Compiler Construction
Bottom-Up Parsing
LR Parsing

LR Parsers

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

LR(k) parsing.

left to right scanning — right-most derivation — k lookhead (k is omitted ➔ it is 1)

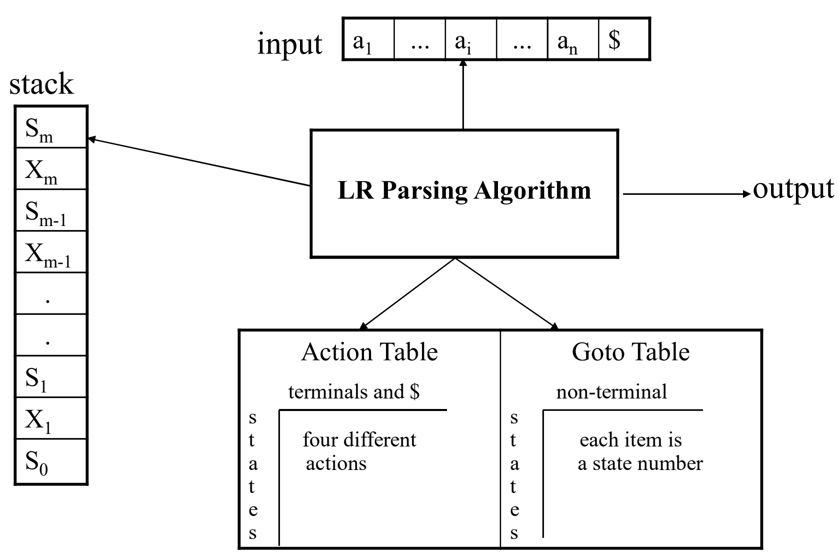
- LR parsing is attractive because:
  - LR parsing is most general non-backtracking shift-reduce parsing, yet it is still efficient.
  - The class of grammars that can be parsed using LR methods is a proper superset of the class of grammars that can be parsed with predictive parsers.

    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 Parsers

- **LR-Parsers**
  - covers wide range of grammars.
  - SLR – simple LR parser
  - CLR – most general LR parser
  - LALR – intermediate LR parser (look-head LR parser)
  - SLR, LR and LALR work same (they used the same algorithm), only their parsing tables are different.

LR Parsing Algorithm



(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

stack	**input**	**action**	**output**
0	id*id+id$	shift 5	
0id5	*id+id$	reduce by F→id	F→id
0F3	*id+id$	reduce by T→F	T→F
0T2	*id+id$	shift 7	
0T2*7	id+id$	shift 5	
0T2*7id5	+id$	reduce by F→id	F→id
0T2*7F10	+id$	reduce by T→T*F	T→T*F
0T2	+id$	reduce by E→T	E→T
0E1	+id$	shift 6	
0E1+6	id$	shift 5	
0E1+6id5	$	reduce by F→id	F→id
0E1+6F3	$	reduce by T→F	T→F
0E1+6T9	$	reduce by E→E+T	E→E+T
0E1	$	accept	

A Configuration of LR Parsing Algorithm

- A configuration of a LR parsing is:

  $(S_o\ X_1\ S_1 \ldots X_m\ S_m,\quad a_i\ a_{i+1} \ldots a_n\ \$)$

  Stack Rest of Input

- S_m and a_i decides the parser action by consulting the parsing action table. (*Initial Stack* contains just S_o)

- A configuration of a LR parsing represents the right sentential form:

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

Actions of A LR-Parser

1. **shift s** -- shifts the next input symbol and the state **s** onto the stack

   $(S_o\ X_1\ S_1 \ldots X_m\ S_m,\quad a_i\ a_{i+1} \ldots a_n\ \$) \rightarrow (S_o\ X_1\ S_1 \ldots X_m\ S_m\ a_i\ s,\quad a_{i+1} \ldots a_n\ \$)$

2. **reduce A→β** (or **rn** where n is a production number)
   - pop 2|β| (=r) items from the stack;
   - then push **A** and **s** where **s=goto[s_{m-r},A]**

   $(S_o\ X_1\ S_1 \ldots X_m\ S_m,\quad a_i\ a_{i+1} \ldots a_n\ \$) \rightarrow (S_o\ X_1\ S_1 \ldots X_{m-r}\ S_{m-r}\ A\ s,\quad a_i \ldots a_n\ \$)$

   - Output is the reducing production reduce A→β

3. **Accept** – Parsing successfully completed

4. **Error** -- Parser detected an error (an empty entry in the action table)

Reduce Action

- pop 2|β| (=r) items from the stack; let us assume that $\beta = Y_1 Y_2 \ldots Y_r$
- then push **A** and **s** where **s=goto[s_{m-r},A]**

$(S_o\ X_1\ S_1 \ldots X_{m-r}\ S_{m-r}\ Y_1\ S_{m-r+1} \ldots Y_r\ S_m,\ a_i\ a_{i+1} \ldots a_n\ \$)$

$\quad \Rightarrow (S_o\ X_1\ S_1 \ldots X_{m-r}\ S_{m-r}\ A\ s,\ a_i \ldots a_n\ \$)$

- In fact, $Y_1 Y_2 \ldots Y_r$ is a handle.

$X_1 \ldots X_{m-r}\ A\ a_i \ldots a_n\ \$ \Rightarrow X_1 \ldots X_m\ Y_1 \ldots Y_r\ a_i\ a_{i+1} \ldots a_n\ \$$

Constructing SLR Parsing Tables – LR(0) Item

- An **LR(0) item** of a grammar G is a production of G a dot at the some position of the right side.
- Ex: $A \rightarrow aBb$ Possible LR(0) Items: $A \rightarrow \bullet aBb$
  (four different possibility) $A \rightarrow a \bullet Bb$
  $A \rightarrow aB \bullet b$
  $A \rightarrow aBb \bullet$
- Sets of LR(0) items will be the states of action and goto table of the SLR parser.
- A collection of sets of LR(0) items (**the canonical LR(0) collection**) is the basis for constructing SLR parsers.
- *Augmented Grammar*:
  G' is G with a new production rule **S'→S** where S' is the new starting symbol.

The Closure Operation

- If I is a set of LR(0) items for a grammar G, then ***closure(I)*** is the set of LR(0) items constructed from I by the two rules:

1. Initially, every LR(0) item in I is added to closure(I).
2. If $\mathbf{A \rightarrow \alpha \bullet B\beta}$ is in closure(I) and $\mathbf{B \rightarrow \gamma}$ is a production rule of G; then $\mathbf{B \rightarrow \bullet\gamma}$ will be in the closure(I). We will apply this rule until no more new LR(0) items can be added to closure(I).

What is happening by $B \rightarrow \bullet\gamma$?

The Closure Operation -- Example

$E' \rightarrow E$ closure($\{E' \rightarrow \bullet E\}$) =

$E \rightarrow E+T$ $\{$ $E' \rightarrow \bullet E$ kernel items

$E \rightarrow T$ $E \rightarrow \bullet E+T$

$T \rightarrow T*F$ $E \rightarrow \bullet T$

$T \rightarrow F$ $T \rightarrow \bullet T*F$

$F \rightarrow (E)$ $T \rightarrow \bullet F$

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

$F \rightarrow \bullet id$ $\}$

Computation of Closure

```
function closure ( I )
begin
        J := I;
        repeat
                for each item A → α.Bβ in J and each production
                        B→γ of G such that B→.γ is not in J do
                                add B→.γ to J
                until no more items can be added to J
end
```

Goto Operation

- If I is a set of LR(0) items and X is a grammar symbol (terminal or non-terminal), then goto(I,X) is defined as follows:
  - If $A \rightarrow \alpha \bullet X\beta$ in I then every item in **closure({$A \rightarrow \alpha X \bullet \beta$})** will be in goto(I,X).
  - If I is the set of items that are valid for some viable prefix γ, then goto(I,X) is the set of items that are valid for the viable prefix γX.

Example:

I ={ $E' \rightarrow \bullet E$, $E \rightarrow \bullet E+T$, $E \rightarrow \bullet T$,
$T \rightarrow \bullet T*F$, $T \rightarrow \bullet F$,
$F \rightarrow \bullet (E)$, $F \rightarrow \bullet id$ }

goto(I,E) = { $E' \rightarrow E\bullet$, $E \rightarrow E\bullet+T$ }
goto(I,T) = { $E \rightarrow T\bullet$, $T \rightarrow T\bullet*F$ }
goto(I,F) = {$T \rightarrow F\bullet$ }

goto(I,() = { $F \rightarrow (\bullet E)$, $E \rightarrow \bullet E+T$, $E \rightarrow \bullet T$, $T \rightarrow \bullet T*F$, $T \rightarrow \bullet F$, $F \rightarrow \bullet (E)$, $F \rightarrow \bullet id$ }
goto(I,id) = { $F \rightarrow id\bullet$ }

Construction of The Canonical LR(0) Collection

- To create the SLR parsing tables for a grammar G, we will create the canonical LR(0) collection of the grammar G'.

- ***Algorithm***:
  C is { closure({S'→ • S}) }
  repeat the followings until no more set of LR(0) items can be added to ***C***.
      for each I in ***C*** and each grammar symbol X
          if goto(I,X) is not empty and not in ***C***
              add goto(I,X) to ***C***

- goto function is a DFA on the sets in C.

The Canonical LR(0) Collection -- Example

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).