LR Parsers

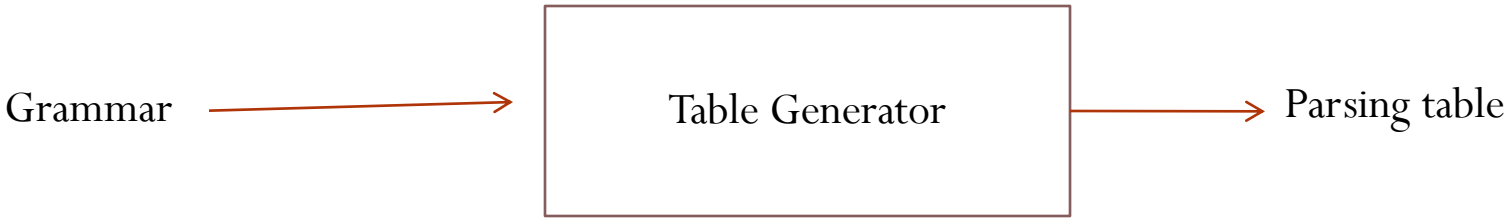
Introduction

- Efficient Bottom-up parsers for a large class of CFG
- LR parsers- scan the input from left to right and construct a rightmost derivation in reverse
- Advantage
  - can be constructed to recognize virtually all programming language constructs for which context-free grammars can be written
  - more general and efficient compared to shift reduce and operator precedence parsing
  - can detect syntax errors as soon as possible to do so on a left to right scan of the input

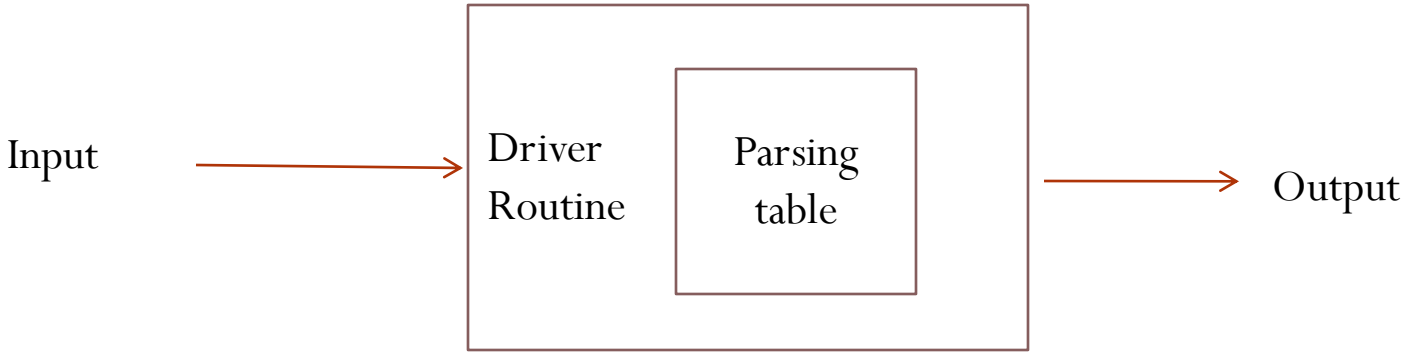
- Principle drawback
  - It is tedious to implement by hand
  - Requires a special tool
- LR parser generator
- logically an LR parser consists of two parts
  1. a driver routine (same for all LR parsers)
  2. parsing table (changes from parser to another)
     - there are many different parsing tables that can be used in an LR parser for a given grammar
     - some may detect errors sooner than others but they all accept the same sentences, exactly the sentences generated by the grammar

Generating an LR Parser

Generating the Parser



Operation of the Parser



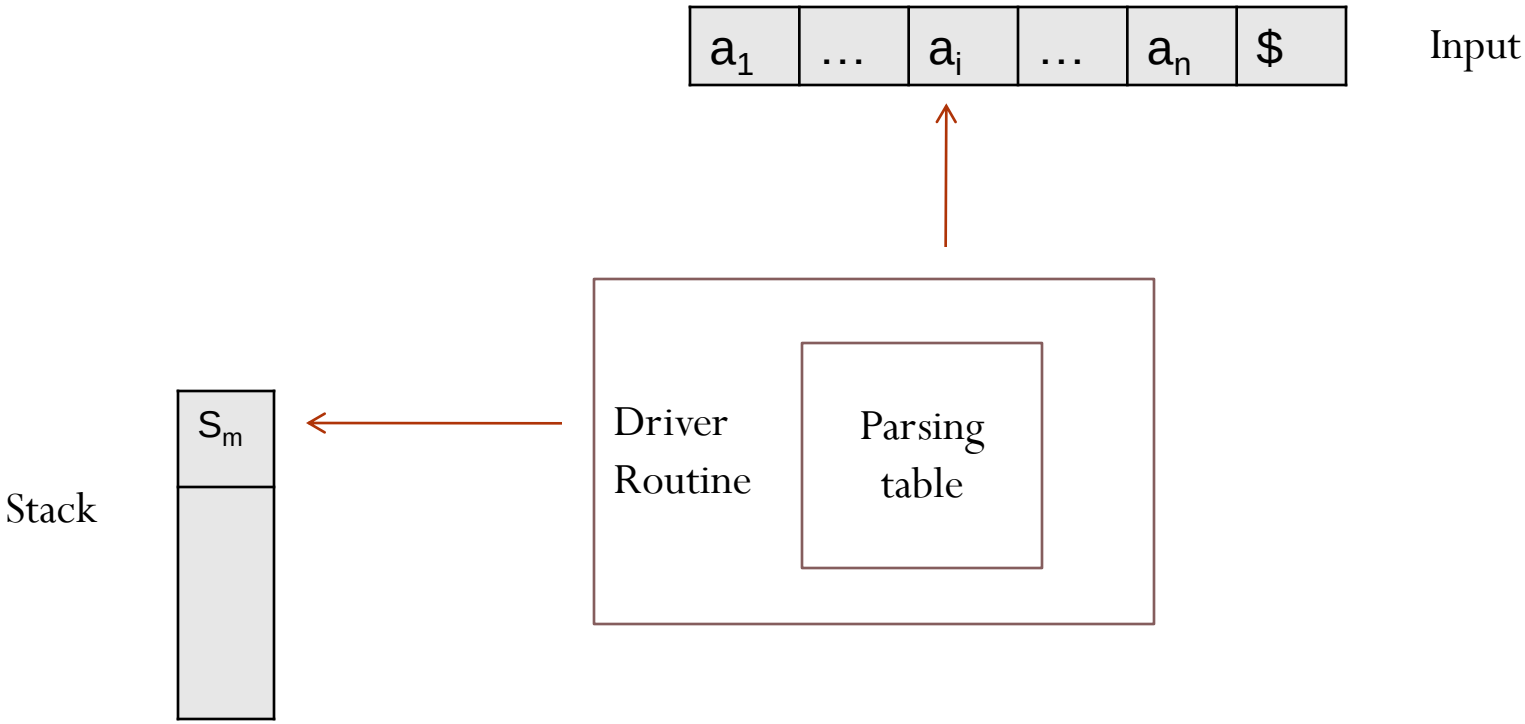
Techniques for producing LR parsing table

- Three different techniques for producing LR parsing tables
  1. simple LR (SLR for short)
  2. Canonical LR - powerful but expensive
  3. lookahead LR (LALR) - is intermediate in power between the SLR and canonical LR

LR Parsers

- The parser has an *input*, a *stack* and a *parsing table*
- The stack contains
  - a string of the form $s_0X_1s_1X_2\ldots X_ms_m$ where s_m is on top
  - 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 the stack below it and it is used to guide the shift reduce decision

LR Parser



LR Parser

- The parsing table consists of two parts
  - a parsing action function ACTION
  - and a goto function GOTO
- The driver behaves as follows
  - It determines s_m the state on top of the stack and a_i the current input symbol
  - It then consults the ACTION[s_m,a_i] which can have 4 values
  - shift s
  - reduce $A \rightarrow \beta$
  - accept
  - error

LR Parsers cont...

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

LR Parsers

- The configuration of an LR parser is a pair whose first component is the stack content and second component is the unexpended input
- Configuration of an LR parser (stack contents + input)
  - (s_0 X_1 s_1 X_2 s_2…X_m $\mathbf{s_m}$, $\mathbf{a_i}$ a_{i+1}…a_n\$)
  - The next move of the parsers is determined by reading a_i the current input symbol and s_m the state on top of the stack and then consulting the parsing action table entry ACTION[s_m,a_i]

Four types of move

1. If ACTION[s_m,a_i]=shift s then
   - (s_0 X_1 s_1 X_2 s_2…X_m s_m a_i s, $\mathbf{a_{i+1}}$…a_n\$)
   - here s=GOTO[s_m,a_i]
2. If ACTION[s_m,a_i]=reduce A→β
   - (s_0 X_1 s_1 X_2 s_2…X_{m-r} s_{m-r} A s, $\mathbf{a_i}$ a_{i+1}…a_n\$)
   - here s=GOTO[s_{m-r},A] and r is the length of β
3. If ACTION[s_m,a_i]=accept parsing is completed
4. If ACTION[s_m,a_i]=error

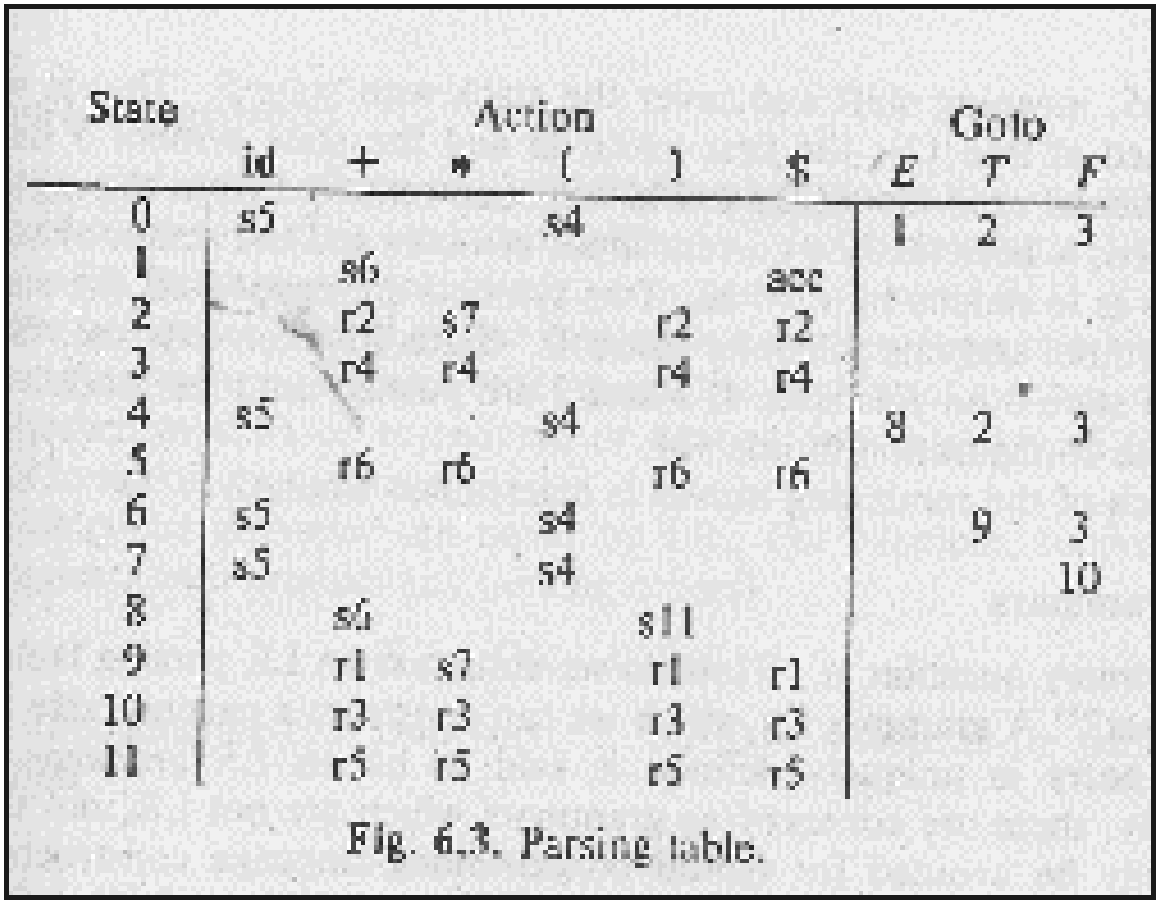
- **Initial configuration is**
- **(s_0, a1 a2…a_n\$) where s_0 is the designated initial state**

example

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

- The codes for the actions are:
  - **si** means shift and stack state i
  - **rj** means reduce by production numbered j
  - acc means accept
  - blank means error

Parsing table



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. 6.3. Parsing table.

Moves of LR parsers

LR grammars

- A grammar for which we can construct a parsing table where every entry is uniquely defined is said to be an *LR grammar*
  - Intuitively in order for a 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
  - does not have to scan the entire stack to know when the handle appears on top of the stack
  - rather the **state symbol** on top of the stack contains all the information it needs
  - *the driver routine of an LR parser is essentially such a finite automaton*
    - it need not read the stack at every move
    - state symbol on top of the stack is enough

LR grammar cont…

- Another information that an LR parser can use to help make its **shift-reduce** decision is the next *k* input symbol
- k=0 and 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 an *LR(k)* grammar

Non-LR Constructs

- A grammar is non LR(1) if every shift-reduce parser for that grammar can reach a configuration in which the parser knowing the entire stack content and the next input symbol cannot decide whether to *shift* or *reduce* or *which reduction* to choose

Canonical Collection of LR(0) Items

- constructing a "simple" LR parser for a grammar
  - The central idea is the construction of a DFA from the grammar
  - Then convert this DFA into an LR parsing table
- the DFA recognizes *viable prefixes* of the grammar
  - Viable Prefix - prefixes of right-sentential forms that do not contain any symbols to the right of the handle
  - -always possible to add terminal symbols to the end of a viable prefix t obtain a right sentential form

Canonical Collection of LR(0) Items

- We define an LR(0) item of a grammar G to be a production of G with a dot at some position of the RHS
- A→XYZ generates the four items
  1. A →.XYZ *this first items indicate we expect to see string derivable from XYZ*
  2. A →X.YZ *this second items indicate we expect to see string derivable from YZ*
  3. A →XY.Z
  4. A →XYZ .
- The production A →∈ generates only one item
- A →.

Grouping of items

- Grouping these items into sets gives rise to **states** of an LR parser
- One such collections of sets of items is called the canonical LR(0) collection provides the basis for constructing SLR
- To construct Canonical Collection of LR(0) Items we need
  1. augmented grammar and
  2. Two functions, CLOSURE and GOTO

Canonical Collection of LR(0) Items

- If G is a grammar with start symbol S, then the Augmented grammar for G is G` with a new start symbol S` and production S`$\rightarrow S$
  - The purpose of this new starting production is to indicate to the parser when it should stop parsing and announce acceptance of the input

Canonical Collection of LR(0) Items

- CLOSURE
  - If *I* is the sets of items for a grammar G, then the sets of items CLOSURE(*I*) is constructed from I
    1. Every item in I is in CLOSURE(*I*)
    2. If $A \rightarrow \alpha.B\beta$ is in CLOSURE(*I*) *and* $B \rightarrow \gamma$ is a production then add item $B \rightarrow .\gamma$ *to I* if its is not there

example

- Consider the augmented grammar
- E→E+T|E →T
- T →T*F|T→F
- F →(E)|F →id
- If I is the set of one item { [E` →.E]}, then CLIOSURE(I) contains the items
- E` →.E
- E→.E+T
- E →.T
- T →.T*F
- T→.F
- F →.(E)
- F →.id

Canonical Collection of LR(0) Items cont...

- GOTO
  - GOTO(I,X) where I is a set of items and X is a grammar symbol
  - GOTO(I,X) is a closure of the set of items
  - [A→αX.β] such that [A→α.Xβ] is in I
  - Example:
  - If I is the set of items {[E→E.],[E →E.+T]}, then GOTO(I,+) consists of

    E →E+.T

    E →.T * F

    T →.F

    F →.(E)

    F →.id

Sets-of-Items Construction

procedure ITEMS(G`)

begin

C={CLOSURE({S`→ .S})};

repeat

for each set of items *I* in C and each grammar symbol X such that GOTO(I,X) is not empty and is not in C

do add GOTO(I,X) to C

until no more sets of items can be added to C

end

Collection of sets of items

I_0: $E' \to \cdot E$
$E \to \cdot E+T$
$E \to \cdot T$
$T \to \cdot T*F$
$T \to \cdot F$
$F \to \cdot(E)$
$F \to \cdot \mathbf{id}$

I_1: $E' \to E\cdot$
$E \to E\cdot+T$

I_2: $E \to T\cdot$
$T \to T\cdot*F$

I_3: $T \to F\cdot$

I_4: $F \to (\cdot E)$
$E \to \cdot E+T$
$E \to \cdot T$
$T \to \cdot T*F$
$T \to \cdot F$
$F \to \cdot(E)$
$F \to \cdot \mathbf{id}$

I_5: $F \to \mathbf{id}\cdot$

I_6: $E \to E+\cdot T$
$T \to \cdot T*F$
$T \to \cdot F$
$F \to \cdot(E)$
$F \to \cdot \mathbf{id}$

I_7: $T \to T*\cdot F$
$F \to \cdot(E)$
$F \to \cdot \mathbf{id}$

I_8: $F \to (E\cdot)$
$E \to E\cdot+T$

I_9: $E \to E+T\cdot$
$T \to T\cdot*F$

I_{10}: $T \to T*F\cdot$

I_{11}: $F \to (E)\cdot$

Fig. 6.7. Collection of sets of items.

Deterministic Finite Automaton

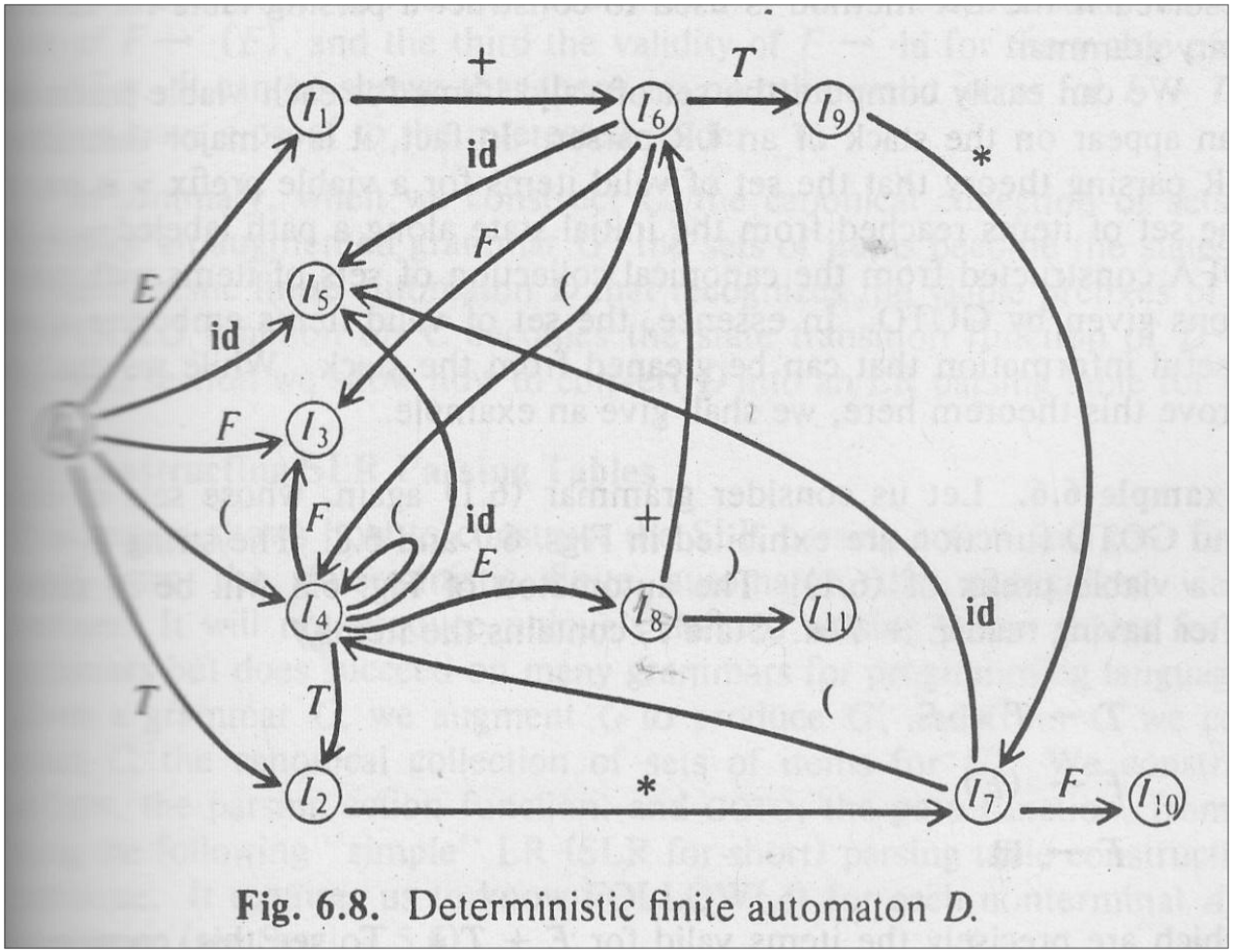


Fig. 6.8. Deterministic finite automaton *D*.

Canonical Collection of LR(0) Items cont...

- Finite Automata of Items
  - The LR(0) of items can be used as the **states** of a finite automaton that maintains information about the *parsing stack* and the progress of a *shift-reduce parse*
  - Start as an NFA and then convert to DFA

Constructing SLR Parsing table

Algorithm 6.1. Construction of an SLR parsing table.

Input. C, the canonical collection of sets of items for an augmented grammar G'.

Output. If possible, an LR parsing table consisting of a parsing action function ACTION and a goto function GOTO.

Method. Let $C = \{I_0, I_1, \ldots, I_n\}$. The states of the parser are $0, 1, \ldots, n$, state i being constructed from I_i. The parsing actions for state i are determined as follows:

1. If $[A \to \alpha \cdot a\beta]$ is in I_i and $\text{GOTO}(I_i, a) = I_j$, then set ACTION$[i, a]$ to "shift j." Here a is a terminal.
2. If $[A \to \alpha \cdot]$ is in I_i, then set ACTION$[i, a]$ to "reduce $A \to \alpha$" for all a in FOLLOW(A).
3. If $[S' \to S \cdot]$ is in I_i, then set ACTION$[i, \$]$ to "accept."

If any conflicting actions are generated by the above rules, we say the grammar is not SLR(1).† The algorithm fails to produce a valid parser in this case.

Constructing SLR Parsing table

The goto transitions for state i are constructed using the rule:

4. If GOTO(I_i, A) = I_j, then GOTO[i, A] = j.
5. All entries not defined by rules (1) through (4) are made "error."
6. The initial state of the parser is the one constructed from the set of items containing $[S' \rightarrow \cdot S]$. □

- Example
- S→AA
- A →aA |b