Example tables(Last slide of the last lecture)

| | Action | | | | GOTO | | |
|---|---|---|---|---|---|---|---|
| | Id | + | * | $ | <expr> | <term> | <factor> |
| $S_0$ | s4 | – | – | – | 1 | 2 | 3 |
| $S_1$ | – | – | – | acc | – | – | – |
| $S_2$ | – | S5 | – | r3 | – | – | – |
| $S_3$ | – | r5 | s6 | r5 | – | – | – |
| $S_4$ | – | r6 | s6 | r5 | – | – | – |
| $S_5$ | s4 | – | – | – | 7 | 2 | 3 |
| $S_6$ | s4 | – | – | – | – | 8 | 3 |
| $S_7$ | – | – | – | r2 | – | – | – |
| $S_8$ | – | r4 | – | r4 | – | – | – |

The Grammar

| | |
|---|---|
| 1 | <goal> ::= <expr> |
| 1 | <expr> ::= <expr> + <term> |
| 2 | \| <term> |
| 3 | <term> ::= <term> * <factor> |
| 4 | \| <factor> |
| 5 | <factor> ::= id |

## LR Parsing

There are three commonly used algorithms to build tables for an "LR" parser:

1. SLR(1) = LR(0) + FOLLOW
   - smallest class of grammars
   - smallest tables(number of states)
   - simple, fast construction
2. LR(1)
   - full set of LR(1) grammars
   - largest tables(number of states)
   - slow, large construction
3. LALR(1)
   - intermediate sized set of grammars
   - same number of states as SLR(1)
   - canonical construction is slow and large
   - better construction techniques exist

An LR(1) parser for either ALGOL or PASCAL has several thousand states, while an SLR(1) or LALR(1) parser for the same language may have several hundred states.

## Viable prefix

A viable prefix is
1. a prefix of a right-sentential form that does not continue past the right end of the rightmost handle of that sentential form
2. a prefix of a right-sentential form that can appear on the stack of a shift-reduce parser.

If the viable prefix is a proper prefix(that is, a handle), it is possible to add terminals onto its end to form a right-sentential form.

As long as the prefix represented by the stack is viable, the parser has not seen a detectable error.

## SLR(1) parsing

Viable prefix of a right-sentential form:

- contains both terminals and nonterminals
- recognized with NFA or DFA

Building a SLR parser

- begin with NFA for recognizing viable prefixes
- construct DFA for recognizing viable prefixes
- augment with FOLLOW to disambiguate reductions

States in the NFA are LR(0) iterms

States in the DFA are sets of LR(0) items

## LR(0) items

An LR(0) item is a string [$\alpha$], where
$\alpha$ is a production from G with a • at some position in the rhs

The • indicates how much of an item we have seen at a given state in the parse.

[A ::= •XYZ] indicates that the parser is looking for a string that can be derived from XYZ
[A ::= XY•Z] indicates that the parser has seen a string derived from XY and is looking for one derivable from Z

LR(0) Items(no lookahead)
A ::= XYZ generates 4 LR(0) items.

1. [A ::= •XYZ]
2. [A ::= X•YZ]
3. [A ::= XY•Z]
4. [A ::= XYZ•]

## Cannonical LR(0) items

The SLR(1) table construction algorithm uses a specific set of sets of LR(0) items

These sets are called the canonical collection of sets of LR(0) items for a grammar G

The canonical collection represents the set of valid states for the LR parser

The items in each set of the canonical collection fall into two classes:

- kernel items: items where • is not at the left end of the rhs and
  [S’ ::= •S]
- non-kernel items: all items where • is at the left end of rhs

## LR(0) items

Each LR(0) item corresponds to a point in the parse

To generate a parser state from a kernel item, we take its closure

- if $[A ::= \alpha \bullet B\beta] \in I_j$ , then, in state j, the parser might next see a string derivable from $B\beta$
- to form its closure, add all items of the form $[B ::= \bullet \gamma] \in G$

(Note) An augmented grammar is one where the start symbol appears only on the lhs of productions. For the rest of LR parsing, we will assume the grammar is augmented with a production S' ::= S

## Canonical LR(0) items

The canonical collection of LR(0) items:

- set of items derivable from [S' ::= • S]
- set of all items that can derive the final configuration

Essentially,

- each set in the canonical collection of sets of LR(0) items represents a state in an NFA that recognizes viable prefixes.
- Grouping together is really the subset construction

To construct the canonical collection, we need two functions:

- Closure(I)
- GOTO(I,X)

## Closure(I)

Given an item $[A ::= \alpha \bullet B\beta]$, its closure contains the item and any other items that can generate legal substrings to follow $\alpha$.

Thus, if the parser has viable prefix $\alpha$ on its stack, the input should reduce to $B\beta$(or $\gamma$ for some other item $[B ::= \bullet \gamma]$ in the closure).

To compute closure(I)

```
function closure(I)
   repeat
      new_item ← false
      for each item [A ::= α • Bβ] ∈ I,
         each production B ::= γ ∈ G'
         if [B ::= •γ] ∉ I then
            add [B ::= •γ] to I
            new_item ← true
         endif
   until (new_item = false)
   return I
```

## Goto(I, X)

Let I be a set of LR(0) items and X be a grammar symbol.
Then, GOTO(I, X) is the closure of the set of all items

$[A ::= \alpha\ X\bullet\ \beta]$ such that $[A ::= \alpha \bullet X\beta] \in I$

If I is the set of valid items for some viable prefix $\gamma$, then goto(I, X) is the set of valid items for the viable prefix $\gamma X$.
goto(I, X) represents state after recognizing X in state I.

To compute goto(I, X)

```
function goto(I, X)
    J ← set of items [A ::= αX•β]
      such that [A ::= α•Xβ] ∈ I
    J' ← closure(J)
    return J'
```

## Collection of sets of LR(0) items

We start the construction of the collection of sets of LR(0) items with the item [S' ::= • S], where

S' is the start symbol of the augmented grammar G'
S is the start symbol of G

To compute the collection of sets of LR(0) items

```
Procedure items(G')
   C ← closure({[S' ::= • S]})
   repeat
      for each set of items I in C do
         for each grammar symbol X
             if goto (I, X) is not empty and not in C
                add goto (I, X) to C
              endif
   until no new sets of items are added to C
return C
```