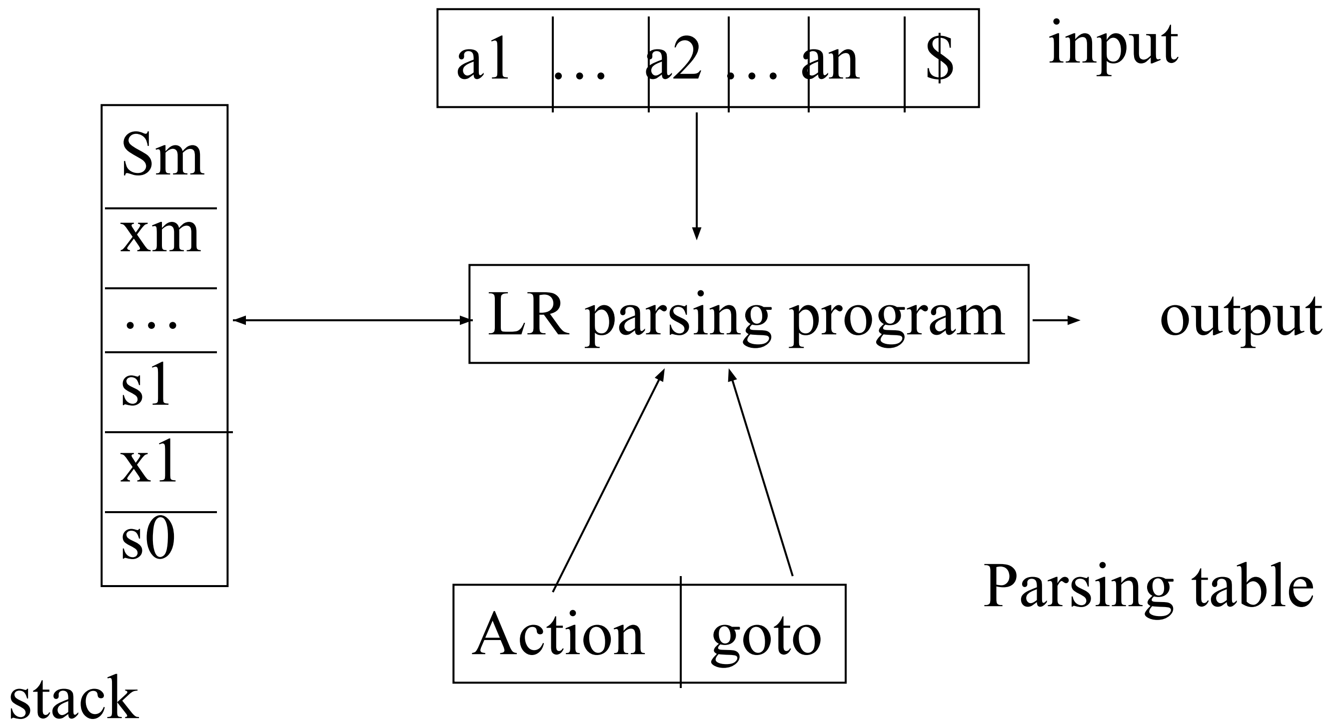
- LR(k) parsers
  - An efficient, bottom-up syntax analysis technique.
  - L -- left to right scanning of the input
  - R -- constructing a rightmost derivation in reverse
  - k -- lookahead for k tokens.
  - Advantages of LR parsing:
    - LR parsers can recognize *almost* all programming language constructs expressed in context-free grammars.
    - LR parsers are efficient and require no backtracking
    - The class of grammars that can be parsed using LR parsing is a superset of the class of grammars that can be parsed with predictive parsers
    - LR parsers can detect a syntactic error as soon as possible on a left to right scan of the input.

Model of an LR parser:



si is a state, xi is a grammar symbol

All LR parsers use the same algorithm, different grammars have different parsing table.

- LR parsing algorithm:
  - input string
  - use a stack to store a string of the form

$$s_0 X_1 s_1 X_2 s_2 \ldots\ldots X_m s_m, \text{ where } s_i \text{ is a state and } X_i \text{ is a symbol}$$

  - Each state summarizes the information contained in the stack below it.
  - The state at the top of the stack and the current input symbol are used to index the parsing table and decide what to do.

- The parsing table has two parts: action table and goto table.
- The entries in the action table, action[s, t] can have four values:
  - shift s1, where s1 is a state (shift t in the stack and put s1 on top of it)
  - reduce by a production A->b (pop b from the stack and replace it with A)
  - accept
  - error

- Goto table entries goto[s, T] = s1, means from current state s, place a non-terminal symbol T results in state s1.

- There are two kinds of operations in the parsing: shift or reduce: this is why this kind of parser is also called shift-reduce parser.
- A configuration of an LR parser is a pair whose first component is the stack and whose second component is the unexpended input:

$$(s_0 X_1 s_1 \ldots X_m s_{m,} a_i a_{i+1} \ldots a_n \$)$$

$$X_1 X_2 \ldots\ldots X_m a_i a_{a+1} \ldots\ldots a_n \quad \text{should be a right - sentential form}$$

```
Set ip to point to the first symbol of the input string and s0 on the stack
repeat forever begin
    let s be the state on top of the stack and a the symbol pointed to by ip
    if (action[s, a] == shift s') then begin
        push a and s' on top of the stack
        advance ip to the next symbol
    end
    else if (action[s, a] == reduce A->b) then begin
        pop 2*|b| symbols off the stack;
        let s' be the state now on top of the stack
        push A then goto[s', A] on top of the stack
        output the production A->b
    end
    else if (action[s, a] = accept) then return
    else error();
end
```

LR parsing program

(1) E->E+T

(2) E->T

(3) T->T*F

(4) T->F

(5) F->(E)

(6) F->id

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			

- The parsing table has two parts: action table and goto table.
- The entries in the action table, action[s, t] can have four values:
  - shift s1, where s1 is a state.
  - reduce by a production A->b.
  - accept
  - error

- Goto table entries goto[s, T] = s1, means from current state s, place a non-terminal symbol T results in state s1.

- A grammar for which we can construct a parsing table is called an **LR grammar**.
- Constructing Simple **LR (SLR) parsing table**:
  - Let G be a grammar with start symbol S, the *augmented grammar* for G, is G with a new start symbol S' and production S'->S.
  - LR(0) item: An LR(0) item of a grammar G is a production of G with a dot at some position of the right side.
  - Example: A->XYZ has four LR(0) items.

    A->.XYZ, A->X.YZ, A->XY.Z, A->XYZ.

– The Closure operation:

- Let I be a set of items for a grammar G, closure(I) can be calculated as follows:

J=I

repeat

for each item **A->a.Bb** in J and production **B->c**,

add B->.c to J (if not there).

Until no more items can be added to J.

- Example:

E' ->E

E->E+T | T

T->T*F | F

F->(E) | id

I = {E'->.E}, what is closure(I)?

– The Goto Operation:

- Goto(I, X), where I is a set of items and X is a grammar symbol, is defined to be the closure of the set of all items A->aX.b such that A->a.Xb is in I.

  Goto(I, X) = closure(the set of items of the form A->aX.b)
  where A->a.Xb is in I.

- Example:

E'->E	I = {E'->E., E->E. + T}
E->E+T | T	
T->T*F | F	Goto(I, +) = ?
F->(E) | id	

– Construct canonical collection of sets of LR(0) items

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.

- Example:

E'->E

E->E+T | T

T->T*F | F

F->(E) | id

- Let each set of a state I and goto(I, X) be the transition. If I0 is the initial state and all other states are final states, this DFA recognizes all viable prefixes of grammar.

- Constructing the SLR parsing table
  - Compute the canonical LR collection sets of LR(0) items for grammar G, let it be C={I0, I1, …., In}.
  - For terminal a, if A->X.aY in Ii and goto(Ii, a) = Ij, then set action[i, a] to shift j.
  - if A->X. is in Ii and A != S’, for all terminals a in Follow(A), set action[i, a] to reduce A->X.
  - if S’->S. is in Ii, then set action[i, $] = accept.
  - For non-terminal symbol A, if goto(Ii, A) = Ij, then set

    goto(i, A) = j
  - set all other table entries to “error”
  - The initial state is the one holding S’->.S
- Example: 4.36 in page 224.

Canonical LR(0) Collections

Closure(I)

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

goto(I_0, E) I_1: $E' \to E \cdot$
$E \to E \cdot + T$

goto(I_0, T) I_2: $E \to T \cdot$
$T \to T \cdot * F$

goto(I_0, F) I_3: $T \to F \cdot$

goto(I_0, ( ) I_4: $F \to (\cdot E)$
$E \to \cdot E + T$
$E \to \cdot T$
$T \to \cdot T * F$
$T \to \cdot F$
$F \to \cdot (E)$
$F \to \cdot \mathbf{id}$

goto(I_0, Fd) I_5: $F \to \mathbf{id} \cdot$

goto(I_1, id) I_6: $E \to E + \cdot T$
$T \to \cdot T * F$
$T \to \cdot F$
$F \to \cdot (E)$
$F \to \cdot \mathbf{id}$

goto(I_2, *) I_7: $T \to T * \cdot F$
$F \to \cdot (E)$
$F \to \cdot \mathbf{id}$

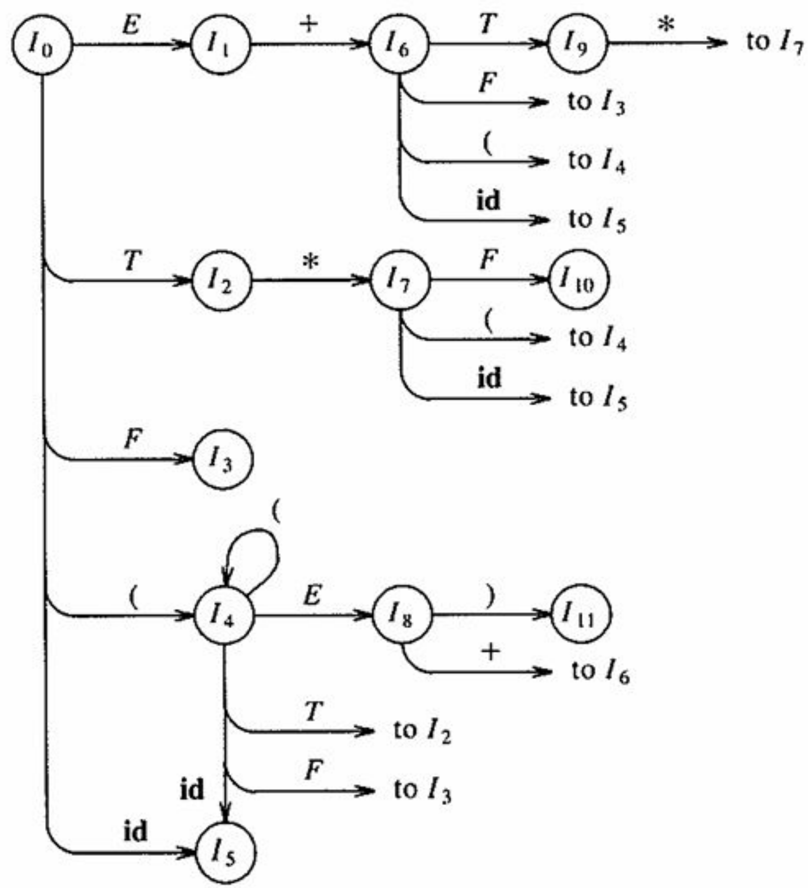
goto(I_4, E) I_8: $F \to (E \cdot)$
$E \to E \cdot + T$

goto(I_6, T) I_9: $E \to E + T \cdot$
$T \to T \cdot * F$

goto(I_7, F) I_{10}: $T \to T * F \cdot$

goto(I_8,)) I_{11}: $F \to (E) \cdot$

Transition Diagram for Prefixes



Construction of SLR Parsing Table

Input. An augmented grammar G'.

Output. The SLR parsing table functions *action* and *goto* for G'.

Method.

1. Construct $C = \{I_0, I_1, \ldots, I_n\}$, the collection of sets of LR(0) items for G'.

2. State i is constructed from I_i. The parsing actions for state i are determined as follows:

   a) If $[A \rightarrow \alpha \cdot a \beta]$ is in I_i and $goto(I_i, a) = I_j$, then set $action[i, a]$ to "shift j." Here a must be a terminal.

   b) If $[A \rightarrow \alpha \cdot]$ is in I_i, then set $action[i, a]$ to "reduce $A \rightarrow \alpha$" for all a in FOLLOW(A); here A may not be S'.

   c) If $[S' \rightarrow S \cdot]$ is in I_i, then set $action[i, \$]$ to "accept."

3. The goto transitions 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 the set of items containing $[S' \rightarrow \cdot S]$. □

(1) E->E+T

(2) E->T

(3) T->T*F

(4) T->F

(5) F->(E)

(6) F->id

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			

SLR Parsing Table

– Exercise: construct the SLR parsing table for grammar:

S->L=R,

S->R

L->*R

L->id

R->L

– The grammar can have shift/reduce conflict or reduce/reduce conflict.

- What about shift/shift conflict
- What about reduce/reduce conflict