliminates the need for backtracking over input.
- It can be viewed as an attempt to find leftmost derivation for input string.
- It can be viewed as attempting to construct a parse tree for the input starting from root and creating node of parse tree.

Example: Consider grammar

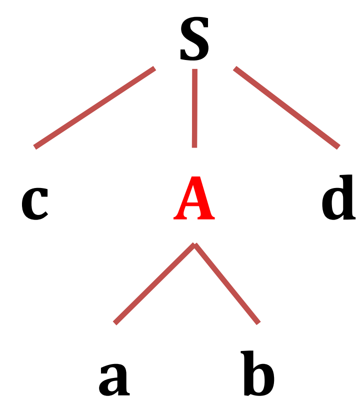
$S \rightarrow c\,A\,d$ $\quad\quad A \rightarrow a\,b \mid a$, and input string is $w = c\,a\,d$.

to construct parse tree for this sentence:

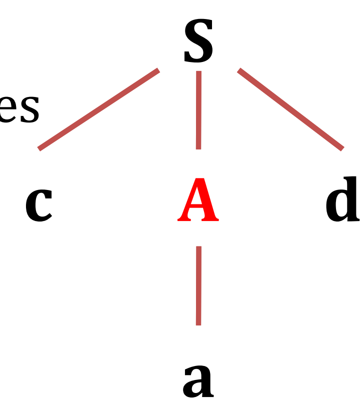
- ✓ **Create** a tree consisting of single node labeled S.
- ✓ Input pointer points to c (first symbol of w).
- ✓ Use first production for S to expand tree.



- The leftmost leaf labeled c matches with first symbol of w.
- Advance input pointer to a (second symbol of w).
- **Consider** next leaf labeled A.
- Expand A using first alternate for A (i.e. A → a b).
- Now second symbol is being matched.
- **Consider** third input symbol d, and next leaf labeled b.



- Since b does not match d, report failure and go back to A.
- **There** is another alternate for A (i.e. A → a), try to produce a match.
- During going back, reset input pointer to position 2.
- The leaf a matches the second symbol of w and leaf d matches third symbol.
- Now we have parse tree for w, halt and announce successful completion of parsing.



- There are several difficulties with top down parsing as previously presented.
- The concern is ***left recursion***.
- A grammar is left recursive if it has a nonterminal A such that there is a derivation $\mathbf{A} \xRightarrow{+} \mathbf{A}\,\boldsymbol{\alpha}$ for some α.
- A left recursive grammar can cause top down parser to go into infinite loop.
- Hence to use top down parsing, eliminate all left recursion from grammar.
- Another difficulty is ***backtracking***.
- Due to sequence of erroneous expansions and discovering mismatch, we may have to undo semantic effects of making erroneous expansions.

- Entries made in symbol table might have to be removed.
- These actions requires an overhead.
- The *recursive descent* and *predictive parsers* are types of top down parsers that avoid backtracking.

Elimination of Left Recursion

- Consider the grammar production

  $A \rightarrow A\alpha \mid \beta$ where β does not begin with A

- Then we eliminate left recursion by replacing this production with

  $\boldsymbol{A \rightarrow \beta A'}$

  $\boldsymbol{A' \rightarrow \alpha A' \mid \varepsilon}$

- This will eliminate all immediate left recursion (if no α is ε).
- To remove left recursion involving derivations of two or more steps, use following algorithm:

1. Arrange nonterminals of G in some order $A_1, A_2, A_3, \ldots, A_n$.
2. ***for*** i := 1 ***to*** n ***do***

    begin

    for j := 1 ***to*** i-1 ***do***

    replace each production of the form $A_i \rightarrow A_j \gamma$

    by the production $A_i \rightarrow \delta_1 \gamma \mid \delta_2 \gamma \mid \ldots \mid \delta_k \gamma$, where

    $A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$ are current A_j productions;

    eliminate the immediate left recursion among all A_j productions

    end

Example: Consider grammar

S → A a | b A → A c | S d | e

- Substitute S productions in A → S d.

  A → A c | A a d | b d | e

- Eliminate immediate left recursion among A productions yields following grammar

  S → A a | b A → b d A’ | e A’ A’ → c A’ | a d A’ | ε

Recursive Descent Parsing

- In many cases top down parser needs no backtrack.
- To implement this we must know, given current input symbol a and nonterminal A to be expanded from a set of A productions.
- A recursive descent parsing program consists of set of procedures for each nonterminal.
- Execution begins with procedure for start symbol which halts and announces success if its procedure scans entire input string.

```
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 current input symbol a )
    6)          advance the input to next symbol ;
    7)      else   /* error has occurred */ ;
        }
}
```

Fig: A typical procedure for nonterminal in a top down parser

- General recursive descent may require backtracking (it may require repeated scans over input).
- To allow backtracking, code needs to be modified.
- We cannot choose a unique A production at line 1, so try each of several productions in some order.

- Failure at line 7 is not ultimate failure.
- In such case suggest only that we need to return to line 1.
- Try another A production
- If there are no more A productions to try, declare that input error has been found.
- In order to try another A production, we need to reset input pointer to, where it was, when reached line 1.
- A local variable is needed to store this input pointer for future use.
- A left recursive grammar can cause a recursive descent parser, even one with backtracking, to go into infinite loop.
- That is when we try to expand a nonterminal A, we may find ourselves again trying to expand A without having consumed any input.

Left Factoring

- It is a grammar transformation that is useful for producing a grammar suitable for predictive or top down parsing.
- When the choice between two alternatives A productions is not clear, we may able to rewrite the productions to defer the decision until we have right choice.
- Example: Consider two productions

  stmt → **if** *expr* **then** *stmt* **else** *stmt* | **if** *expr* **then** *stmt*

- On seeing the input **if**, we may not sure which production to choose to expand *stmt*.
- If $A \rightarrow \alpha \beta_1 \mid \alpha \beta_2$ are two A productions, input begins with a nonempty string derived from α, we do not know whether to expand A to $\alpha \beta_1$ or $\alpha \beta_2$.

- We may differ the decision by expanding A to α A'.
- Then, after seeing the input derived from α, we expand A' to β_1 or to β_2.
- That is, left factored, the productions become

$$A \rightarrow \alpha A'$$

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

Algorithm: Left factoring a grammar

- Input : Grammar G.
- Output : An equivalent left factored grammar.
- Method : For each nonterminal A, find the longest prefix α common to two or more of its alternatives.
- If $\alpha \neq \varepsilon$, replace all A productions $A \rightarrow \alpha \beta_1 \mid \alpha \beta_2 \mid \ldots \mid \alpha \beta_n \mid \gamma$, where γ represents all alternatives that do not start with α

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

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

- Here A’ is a new nonterminal.
- Repeat this transformation.

Predictive Parsers

- A predictive parser is an efficient way of implementing recursive descent parsing by handling stack of activation records explicitly.

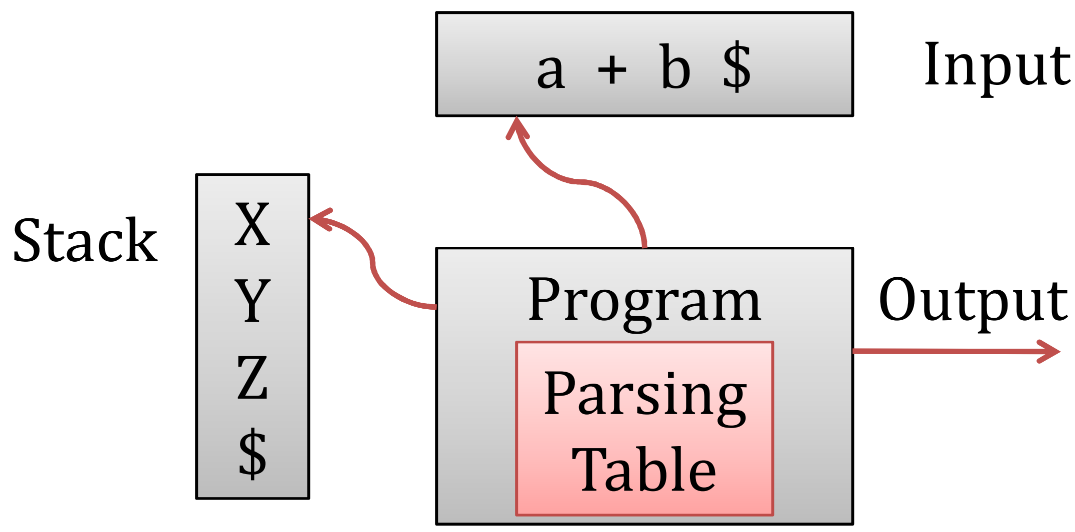


Fig: Model of predictive parser

- The predictive parser has an input tape, a stack, a parsing table, and an output.
- Input contains the string to be parsed, followed by $ (right endmarker).
- The stack contains sequence of grammar symbols, preceded by $ (bottom of stack marker).
- Initially stack contains start symbol of the grammar preceded by $.
- Parsing table is a two dimensional array M [A, a], where A is nonterminal, a is a terminal or the symbol $.
- Parser is controlled by a program. The program determines X (symbol on top of stack), and a (the current input symbol). These two symbols determine action of parser.

❖ There are three possibilities:

1. If X = a = $, parser halts and announces successful completion of parsing.
2. If X = a ≠ $, parser pops X off the stack and advances input pointer to next input symbol.
3. If X is a nonterminal, program consults entry M [X, a] of the parsing table M. This entry will be either an X production of grammar or an error entry.
   a. If M [X, a] = { X → U V W }, parser replaces X on top of stack by W V U (U on top).
   b. The grammar does semantic action associated with this production.
   c. If M [X, a] = ***error***, the parser calls an error recovery routine.

FIRST and FOLLOW

- The construction of top down and bottom up parsers is aided by two functions FIRST and FOLLOW associated with a grammar G.
- During top down parsing FIRST and FOLLOW allow us to choose which production to apply, based on next input symbol.
- FIRST and FOLLOW, indicate the proper entries in table for G, if such parsing table for G exists.
- Define FIRST (α), where α is any string of grammar symbols, to be set of terminals that begin strings derived from α.
- If $\boldsymbol{\alpha} \overset{*}{\Longrightarrow} \boldsymbol{\varepsilon}$, then ε is also in FIRST (α).
- Example: if $\mathbf{A} \overset{*}{\Longrightarrow} \mathbf{c}\,\boldsymbol{\gamma}$, then c is in FIRST (A).
- Define FOLLOW (A), for nonterminal A, to be set of terminals a that can appear immediately to the right of A in some sentential form.

- That is, $S \overset{*}{\Longrightarrow} \alpha A a \beta$ for some α and β.
- Note: there may have been symbols between A and a, at some time during the derivation, but if so, they derived ε and disappear.
- If A can be the rightmost symbol in some sentential form, then we add $ to FOLLOW (A).
- $ is a special “endmarker” symbol that is assumed not to be symbol of any grammar.
- To compute FIRST (X) for all grammar symbols X, apply following rules until no more terminal or ε can be added to any FIRST set.

1. If X is terminal, then FIRST (X) is { X }.
2. If X is nonterminal and $X \rightarrow Y_1 Y_2 \ldots Y_k$ is a production for some k >= 1, then place a in FIRST (X) if for some i, a is in FIRST (Y_i) and ε is in all of FIRST (Y_1), …, FIRST (Y_{i-1}); that is, $Y_1 Y_2 \ldots Y_{i-1} \overset{*}{\Longrightarrow} \varepsilon$

- If ε is in FIRST (Y_j) for all j = 1, 2, …., k, then add ε to FIRST (X).
- Example: everything in FIRST (Y_1) is surely in FIRST (X).
- If Y_1 does not derive ε, then we add nothing more to FIRST (X), but if $Y_1 \stackrel{*}{\Longrightarrow} \varepsilon$, then we add FIRST (Y_2) and so on.

3. If X → ε is a production, then add ε to FIRST (X).

- Now we can computer FIRST for any string $X_1 X_2 \ldots X_n$ as follows.
- Add to FIRST ($X_1 X_2 \ldots X_n$) all non ε symbols of FIRST (X_1).
- Also add non ε symbols of FIRST (X_2), if ε is in FIRST (X_1), the non ε symbols of FIRST (X_3), if ε is in FIRST (X_1) and FIRST (X_2) and so on.
- Finally add ε to FIRST ($X_1 X_2 \ldots X_n$) if, for all i, ε is in FIRST (X_i).

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

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

Example

❖ Consider a grammar

E → T E’

E’ → + T E’ | ε

T → F T’

T’ → * F T’ | ε

F → (E) | id

1. FIRST (E) = FIRST (T) = FIRST (F) = { (, id }
   - T has only one production and body start with F.
   - Two productions for F have bodies that start with terminal symbol id and).
   - Since F does not derive ε, FIRST (T) must be the same as FIRST (F).
2. FIRST (E’) = { +, ε }
   - one of two productions has body start with terminal +, and other’s body is ε.
   - whenever a nonterminal derives ε, we replace ε in FIRST for that nonterminal.

3. FIRST (T') = { *, ε }
    - Same as FIRST (E').
4. FOLLOW (E) = FOLLOW (E') = {), $ }
    - Since E is the start symbol, FOLLOW (E) must contain $.
    - For production body (E),) is in FOLLOW (E).
    - For E', this appears only at the end of bodies of E production. Thus FOLLOW (E') must be same as FOLLOW (E).
5. FOLLOW (T) = FOLLOW (T') = { +,), $ }
    - T appears in bodies only followed by E'. Thus everything except ε that is in FIRST (E') must be in FOLLOW (T); so it contains symbol +.
    - Since FIRST (E') contains ε, and E' is entire string following T in bodies of E productions, everything in FOLLOW (E) must also be in FOLLOW (T). So symbol $ and).
    - For T', since it appears only at the ends of T productions, it must be that FOLLOW (T') = FOLLOW (T).
6. FOLLOW (F) = { +, *,), $ }
    - Same as for (T) in step 5.

Construction of Predictive Parsing Table

- The algorithm considers two types of productions in the grammar.
- For non null production of for $A \rightarrow \alpha$, entry in parsing table will be

  $M[A, a] = \{A \rightarrow \alpha\}$, where $\{a\} \in$ FIRST (α)
- It means, parser will expand A by α when current input symbol (lookahead) is a.
- For all null productions of form $A \rightarrow \varepsilon$, entry in parsing table will be
  $M[A, a] = \{A \rightarrow \varepsilon\}$, where $\{a\} \in$ FOLLOW (A).
- That means, parser will use production $A \rightarrow \varepsilon$ to expand A, when current input symbol is a.

Algorithm

1. For each production $A \rightarrow \alpha$, do step 2 and 3.
2. For each terminal a in FIRST (α), add $A \rightarrow \alpha$ to M[A, a].
3. If ε is in FIRST(α), add $A \rightarrow \alpha$ to M[A, b] for each terminal b in FOLLOW (A).
   If ε is in FIRST (α) and {$} is in FOLLOW (A), add $A \rightarrow \alpha$ to M[A, $].
4. Make each undefined entry of M be error.

Example: Consider the following grammar and construct the predictive parsing table for it.

$S \rightarrow AaAb \mid BbBa$, $A \rightarrow \epsilon$, $B \rightarrow \epsilon$

Solution:

To construct parsing table, find FIRST and FOLLOW:

- FIRST (S) = {a, b}
- FIRST (A) = {ϵ}
- FIRST (B) = {ϵ}
- FOLLOW (S) = {$}
- FOLLOW (A) = {a, b}
- FOLLOW (B) = {b, a}

	a	b	$
S	$S \rightarrow AaBb$	$S \rightarrow BbAa$	---
A	$A \rightarrow \epsilon$	$A \rightarrow \epsilon$	---
B	$B \rightarrow \epsilon$	$B \rightarrow \epsilon$	---

LR Parser

- This concept is used in bottom up parser.
- “L” is for left to right scanning of input, “R” for constructing a rightmost derivation in reverse.
- LR parser are table driven.
- For a grammar to be LR, it is sufficient that a left to right shift reduce parser be able to recognize handles of right sentential forms when they appear on top of the stack.
- LR parser can be constructed to recognize all programming language constructs for which CFG can be written.
- LR parsing method is most general backtracking shift reduce parsing method.
- LR parser can detect a syntactic error as soon as it is possible to do so on a left to right scan of input.
- Drawback is that it is too much work to implement LR parser by hand for a typical programming language grammar.
- A specialized tool, LR parser generator is needed.

Generating LR Parser

- LR parser consists of two parts, a driver routine and a parsing table.
- Driver routine is same for all parser, only parsing table changes from one parser to another.

Fig: Generating the parser

Fig: Operation of parser

- There are many different parsing tables that can be used in LR parser for a given grammar.
- Some parsing tables may detect errors sooner than others, but they all accept same sentences generated by grammar.

Techniques for Constructing LR Parsing Table

1. Simple LR (SLR in short)
   - It is easiest to implement.
   - It may fail to produce table for certain grammars on which other method succeed.
2. Canonical LR
   - It is most powerful and work on a large class of grammars.
   - It can be very expensive to implement.
3. Lookahead LR (LALR)
   - It is intermediate in power between SLR and Canonical LR.
   - It work on most programming language grammars, and with some effort, can be implemented efficiently.

LR Parsers

- The parser has an input, a stack and a parsing table.
- The input is read from left to right, one symbol at a time.
- The stack contains a string of form $s_0\ X_1\ s_1\ X_2\ \ldots..\ X_m s_m$, where s_m in on top.

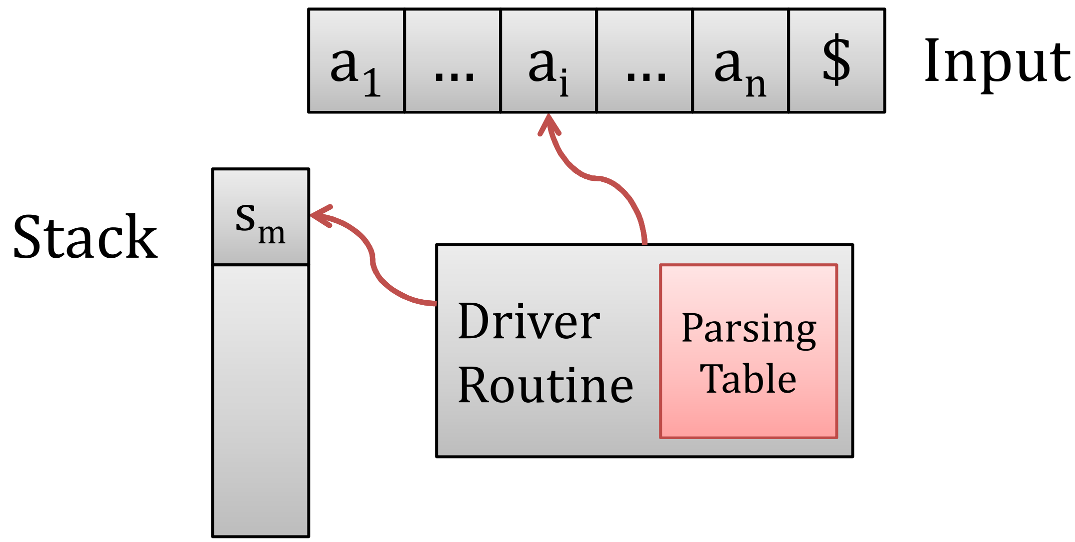


Fig: LR parser

- Each X_i is a grammar symbol and each s_i is a symbol called a state.
- Each state symbol summarizes the information contained in stack below it.
- And is used to guide shift reduce decision.

- In actual implementation, grammar symbols need not appear on the stack.
- We include them to help explain the behavior of LR parser.
- Parsing table consists of two parts:
  1. Parsing action function ACTION
  2. Goto function GOTO
- The program driving LR parser behaves as follows:
  - It determines s_m, state currently on top of stack, and a_i, current input symbol.
  - Then consult ACTION [s_m, a_i], parsing action table entry for state s_m and input a_i.
  - The entry ACTION [s_m, a_i] can have four values:
    1. Shift s
    2. Reduce A $\rightarrow$ β
    3. Accept
    4. error

- The function GOTO takes a state and grammar symbol as arguments and produces a state.
- It is essentially the transition table of DFA whose input symbols are terminals and nonterminals of grammar.

Working of LR Parser

- The configuration of LR parser is a pair whose first component is stack contents and second one is unexpected input:

$$(s_0\, X_1\, s_1\, X_2\, s_2 \ldots X_m\, s_m,\ a_i\, a_{i+1} \ldots a_n\, \$)$$

- The next move of parser is determined by reading a_i (current input symbol), and s_m (state on top of stack).
- Then consult parsing action table entry ACTION [s_m, a_i].
- Resulting configuration after each of four types of moves are as follows:

1. If ACTION [s_m, a_i] = shift s, parser executes shift move, entering the configuration

$$(s_0\, X_1\, s_1\, X_2\, s_2 \ldots X_m\, s_m\, \mathbf{a_i}\, \mathbf{s},\ a_{i+1} \ldots a_n\, \$)$$

- Here parser shifted current input symbol and next state s = GOTO [s_m, a_i] onto the stack; a_{i+1} becomes new current input symbol.

2. If ACTION [s_m, a_i] = reduce A $\rightarrow$ β, parser executes a reduce move, entering the configuration

$$(s_0\ X_1\ s_1\ X_2\ s_2 \ldots X_{m-r}\ s_{m-r}\ \mathbf{A\ s},\ a_i\ a_{i+1} \ldots a_n\ \$)$$

Where s = GOTO [s_{m-r}, A] and r is length of β (right side of production).

- Here parser first popped 2r symbols off the stack (r stack and r grammar symbol) exposing state s_{m-r}.
- Parser then pushed both A (left side of production) and s, the entry for ACTION [s_{m-r}, A] onto the stack.
- The current input symbol is not changed in reduce move.
- For LR parser we shall construct, Xm-r+1 . . . Xm, the sequence of grammar symbols popped off the stack, will always match β (right side of the reducing production).

3. If ACTION [s_m, a_i] = accept, parsing is completed.

4. If ACTION [s_m, a_i] = error, parser has discovered an error and calls error recovery routine.

- Initially LR parser is in configuration (s_0, a_1 a_2 . . . a_n \$) where s_0 is a designated initial state and a_1 a_2 . . . a_n is string to be parsed.
- The parser execute moves until an accept or error action is encountered.
- All parsers works in this concept.
- Difference between one LR parser and another is the information in parsing action and goto fields of parsing table.

Example

Consider the grammar

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

The codes for actions are:

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

Parsing table is as follows

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

Fig: Parsing table for expression grammar

- The value of GOTO [s, a] for terminal a found in ACTION field connected with shift action on input a for state s.
- The GOTO field gives GOTO [s, A] for nonterminals A.
- On input **id * id + id**, the sequence of input contents is shown.
- Also, the sequence of grammar symbols corresponding to states held on stack.

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

Fig: Moves of an LR parser on **id * id + id**

- At line (1), the LR parser is in state 0, initial state with no grammar symbol.
- **id** is the first input symbol.
- Action in row **0** and column **id** is **s5**, means shift by pushing state 5.
- At line (2), the state symbol 5 has been pushed onto the stack, and **id** is removed from the input.
- Hence * becomes current input symbol.
- Now action in row **5** and column * is **r6**, means reduce by production 6 i.e. F → id.
- One state symbol is popped of the stack.
- State 0 is then exposed.
- Since goto of state **0** on **F** is 3, state 3 is pushed onto the stack.
- In this way remaining moves are determined.

LR Grammars

- A grammar for which we can construct a parsing table in which every entry is uniquely defined is said to be an LR grammar.
- There are context free grammars which are not LR.
- In order for grammar to be LR, it is sufficient that a left to right parser be able to recognize handles when they appear on top of the stack.
- An LR parser does not have to scan entire stack to know when the handle appears on top.
- The state symbol on top of stack contains all the information it needs.
- If it is possible to recognize a handle in the stack, then a finite automaton can determine what handle is on top of stack if any (by reading stack from bottom to top).
- The driver routine of LR parser is such a finite automaton.

- It need not read the stack on every move.
- State symbol stored on top of stack is the state in which handle recognizing finite automata would be if it had read stack from bottom to top.
- So, LR parser can determine from state on top of the stack everything that it needs to know about what is in the stack.
- --------------------
- An LR parser can use to help make its shift reduce decision is the next k input symbols.
- K=0 or k=1 is sufficient.
- A grammar that can be parsed by an LR parser examining up to k input symbols on each move is called LR (k) grammar.

The Canonical Collection of LR (0) Items

- It shows how to construct a simple LR parser for a grammar.
- The idea is construction of DFA from the grammar.
- The DFA recognizes *viable prefixes* of the grammar, i.e. prefixes of right sentential forms that do not contain any symbols to the right of the handle.
- It is so called since it is always possible to add terminal symbols to the end of viable prefixes to obtain a right sentential form.
- We shall study about how does a shift reduce parser know when to shift and when to reduce?
- An LR parser makes shift reduce decisions by maintaining state to keep track of where we are in parse tree (our position in parse tree).
- States represent sets of “items”.

- An LR (0) item (item in short) of a grammar G is a production of G with a dot at some position of the right side (body).
- Production A → X Y Z yields four items (LR (0) item)
  - A → . X Y Z
  - A → X . Y Z
  - A → X Y . Z
  - A → X Y Z .
- The production A → ϵ generates only one item, A → .
- Items are easily represented by pairs of integers, the first shows no. of productions and second the position of dot.
- An item indicates how much a production we have seen at a given point in parsing process.

Example

- Item A → . X Y Z indicates that we hope to see a string derivable from XYZ next on the input.
- Item A → X . Y Z indicates that we have just seen on the input a string derivable from X and that we hope next to see a string derivable from YZ.
- Item A → X Y Z . indicates that we have seen the body XYZ and that it may be time to reduce XYZ to A.

- A parser generator that produces a bottom up parser may need to represent items and sets of items.
- An item can be represented by a pair of integers, the first of which is the number of one of the production of the underlying grammar.
- And second of which is position of the dot.
- We group items together into sets, which give rise to states of an LR parser.
- One collection of sets of items, which is called *canonical LR (0)* collection, provides the basis for constructing a class of LR parsers called simple LR (SLR).
- To construct canonical LR (0) collection for a grammar we need to define an augmented grammar and two functions, CLOSURE and GOTO.

- If G is a grammar with start symbol S, then G' (the augmented grammar for G) is a grammar G with a new start symbol S' and production **S' → S**.
- Purpose of new starting production is to indicate to the parser when it should stop parsing and announce acceptance of input.
- Acceptance occurs when parser is about to reduce by **S' → S**.

CLOSURE

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

1. Initially, add every item in I to CLOSURE (I).
2. If **A → α . B β** is in CLOSURE (I) and **B → γ** is a production, then add item **B → . γ** to CLOSURE (I), if it is not already there.

- Apply this rule until no more items can be added to CLOSURE (I).

- **A → α . B β** in CLOSURE (I) indicates that, at some point in parsing process, we next expect to see a string derivable from **Bβ** as input.
- The substring derivable from **Bβ** will have a prefix derivable from B by applying one of the B productions.
- So we add items for all B productions, i.e. if **B → γ** is a production, we also include **B → . γ** in CLOSURE (I).

Example

- Consider augmented expression grammar:

  E' → E   **T → T * F | F**

  E → E + T | T   **F → (E) | id**

- If I is the set of one item { [E' → . E] }, then CLOSURE (I) contains items

  E' → . E

  E → . E + T

  E → . T

  T → . T * F

  T → . F

  F → . (E)

  F → . id

- **E' → . E** is put in CLOSURE (I) by rule 1.
- Since there is an E immediately to the right of dot (.) we add the E productions with dots at the left ends: **E → . E + T** and **E → . T**
- Now there is a T immediately to the right of dot, so we add T productions with dots at left end: **T → . T * F** and **T → . F**
- F is immediately to the right of dot, so add **F → . (E)** and **F → . id**
- Now no other items need to be added.

- A convenient way to implement CLOSURE is to keep a boolean array *added*, indexed by the nonterminals of G.
- added [B] is set to **true** if and when we add items **B → . γ** for each B productions B → γ

```
SetOfItems  CLOSURE ( I )
{
    repeat
        for ( each item A → α . B β in I )
            for ( each production B → γ of G )
                if ( B → . γ is not in I )
                    add B → . γ to I ;
    until no more items are added to I ;
    return I ;
}
```

- If one B production is added to the closure of I with dot at left end, then all B productions will be similarly added to the CLOSURE.

GOTO

- The second function is GOTO (I, X) where I is a set of items and X is a grammar symbol.
- GOTO (I, X) is defined to be the closure of set of all items **{A → αX.β}** such that [A → α.Xβ] is in I.
- The GOTO function is used to define the transition in LR (0) automaton for a grammar.
- State of automaton correspond to sets of items.
- GOTO (I, X) specifies the transition from the state for I under X.

Example

- If I is the set of two items { [E' → E.], [E → E . + T] }, then GOTO (I, +) contains the items

  E → E + . T

  T → . T * F

  T → . F

  F → . (E)

  F → . id

- We compute GOTO (I, +) by examining I for items with + immediately to right of dot.
- E' → E. is not such an item, but E → E. + T is.
- We move dot over + to get E → E + . T and then take closure of this set.

Sets of Items Construction

- Algorithm for constructing C (canonical collection of sets of LR (0) items for augmented grammar G')

```
void items (G')
{
    C = CLOSURE ( { [ S' → . 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 on a round;
}
```

Example

- The canonical collection of sets of items for grammar given below:

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

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

is:

I_0: $E' \rightarrow .E$
$E \rightarrow .E + T$
$E \rightarrow .T$
$T \rightarrow .T * F$
$T \rightarrow .F$
$F \rightarrow .(E)$
$F \rightarrow .id$

I_1: $E' \rightarrow E.$
$E \rightarrow E. + T$

I_2: $E \rightarrow T.$
$T \rightarrow T. * F$

I_3: $T \rightarrow F.$

I_4: $F \rightarrow (.E)$
$E \rightarrow .E + T$
$E \rightarrow .T$
$T \rightarrow .T * F$
$T \rightarrow .F$
$F \rightarrow .(E)$
$F \rightarrow .id$

I_5: $F \rightarrow id.$

I_6: $E \rightarrow E + .T$
$T \rightarrow .T * F$
$T \rightarrow .F$
$F \rightarrow .(E)$
$F \rightarrow .id$

I_7: $T \rightarrow T * . F$
$F \rightarrow . (E)$
$F \rightarrow . id$

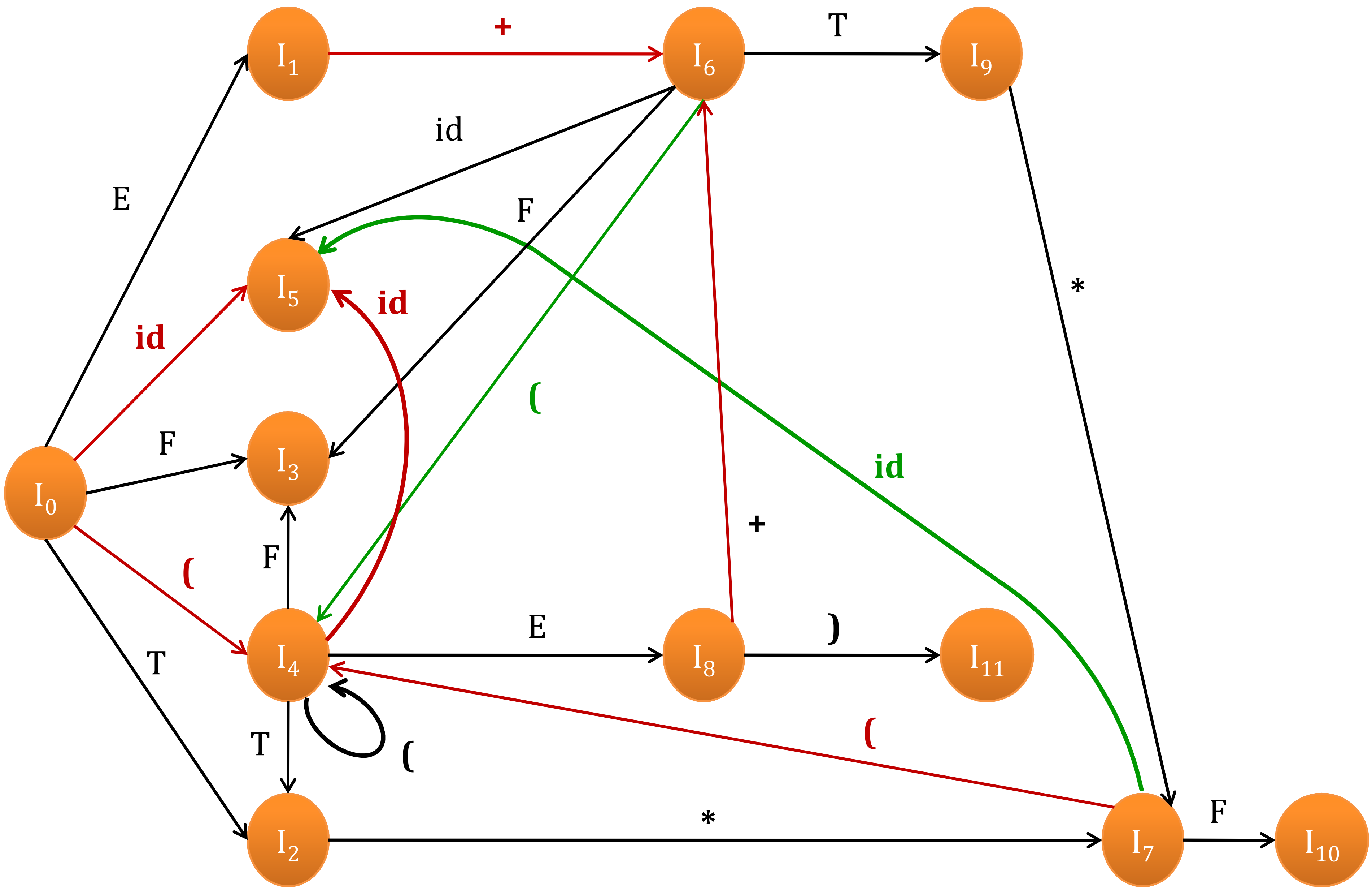
I_8: $F \rightarrow (E .)$
$E \rightarrow E . + T$

I_9: $E \rightarrow E + T .$
$T \rightarrow T . * F$

I_{10}: $T \rightarrow T * F .$

I_{11}: $F \rightarrow (E) .$

Fig: Deterministic finite automaton D



- If each state of D is a final state and I_0 is the initial state then D recognizes exactly viable prefixes of grammar.
- For every grammar G, the GOTO function of canonical collection of sets of items define a DFA that recognizes the viable prefixes of G.
- There is a transition from $A \rightarrow \alpha.X\beta$ to $A \rightarrow \alpha X.\beta$ labeled X.
- There is a transition from $A \rightarrow \alpha.B\beta$ to $B \rightarrow .\gamma$ labeled ϵ.
- CLOSURE (I) for set of items (states of N) I is exactly ϵ CLOSURE of a set of NFA states.
- GOTO (I,X) gives transition from I on symbol X in the DFA constructed from N by subset construction.

Constructing SLR Parsing Table

- The SLR method begins with LR(0) items and LR(0) automata.
- That is, given a grammar G, we augment G to produce G' with a new start symbol S'.
- From G', we construct C, the canonical collection of sets of items for G' together with GOTO function.
- The ACTION and GOTO entries in the parsing table are then constructed using algorithm.
- **Algorithm** (construction of SLR parsing table):
- Input: C, canonical collection of sets of items for augmented grammar G'.
- Output: SLR parsing table functions ACTION and GOTO for G'.

Method

1. Construct C = { I_0, I_1, …. , I_n }, the collection of sets of LR (0) items for grammar G'.
2. State i is constructed from I_i. The parsing actions for state i are determined as follows:
   a) If [A → α.aβ] is in I_i, and GOTO (I_i, a) = I_j, then set ACTION [i, a] to "shift j"; here a must be terminal.
   b) If [A → α.] is in I_i, then ACTION [i,a] to "reduce A → a" for all a in FOLLOW (A); here A may not be S'.
   c) If [S' → S.] is in I_i, then set ACTION [i, $] to "accept".

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

3. The goto transition for state i are constructed for all nonterminals A using the rule: if GOTO(I_i, A) = I_j, then 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 set of items containing [S’ → .S].

- The parsing table consisting of parsing action and goto functions obtained by this algorithm is called SLR table for G.
- An LR parser using SLR table for G is called SLR parser for G.
- A grammar having SLR parsing table is said to be SLR(1).

Example

- Construct SLR table for grammar given below:

E → E + T | T

T → T * F | F   F → (E) | id

The canonical collection of sets of items for grammar is shown as:

I_0: E' → . E
E → . E + T
E → . T
T → . T * F
T → . F
F → . (E)
F → . id

I_1: E' → E .
E → E . + T

I_2: E → T .
T → T . * F

I_3: T → F .

I_4: F → (. E)
E → . E + T
E → . T
T → . T * F
T → . F
F → . (E)
F → . id

I_5: F → id .

I_6: E → E + . T
T → . T * F
T → . F
F → . (E)
F → . id

I_7: T → T * . F
F → . (E)
F → . id

I_8: F → (E .)
E → E . + T

I_9: E → E + T .
T → T . * F

I_{10}: T → T * F .

I_{11}: F → (E) .

I_0: E' → . E
E → . E + T
E → . T
T → . T * F
T → . F
F → . (E)
F → . id

- Consider I_0:
- The item F → . (E) gives rise to entry

  ACTION [0,( ] = shift 4
- Item F → . id do the entry

  ACTION [0, id] = shift 5

I_1: $E' \rightarrow E\,.$
$E \rightarrow E\,.+T$

- Consider I_1:
- The first item yields **ACTION [1, $] = accept**
- Second item yields **ACTION [1, +] = shift 6**

I_2: $E \rightarrow T\,.$
$T \rightarrow T\,.*F$

- Consider I_2:
  Since FOLLOW (E) = { $, +,) } the first item makes **ACTION [2, $] = ACTION [2, +] = ACTION [2,)] = reduce E → T**
- The second item makes **ACTION [2, *] = shift 7**.
- And so on….
- Parsing table is shown in next slide.

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

Fig: Parsing table for expression grammar

Constructing Canonical LR Parsing Tables

- In SLR method, state i calls for reduction by A → α on input symbol a if set of items I_i contains [A → α.] and a is in FOLLOW(A).
- In some cases, when state i appears on top of stack, a viable prefix βα may be on stack such that βA cannot be followed by a in right sentential form.
- So reduction A → α would be invalid for a.
- The extra information is incorporated into state by redefining items to include a terminal symbol as second component.
- General form of an item becomes [A → α.β, a], where A → αβ is a production and a is a terminal of right endmarker $.
- We say this an LR(1) item.
- 1 refers to the length of second component (lookahead of item).

- The lookahead has no effect in an item of form [$A \rightarrow \alpha . \beta$, a], where β is not ε, but item of the form [$A \rightarrow \alpha .$, a] calls for a reduction by $A \rightarrow \alpha$ only if next input symbol is a.
- So, we are bounded to reduce by $A \rightarrow \alpha$ only on those input symbols a for which [$A \rightarrow \alpha.$, a] is an LR(1) item in state on top of stack.
- We say LR(1) item [$A \rightarrow \alpha.\beta$, a] is valid for a viable prefix γ if there is a derivation

$$S \overset{*}{\underset{rm}{\Longrightarrow}} \delta A w \underset{rm}{\Longrightarrow} \delta\alpha\beta w$$

where

- $\gamma = \delta\alpha$, and
- Either a is first symbol of w, or w is ε and a is \$.

Example

- Consider the grammar

  S → B B

  B → a B | b

- There is a rightmost derivation

$$S \overset{*}{\underset{rm}{\Longrightarrow}} aaBab \underset{rm}{\Longrightarrow} aaaBab$$

- We see that item [B → a.B, a] is valid for a viable prefix γ = aaa, by letting δ = aa, A = B, w = ab, α = a, and β = B
- There is also a rightmost derivation

$$S \overset{*}{\underset{rm}{\Longrightarrow}} Bab \underset{rm}{\Longrightarrow} BaaB$$

- From this derivation we see that item [B → a.B, $] is valid for viable prefix Baa.

Constructing LR (1) Sets of Items

- **Input**: a grammar G.
- **Output**: sets of LR (1) items which are sets of items valid for one or more viable prefixes of G.
- **Method**:
- The method is same as for building the canonical collection of sets of LR (0) items.
- Needed modification in only two procedures CLOSURE and GOTO.

```
SetOfItems  CLOSURE ( I )
{
    repeat
        for ( each item [A → α . B β, a] in I )
            for ( each production B → γ of G' )
                    for ( each terminal b in FIRST ( βa ) )
                            add [B → . γ, b] to set I ;
    until no more items are added to I ;
    return I ;
}
```

```
SetOfItems  GOTO ( I )
{
    initialize J to be the empty set ;
        for ( each item [A → α . X β, a] in I )
            add item [A → α X . β, a] to set J ;
    return CLOSURE ( J ) ;
}
```

```
void  items ( G' )
{
    initialize C to CLOSURE ( { [ S' → . 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 following augmented grammar

  S’ → S

  S → C C

  C → c C | d

- We begin by computing closure of { [S’ → . S, $] }.
- We match the item [S’ → . S, $] with item [A → α.Bβ, a] in procedure CLOSURE.
- So A = S’, α = ε, B = S, β = ε, and a = $.
- CLOSURE tells to add [B → . γ, b] for each production B → γ and terminal b in FIRST (βa).
- So add [S → . CC, $]

- We continue to compute closure by adding all items [C → . γ, b] for b in FIRST (C$).
- So matching [S → . C C, $] with [A → α . B β, a], we have A = S, α = ε, B = C, β = C, and a = $.
- Since C does not derive empty string , FIRST (C$) = FIRST (C).
- Since FIRST (C) contains terminals c and d, add items [C → . c C, c], [C → . c C, d], [C → . d, c] and [C → . d, d].
- None of new items has a nonterminal immediately to the right of dot, so we completed first set of LR (1) items.
- I_0: S' → . S, $

  S → . C C, $

  C → . c C, c | d

  C → . d, c | d

- Now we compute GOTO (I_0, X) for various values of X.
- For X = S we must close the item [S' → S ., $].
- Since dot is at right end, no additional closure is possible. So
- I_1: S' → S . , $
- For X = C we close item [S → C . C, $].
- Add C productions with second component $ and hence
- I_2: S → C . C , $

  C → . c C , $

  C → . d , $
- Now X = c, we close { [C → c . C, c | d] }.
- Add C productions with second component c | d, hence

- I_3: C → c . C , c | d

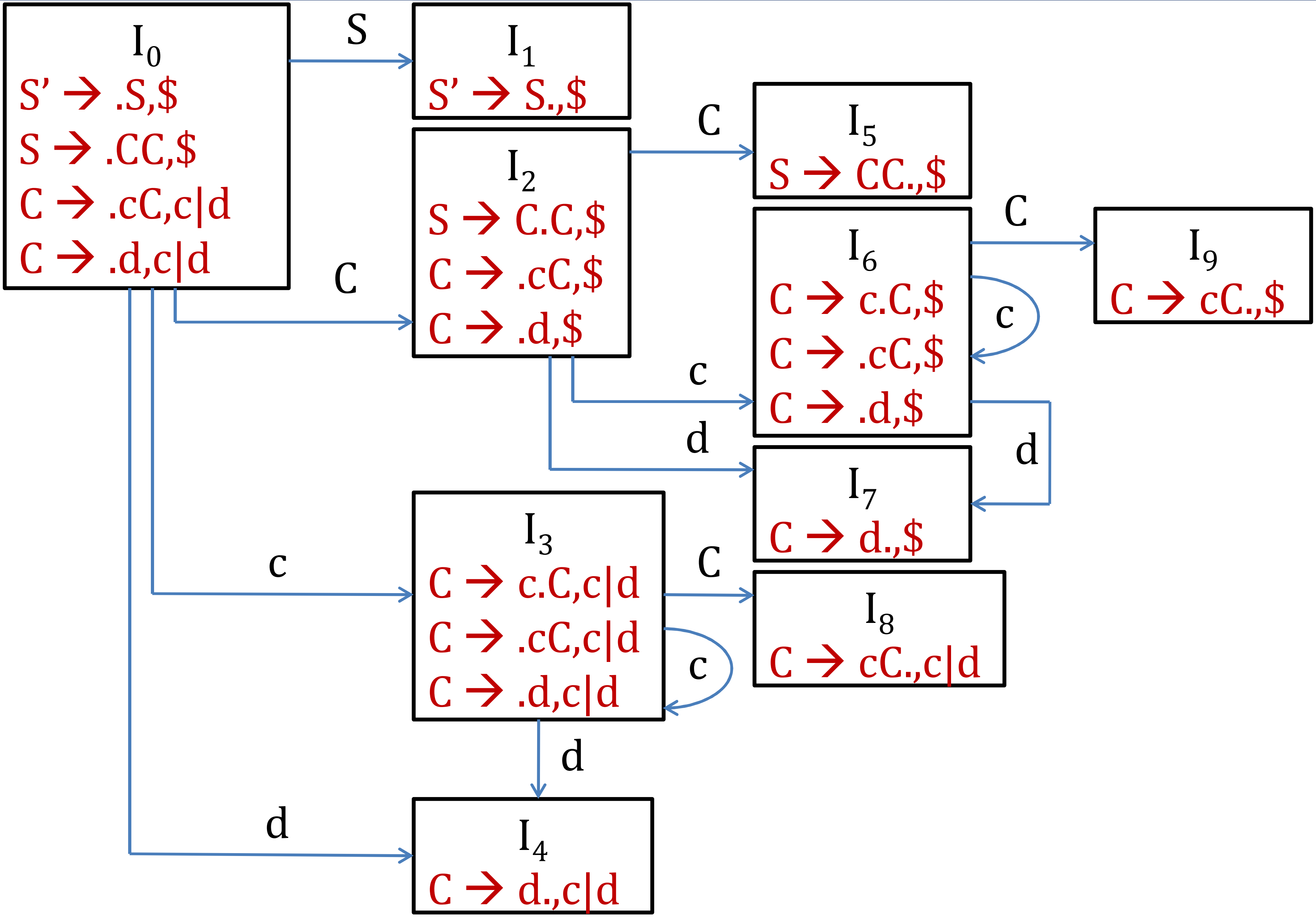
  C → . c C , c | d

  C → . d , c | d
- Let X = d, then
- I_4: C → d . , c | d
- Finished considering GOTO on I_0 and no new sets from I_1.
- I_2 has goto's on C, c and d. For GOTO (I_2, C) we get
- I_5: S → C C . , $
- To compute GOTO (I_2, c) we take closure of { [C → c . C, $] }.
- I_6: C → c . C , $

  C → . c C , $

  C → . d , $
- I_6 differs from I_3 only in second component.

- GOTO function for I_2, GOTO (I_2, d) is:
- I_7: C → d . , $
- GOTO's of I_3 on c and d are I_3 and I_4 resp. GOTO (I_3, C) is:
- I_8: C → c C . , c | d
- I4 and I5 have no GOTO's, since all items have their dots at right end.
- GOTO's of I6 on c and d are I6 and I7, resp. GOTO (I6, C) is:
- I_9: C → c C . , $
- Remaining sets of items yield no GOTO's.
- GOTO graph for ten sets of items is shown in next slide.



I0
S' → .S,$
S → .CC,$
C → .cC,c|d
C → .d,c|d
S
I1
S' → S.,$
C
I2
S → C.C,$
C → .cC,$
C → .d,$
C
I5
S → CC.,$
c
I6
C → c.C,$
C → .cC,$
C → .d,$
C
I9
C → cC.,$
c
d
d
I7
C → d.,$
c
I3
C → c.C,c|d
C → .cC,c|d
C → .d,c|d
C
I8
C → cC.,c|d
c
d
d
I4
C → d.,c|d

Canonical LR (1) Parsing Tables

Algorithm: Construction of canonical LR parsing tables.

- **Input**: An augmented grammar G'.
- **Output**: The canonical LR parsing table functions ACTION and GOTO for G'.
- **Method**:

1. Construct C' = { I_0, I_1, …, I_n }, collection of sets of LR (1) items for G'.
2. State i of the parser is constructed from I_i. Parsing action for state i is determined as follows:
   a) If [A → α . a β , b] is in I_i and GOTO (I_i, a) = I_j,
   then set ACTION [i, a] to "**shift j**". a must be terminal.
   b) If [A → α . , a] is in Ii, A ≠ S',
   then set ACTION [i , a] to "**reduce A → α**".

c) If [S’ → S . , $] is in I_i then set ACTION [i, $] to “**accept**”.

In case of conflicting actions resulting from above rules, we say that grammar is not LR (1). And algorithm fails to produce parser.

3. The goto transitions for state i are constructed for all nonterminals A using rule : if GOTO (I_i , A) = I_j, then 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’ → . S , $].

- The table formed by using this parsing action and goto fnction is called *canonical LR (1) parsing table*.
- LR parser using this table is called *canonical LR (1) parser*.

Example

❖ The canonical parsing table for grammar
$S \rightarrow C\ C \qquad C \rightarrow c\ C \mid d$ is shown in table.

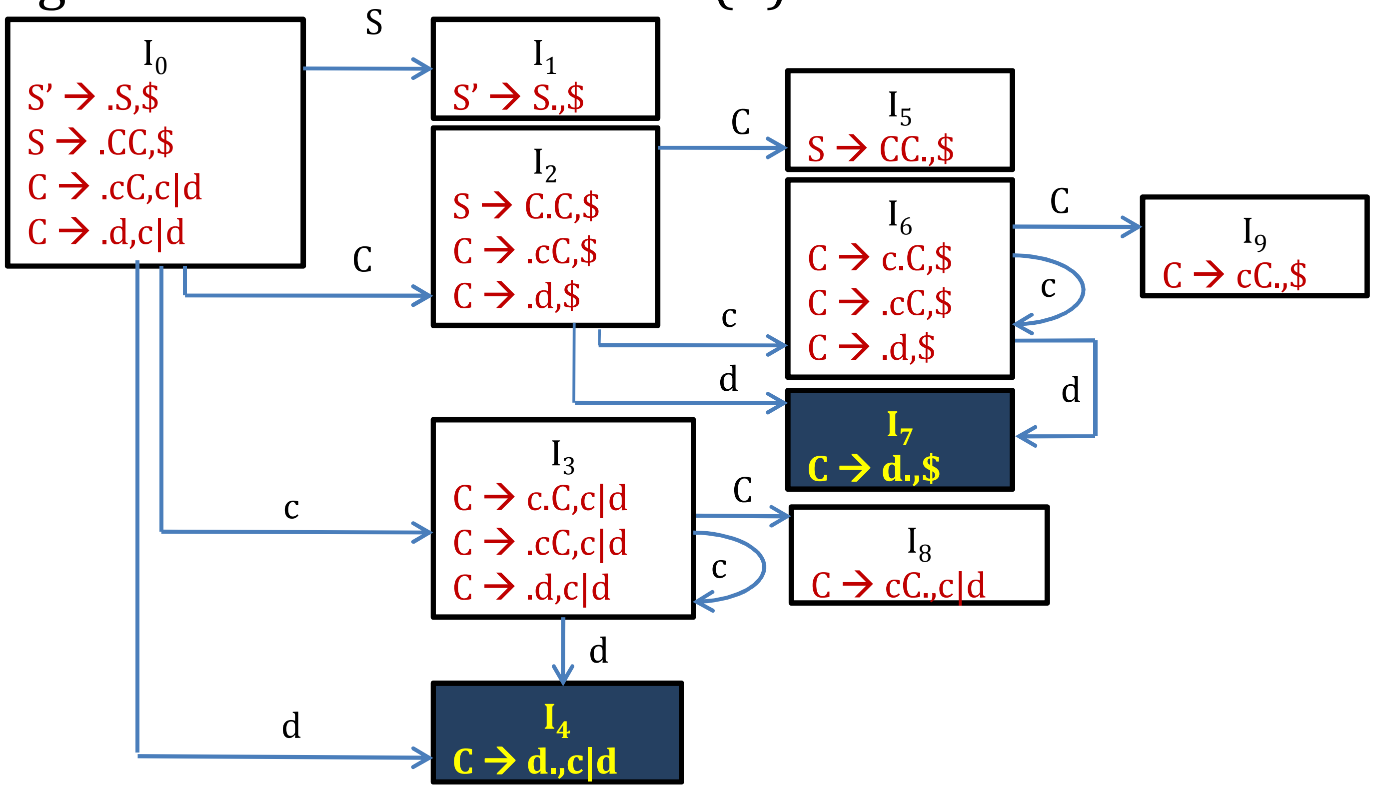
STATE	ACTION			GOTO	
	c	**d**	**$**	**S**	**C**
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

- Every SLR (1) grammar is an LR (1) grammar.
- But for an SLR (1) grammar the canonical LR parser may have more states than SLR parser for same grammar.

Constructing LALR Parsing Tables

- This method is often used, since tables obtained by it are smaller than canonical LR tables.
- Most common syntactic constructs of programming languages can be expressed conveniently by LALR grammar.
- Same is true for SLR grammar but a few constructs cannot be handled by SLR technique.
- SLR and LALR tables have same number of states (several hundred states for language like C).

- Canonical LR table would have several thousand states for same language.
- So it is easier to construct SLR and LALR tables than canonical LR tables.
- Consider the grammar whose sets of LR (1) items are



- Take a pair of similar looking states like I_7 and I_4.
- Both states has only items with first component $C \rightarrow d$.
- In I_4 lookaheads are **c or d** whereas only **$** for I_7.
- Let us replace I_4 and I_7 by I_{47} (union of I_4 and I_7), consisting of set of three items represented by $[\, C \rightarrow d\,.\,,\ c \mid d \mid \$\,]$.
- The goto's on d to I_4 or I_7 from I_0, I_2, I_3 and I_6 now enter I_{47}.
- The action of state 47 is to reduce on any input.
- Similarly we can look for sets of LR (1) items having the same core, that is, set of first components.
- We may merge these sets with common cores into one set of items.
- A core is a set of LR (0) items for the grammar at hand and an LR (1) grammar may produce more than two sets of items with same core.

Example

Consider the grammar

$S' \rightarrow S$

$S \rightarrow a\ A\ d \mid b\ B\ d \mid a\ B\ e \mid b\ A\ e$

$A \rightarrow c \qquad\qquad B \rightarrow c$

- Generating four strings ***acd, bcd, ace, bce***.
- The grammar is LR (1).
- By constructing set of items, we get
- $\{ [A \rightarrow c\ .\ , d] \}, \{ [B \rightarrow c\ .\ , e] \}$ valid for viable prefix ac
- $\{ [A \rightarrow c\ .\ , e] \}, \{ [B \rightarrow c\ .\ , d] \}$ valid for bc.
- Their cores are same. So their union is

  $A \rightarrow c\ .\ , d \mid e \qquad B \rightarrow c\ .\ , d \mid e$

- There is a conflict, since reductions by both A → c and B → c are called for inputs d and e.
- So we prepared two LALR table.
- Construct set of LR (1) items, and if no conflict arise, merge sets with common cores.
- Construct parsing table from collection of merged sets of items.
- No. of sets of items will be same as no. of sets of LR (0) items.
- Constructing collection of LR (1) sets of items requires too much space and time to be used in practice.

Algorithm : LALR table construction.

Input: A grammar G augmented by production S' → S.

Output: The LALR parsing tables ACTION and GOTO.

Method

1. Construct C = { I_0, I_1, …., I_n }, collection of sets of LR (1) items.
2. For each core present among the sets of LR (1) items, find all sets having that core, and replace these sets by their union.
3. Let C' = { J_0, J_1, …., J_n } be the resulting sets of LR (1) items. The parsing action of state i are constructed from J_i, same as canonical LR parsing table.

   if parsing action conflict, algorithm fails to produce a parser and grammar is said not to be LALR (1).

4. The GOTO table is constructed as: if J is union of one or more sets of LR (1) items, $\mathbf{J = I_1 \cup I_2 \cup \ldots \cup I_m}$, then cores of **GOTO ($I_1$, X), GOTO ($I_2$, X), ..., GOTO ($I_k$, X)** are same, since **I_1, I_2, ..., I_k** are having same core.

   let K be the union of all sets of items having same core as **GOTO (I_1, X), then GOTO (J, X)** = K.

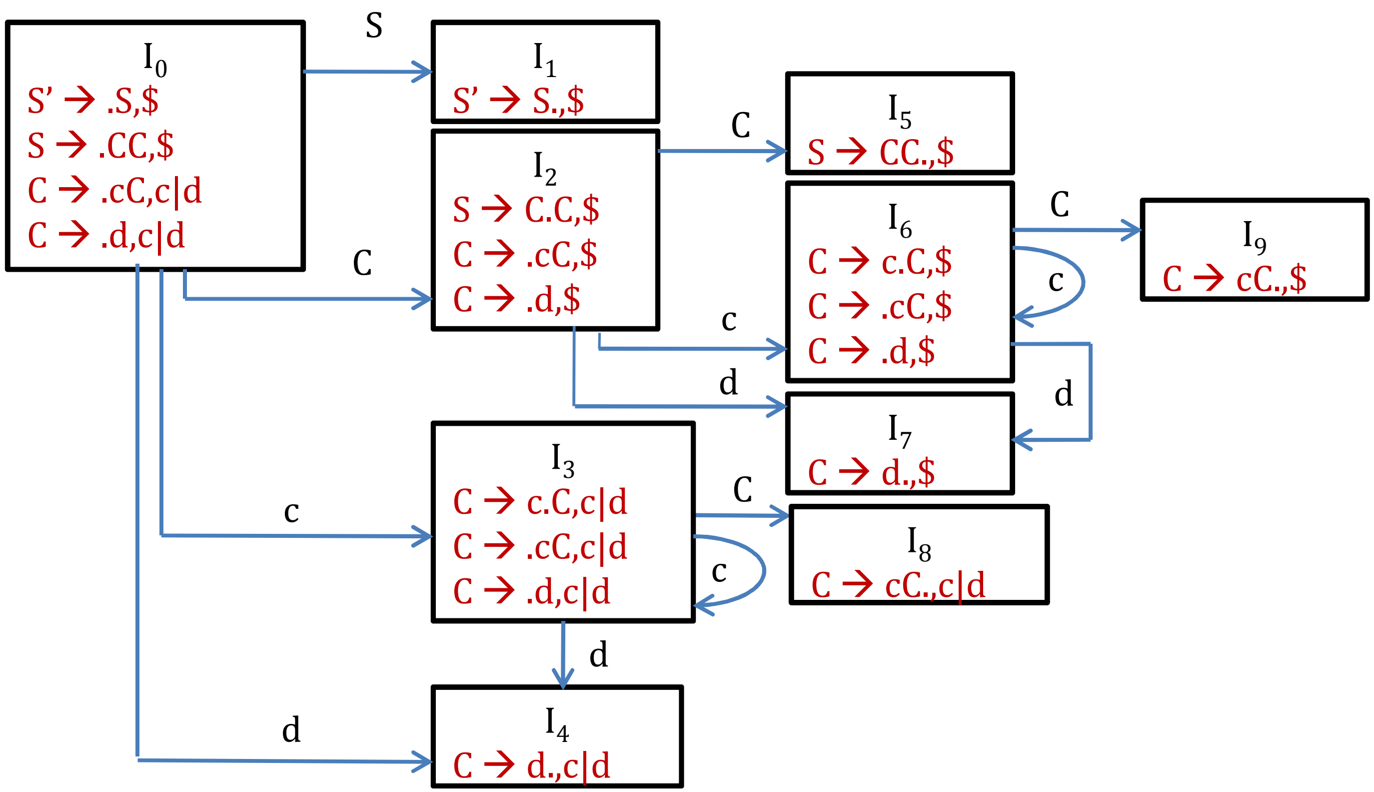
- In this way LALR parsing table produced for G.
- If there are no parsing action conflicts, then the given grammar is said to be an LALR (1) grammar.

Example

Consider the grammar

S' → S   S → C C   C → c C | d

Whose GOTO graph is shown below



- There are 3 pairs of sets of items that can be merged.
- I_3 and I_6 are replaced by I_{36}
- I_{36}: $C \rightarrow c\ .\ C\ ,\ c\ |\ d\ |\ \$$

  $C \rightarrow .\ c\ C\ ,\ c\ |\ d\ |\ \$$

  $C \rightarrow .\ d\ ,\ c\ |\ d\ |\ \$$
- I_4 and I_7 are replaced by I_{47}
- I_{47}: $C \rightarrow d\ .\ ,\ c\ |\ d\ |\ \$$
- I_8 and I_9 are replaced by I_{89}
- I_{89} : $C \rightarrow c\ C\ .\ ,\ c\ |\ d\ |\ \$$

- LALR action and goto functions for condensed sets of items are shown below

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		

- Consider GOTO (I_{36}, C). In original set of LR (1) items, GOTO(I_3, C) = I_8 and I_8 is now part of I_{89}, so we make GOTO(I_{36}, C) be I_{89}.

Using Ambiguous Grammars

- if parsing action conflict, algorithm fails to produce a parser and grammar is said not to be LALR (1).
- Ambiguous grammar can't be handled by all parsers.
- These are quite useful in specification of languages.
- Consider:

  <Exp> → <Exp> + <Exp> / <Exp> * <Exp>

  <Exp> → x
- The above grammar is ambiguous since precedence and association of operation has not been specified.

- But this grammar can be made unambiguous as follows:

  <Exp> → <Exp> + <Term> / <Term>

  <Term> → <Term> * <Fact> / <Fact>

  <Fact> → x

- Now grammar is suitable for LR parsing technique but here we introduce two unit production

  <Exp> → <Term>

  <Term> → <Fact>

- These unit productions make parsing time excessive. So we cannot always adopt this technique.

- If we go for ambiguous grammar then parsing time is not excessive.
- But there will be conflicts in LR parsing.
- These conflicts can be resolved by using precedence and association of + and * as per specification of language.
- **Example**:
- Consider following ambiguous grammar

$E \rightarrow E + E \qquad E \rightarrow E * E \qquad E \rightarrow x$

- Its augmented grammar is

$S \rightarrow E \qquad E \rightarrow E + E \qquad E \rightarrow E * E \qquad E \rightarrow x$

- Let C is canonical collection of LR (0) items.

- I_0: $S \rightarrow .E$
  $E \rightarrow .E + E$
  $E \rightarrow .E * E$
  $E \rightarrow .x$

- I_1: $S \rightarrow E.$
  $E \rightarrow E. + E$
  $E \rightarrow E. * E$

- I_2: $E \rightarrow x.$

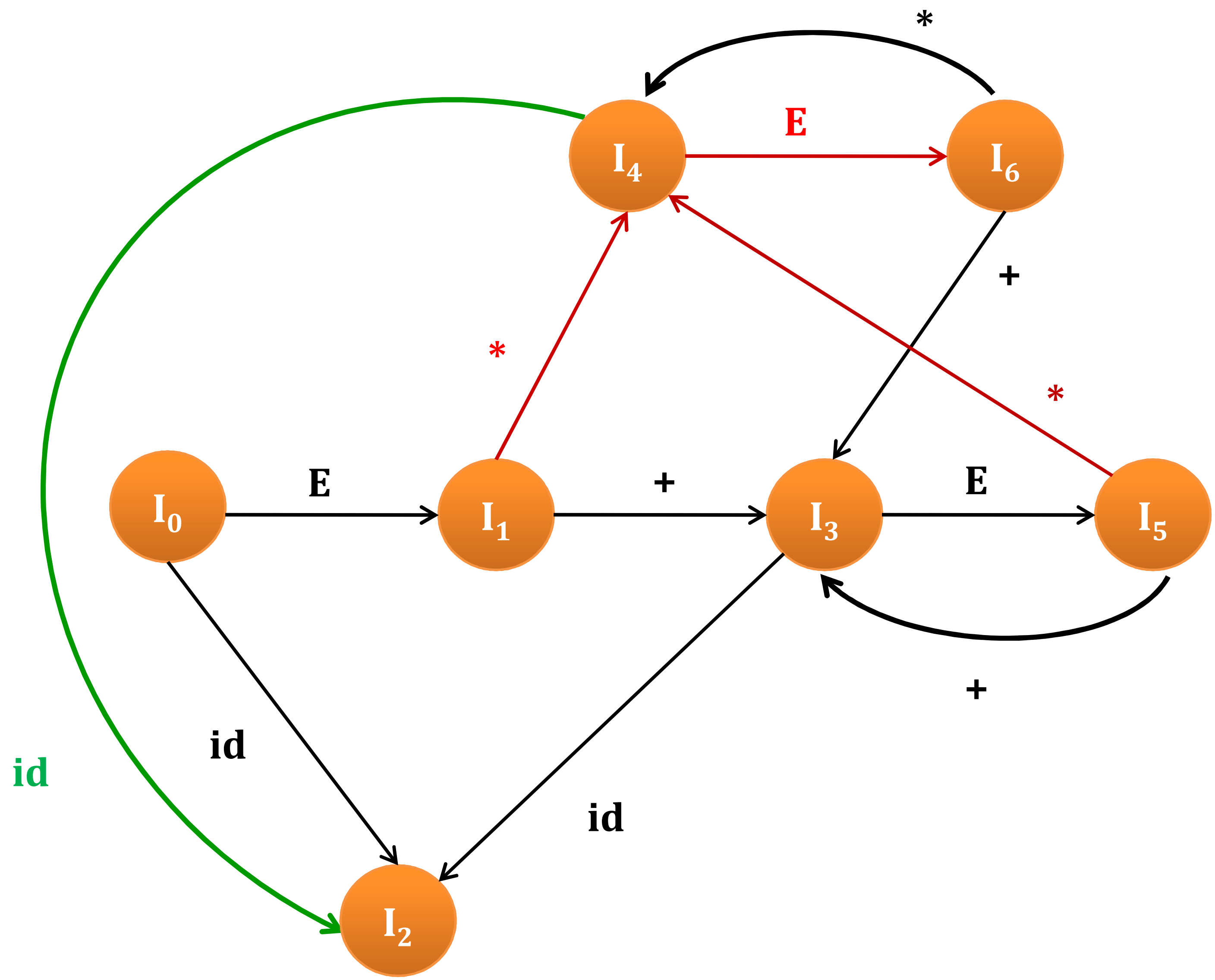
- I_3: $E \rightarrow E + .E$
  $E \rightarrow .E + E$
  $E \rightarrow .E * E$
  $E \rightarrow .x$

- I_4: $E \rightarrow E * .E$
  $E \rightarrow .E + E$
  $E \rightarrow .E * E$
  $E \rightarrow .x$

- I_5: $E \rightarrow E + E.$
  $E \rightarrow E. + E$
  $E \rightarrow E. * E$

- I_6: $E \rightarrow E * E.$
  $E \rightarrow E. + E$
  $E \rightarrow E. * E$

Fig: Transition Diagram



❖ SLR parsing table is as follows:

STATE	ACTION TABLE				GOTO TABLE
	+	*	id	$	E
I_0			S_2		1
I_1	S_3	S_4		acc	
I_2	R_3	R_3		R_3	
I_3			S_2		5
I_4			S_2		6
I_5	S_3 / R_1	S_4 / R_1		R_1	
I_6	S_3 / R_2	S_4 / R_2		R_2	

- It is clear from table that

  action [I5, +] = S3 / R1 (shift reduce conflict)

  action [I5, *] = S4 / R1 (shift reduce conflict)

  action [I6, +] = S3 / R2 (shift reduce conflict)

  action [I6, *] = S4 / R2 (shift reduce conflict)

- These conflicts can be resolved by defining associativity and precedence relations.

Precedence of Terminals and Productions

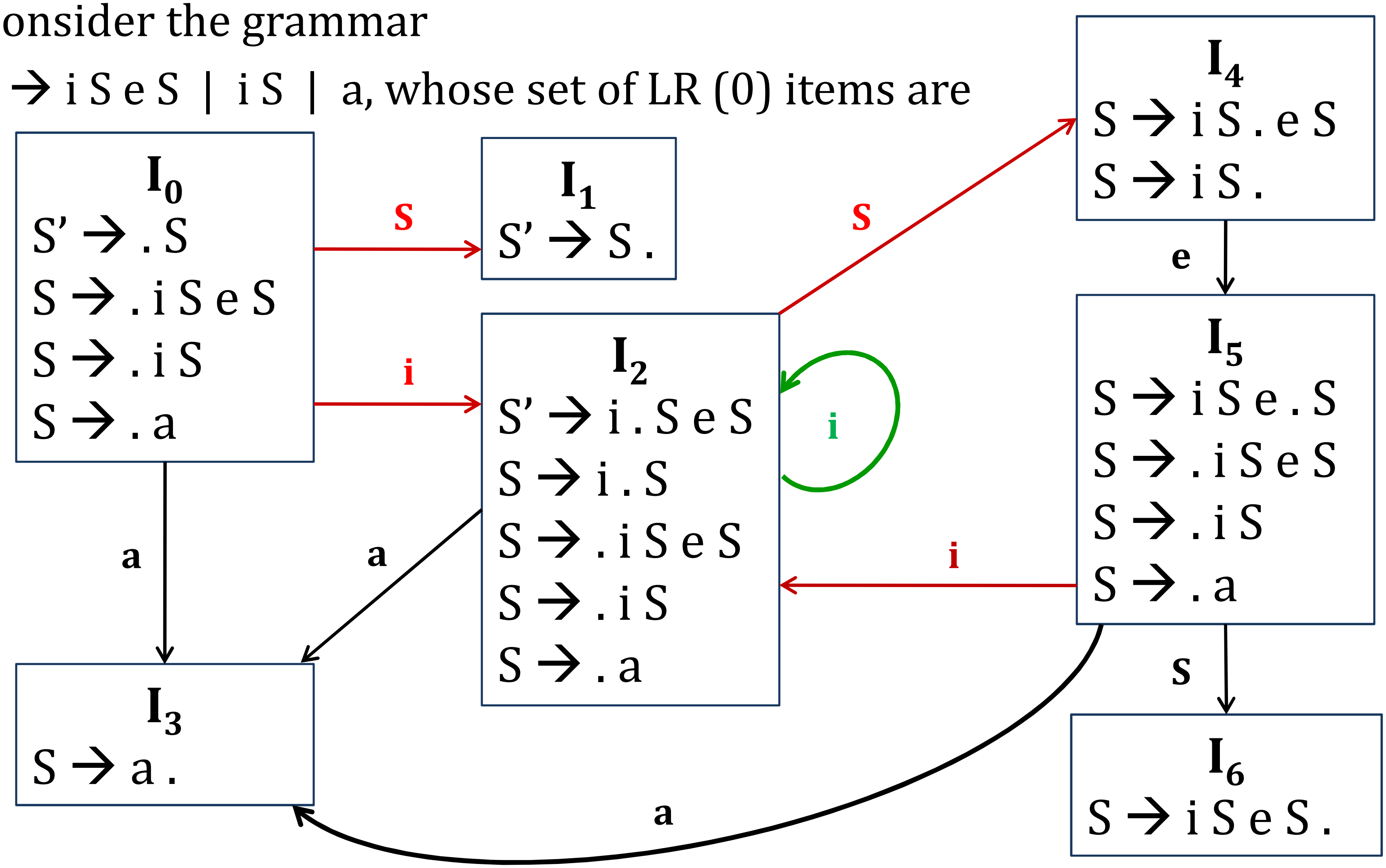
- Idea behind conflict resolution in YACC is that each production and each terminal symbol may be given a "precedence".
- If on input a we have conflict between reducing by production $A \rightarrow \alpha$ and shifting, we compare precedence of $A \rightarrow \alpha$ with precedence of a.
- If $A \rightarrow \alpha$ has higher precedence that a, we reduce; if not we shift.
- YACC has facility that allows user to assign a precedence to every production and to every terminal.

- A more useful way to give precedence to productions is to follow the rule that, in absence of specific precedence for the production, the precedence of $A \rightarrow \alpha$ is same as precedence of rightmost terminal of α.
- Not every terminal and production need be given a precedence; those not involved in conflicts need not have precedence.

Example

Consider the grammar

$S \rightarrow i\,S\,e\,S \mid i\,S \mid a$, whose set of LR (0) items are



- If we simply state that e is of higher precedence than i, then the production S → i S has lower precedence than e
- Since i is the rightmost terminal of right side i S.
- So, in I_4, the conflict between shifting e and reducing by S → i S on input e is resolved.
- We could also specify to YACC directly that precedence of production S → i S is lower than precedence of e by creating dummy terminal of lower precedence than e, by following YACC like notation:

TERMINAL e

TERMINAL dummy

/* terminals with precedence are listed highest precedence first */

S → i S e S

/* then come the productions */

S → i S PRECEDENCE dummy

/* the keyword PRECEDENCE gives the production the precedence of "terminal" dummy */

S → a

Associativity

- Consider ambiguous grammar

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

- We would reduce by $E \rightarrow E * E$ on input +, since given production $E \rightarrow E * E$, with rightmost terminal *, is having higher precedence than terminal +.

Parser Generator

- It can be used to facilitate the construction of front end of a compiler.
- One of the parser generator YACC (Yet Another Compiler-Compiler) reflecting the popularity of parser generators.
- YACC is available as a command on the UNIX system, & has been used to implement hundreds of compilers.
- Parser Generator YACC
- A translator can be constructed using YACC in the following steps:

Automatic Parser Generators

YACC (Yet Another Compiler - Compiler)

- YACC allows user to specify a possibly ambiguous grammar along with precedence and associativity information about operators.
- YACC resolves any parsing action conflicts arise.
- User provides YACC with a grammar, and YACC builds LALR(1) states.
- YACC then attempts to select parsing actions for each state.
- If there are no conflicts (grammar is LALR (1)) then user need not supply anything more than grammar.
- If source grammar is ambiguous, user may provide more information to help YACC resolve parsing action conflicts.

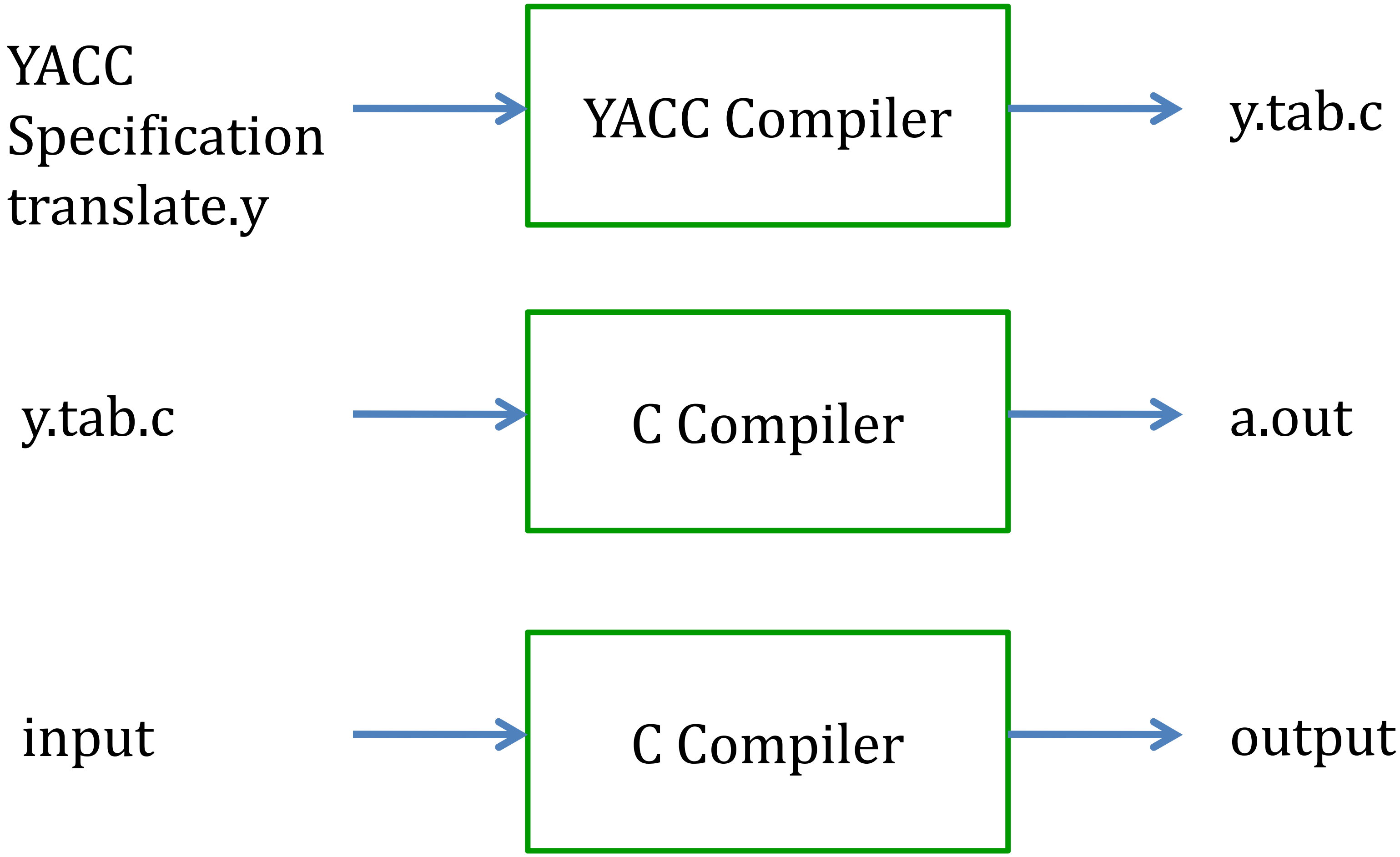


Fig: Input – output Translator with YACC

- First, a file name, say translate.y, containing a Yacc specification of the translator us prepared.
- Then we have to compile the file by using the UNIX system command.
- Yacc translate.y ↲
- It transforms the file translate.y into equivalent C program called y.tab.c
- The program y.tab.c is a representation if some parser e.g. LALR parser written in C.
- By compiling y.tab.c along with by library that contains the LR parsing program using the command un UNIX system.

- cc y.tab.c - ly
- By doing this desired object program a.out that performs the translation specified by the original YACC program.
- YACC source program has three parts:

  Declaration

  %%

  Translation rules

  %%

  Supporting c-routines

Error Recovery in YACC:

- YACC has some provision for error recovery, by using error token.
- Essentially, the error token is used to find a synchronization point in the grammar from which it is likely that processing can continue.
- Sometimes our attempts at recovery will not remove enough of the erroneous state to continue, and the error message will cascade.
- Either the parser will reach a point from which processing can continue or the entire parser will abort.

- After reporting a syntax error, a YACC parser discards any partially parsed rules until it finds one in which it can shift an error token.
- It then reads and discards input tokens until it finds one which can follow the error token in the grammar.
- This later process is called resynchronizing.