CS3300 - Compiler Design

Bottom-up Parsing

KC Sivaramakrishnan

IIT Madras

Some definitions

Recall

- For a grammar G, with start symbol S, any string α such that $S \Rightarrow^* \alpha$ is called a *sentential form*
- If $\alpha \in V_t^*$, then α is called a *sentence* in $L(G)$

A *left-sentential form* is a sentential form that occurs in the leftmost derivation of some sentence.

A *right-sentential form* is a sentential form that occurs in the rightmost derivation of some sentence.

An unambiguous grammar will have a unique leftmost/rightmost derivation.

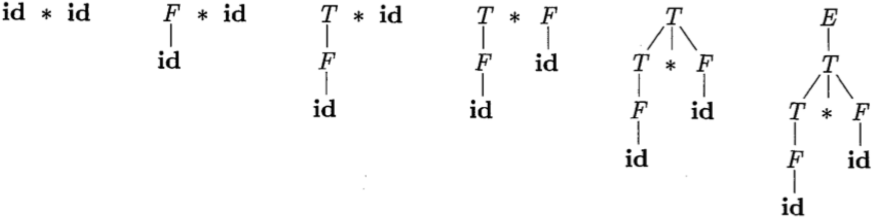
Bottom-up parsing

Consider:

$$\begin{array}{lcl} E & \rightarrow & E+T \mid E-T \mid T \\ T & \rightarrow & T*F \mid T/F \mid F \\ F & \rightarrow & \texttt{num} \mid \texttt{id} \end{array}$$

Goal:

Given an input string w and a grammar G, construct a parse tree by starting at the leaves and working to the root.



Reductions

Reduction:

- At each reduction step, a specific substring matching the body of a production is replaced by the non-terminal at the head of the production.

Key decisions

- When to reduce?
- What production rule to apply?

Reductions VS Derivations

- Recall: In derivation: a non-terminal in a sentential form is replaced by the body of one of its productions.
- A reduction is reverse of a step in derivation.

- Bottom-up parsing is the process of “reducing” a string w to the start symbol.
- Goal of bottom-up parsing: build derivation tree in reverse.

Example

Consider the grammar

$$\begin{array}{c|lcl} 1 & S & \rightarrow & \mathtt{a}AB\mathtt{e} \\ 2 & A & \rightarrow & A\mathtt{bc} \\ 3 & & | & \mathtt{b} \\ 4 & B & \rightarrow & \mathtt{d} \end{array}$$

and the input string `abbcde`

The Reduction:

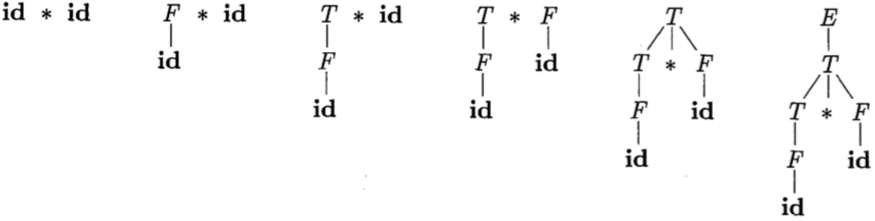
Prod'n.	Sentential Form
3	$\mathtt{a}\boxed{\mathtt{b}}\mathtt{bcde}$
2	$\mathtt{a}\boxed{A\mathtt{bc}}\mathtt{de}$
4	$\mathtt{a}A\boxed{\mathtt{d}}\mathtt{e}$
1	$\boxed{\mathrm{a}AB\mathrm{e}}$
–	S

Rightmost Derivation:

$$\begin{array}{rcl} S & \Rightarrow & \mathtt{a}AB\mathtt{e} \\ & \Rightarrow & \mathtt{a}A\mathtt{de} \\ & \Rightarrow & \mathtt{a}A\mathtt{bcde} \\ & \Rightarrow & \mathtt{abbcde} \end{array}$$

Notice that the reduction is actually reverse of the rightmost derivation.

Another Example



$$E \Rightarrow T \Rightarrow T * F \Rightarrow T * \texttt{id} \Rightarrow F * \texttt{id} \Rightarrow \texttt{id} * \texttt{id}$$

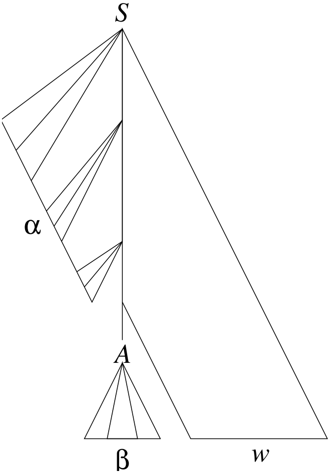
Bottom-up Parsing and Rightmost Derivations

A bottom-up parser traces a rightmost derivation in reverse.

Consequence of this fact:

- Suppose $\alpha\beta\omega$ is a step of a bottom-up parse.
- Assume that the next reduction is by $X \rightarrow \beta$
- Then, what can we say about ω?
  - ω must consist of only terminal symbols.
  - $\alpha X\omega \Rightarrow \alpha\beta\omega$ is a step in a rightmost derivation.

Handles



The handle $A \rightarrow \beta$ in the parse tree for $\alpha\beta w$

Informally, a “handle” is

- a substring that matches the body of a production (not necessarily the first such substring),
- and reducing this handle, represents one step of reduction (or reverse rightmost derivation).

Handle-pruning

The process to construct a bottom-up parse is called *handle-pruning*. To construct a rightmost derivation in reverse

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

we apply the following simple algorithm

```
for i = $n$ downto 1
  1 find the handle $A_i \to \beta_i$ in $\gamma_i$
  2 replace $\beta_i$ with $A_i$ to generate $\gamma_{i-1}$
```

How to find handles?

- We know that all symbols to the right of a handle must be terminal symbols.
- Idea: Split the string into two substrings
  - Right substring is as yet unexamined by parsing (a string of terminals)
  - Left substring has terminals and non-terminals
- The dividing point is marked by a $|$
  - $|$ is not part of the string
- Initially, all input is unexamined $| x_1 x_2 \ldots x_n$.

Bottom-up Parsing

Bottom-up parsing uses only two kinds of actions:

- *Shift:* Move $|$ one place to the right.
  - That is, shift a terminal to the left substring.
  - $\alpha \mid aw \rightsquigarrow \alpha a \mid w$
- *Reduce:* Apply an inverse production to the right end of the left sub-string.
  - If $A \rightarrow \gamma$ is a production, then $\alpha\gamma \mid w \rightsquigarrow \alpha A \mid w$

Bottom-up parsing is also called Shift-Reduce Parsing.

Shift-Reduce Parsing: Example 1

$\mid \texttt{id} \star \texttt{id}$	
$\texttt{id} \mid \star\ \texttt{id}$	Shift
$F \mid \star\ \texttt{id}$	Reduce by $F \rightarrow \texttt{id}$
$T \mid \star\ \texttt{id}$	Reduce by $T \rightarrow F$
$T \star \texttt{id} \mid$	Shift
$T \star F \mid$	Reduce by $F \rightarrow \texttt{id}$
$T \mid$	Reduce by $T \rightarrow T \star F$
$E \mid$	Reduce by $E \rightarrow T$

Shift-Reduce Parsing: Example 2

1	S	$\rightarrow$	a$AB$e
2	A	$\rightarrow$	Abc
3		$\mid$	b
4	B	$\rightarrow$	d

$\mid$ abbcde	
ab $\mid$ bcde	Shift
a A $\mid$ bcde	Reduce by $A \rightarrow$ b
a A bc $\mid$ de	Shift
a A $\mid$ de	Reduce by $A \rightarrow A$bc
a A d $\mid$ e	Shift
a AB $\mid$ e	Reduce by $B \rightarrow$ d
a AB e $\mid$	Shift
S $\mid$	Redece by $S \rightarrow$ a$AB$e

Stack implementation

We can implement the division into left and right sub-strings using a *stack*.

- Top of the stack will be the marker $|$ (implicitly).
- Shift-reduce parsers use a *stack* and an *input buffer*

1. initialize stack with $
2. Repeat until the top of the stack is the goal symbol and the input token is $
   a) *find the handle*
      if we don't have a handle on top of the stack, *shift* an input symbol onto the stack
   b) *prune the handle*
      if we have a handle $A \rightarrow \beta$ on the top of the stack, *reduce*
      i) pop $|\beta|$ symbols off the stack
      ii) push A onto the stack

Example: Parsing `x − 2 * y`

1 $S \to E$
2 $E \to E + T$
3 $\quad\mid\ E - T$
4 $\quad\mid\ T$
5 $T \to T * F$
6 $\quad\mid\ T / F$
7 $\quad\mid\ F$
8 $F \to \langle\text{num}\rangle$
9 $\quad\mid\ \langle\text{id}\rangle$

Stack	Input	Action
\$	$\langle\text{id}\rangle - \langle\text{num}\rangle * \langle\text{id}\rangle$	S
$\$\underline{\langle\text{id}\rangle}$	$- \langle\text{num}\rangle * \langle\text{id}\rangle$	R9
$\$\underline{\overline{\langle\text{factor}\rangle}}$	$- \langle\text{num}\rangle * \langle\text{id}\rangle$	R7
$\$\underline{\overline{\langle\text{term}\rangle}}$	$- \langle\text{num}\rangle * \langle\text{id}\rangle$	R4
$\$\overline{\langle\text{expr}\rangle}$	$- \langle\text{num}\rangle * \langle\text{id}\rangle$	S
$\$\langle\text{expr}\rangle -$	$\langle\text{num}\rangle * \langle\text{id}\rangle$	S
$\$\langle\text{expr}\rangle - \underline{\langle\text{num}\rangle}$	$* \langle\text{id}\rangle$	R8
$\$\langle\text{expr}\rangle - \underline{\overline{\langle\text{factor}\rangle}}$	$* \langle\text{id}\rangle$	R7
$\$\langle\text{expr}\rangle - \overline{\langle\text{term}\rangle}$	$* \langle\text{id}\rangle$	S
$\$\langle\text{expr}\rangle - \langle\text{term}\rangle *$	$\langle\text{id}\rangle$	S
$\$\langle\text{expr}\rangle - \langle\text{term}\rangle * \underline{\langle\text{id}\rangle}$		R9
$\$\langle\text{expr}\rangle - \underline{\langle\text{term}\rangle * \overline{\langle\text{factor}\rangle}}$		R5
$\$\underline{\langle\text{expr}\rangle - \overline{\langle\text{term}\rangle}}$		R3
$\$\overline{\langle\text{expr}\rangle}$		R1
$\$\overline{\langle\text{goal}\rangle}$		A

Example: Rightmost derivation of `x − 2 * y`

The left-recursive expression grammar

1	$\langle\text{goal}\rangle$	$::=$	$\langle\text{expr}\rangle$
2	$\langle\text{expr}\rangle$	$::=$	$\langle\text{expr}\rangle + \langle\text{term}\rangle$
3		$\mid$	$\langle\text{expr}\rangle - \langle\text{term}\rangle$
4		$\mid$	$\langle\text{term}\rangle$
5	$\langle\text{term}\rangle$	$::=$	$\langle\text{term}\rangle * \langle\text{factor}\rangle$
6		$\mid$	$\langle\text{term}\rangle / \langle\text{factor}\rangle$
7		$\mid$	$\langle\text{factor}\rangle$
8	$\langle\text{factor}\rangle$	$::=$	$\langle\text{num}\rangle$
9		$\mid$	$\langle\text{id}\rangle$

Prod'n.	Sentential Form
–	$\langle\text{goal}\rangle$
1	$\underline{\langle\text{expr}\rangle}$
3	$\underline{\langle\text{expr}\rangle - \langle\text{term}\rangle}$
5	$\langle\text{expr}\rangle - \underline{\langle\text{term}\rangle * \langle\text{factor}\rangle}$
9	$\langle\text{expr}\rangle - \langle\text{term}\rangle * \underline{\langle\text{id}\rangle}$
7	$\langle\text{expr}\rangle - \underline{\langle\text{factor}\rangle} * \langle\text{id}\rangle$
8	$\langle\text{expr}\rangle - \underline{\langle\text{num}\rangle} * \langle\text{id}\rangle$
4	$\underline{\langle\text{term}\rangle} - \langle\text{num}\rangle * \langle\text{id}\rangle$
7	$\underline{\langle\text{factor}\rangle} - \langle\text{num}\rangle * \langle\text{id}\rangle$
9	$\underline{\langle\text{id}\rangle} - \langle\text{num}\rangle * \langle\text{id}\rangle$

Handle position

In shift-reduce parsing, handles will appear only at the top of the stack.

Proof.

The two successive steps in a rightmost derivation will be of the form:

1. $S \overset{rm}{\Longrightarrow}^{*} \alpha A z \overset{rm}{\Longrightarrow} \alpha\beta B y z \overset{rm}{\Longrightarrow} \alpha\beta\gamma y z$ (for $A \rightarrow \beta B y$ and $B \rightarrow \gamma$)
2. $S \overset{rm}{\Longrightarrow}^{*} \alpha B x A z \overset{rm}{\Longrightarrow} \alpha B x y z \overset{rm}{\Longrightarrow} \alpha\gamma x y z$ (for $A \rightarrow y$ and $B \rightarrow \gamma$)

where x, y, z string of terminals. A is the right-most non-terminal in both cases.

Case 1:

STACK	INPUT	ACTION
$\$\alpha\beta\gamma$	$yz\$$	
$\$\alpha\beta B$	$yz\$$	R
$\$\alpha\beta B y$	$z\$$	S
$\$\alpha A$	$z\$$	R

Case 2:

STACK	INPUT	ACTION
$\$\alpha\gamma$	$xyz\$$	
$\$\alpha B$	$xyz\$$	R
$\$\alpha B x y$	$z\$$	S
$\$\alpha B x A$	$z\$$	R

When to shift and when to reduce?

Consider:

$$\begin{array}{lcl} E & \rightarrow & E+T \mid E-T \mid T \\ T & \rightarrow & T*F \mid T/F \mid F \\ F & \rightarrow & \texttt{num} \mid \texttt{id} \end{array}$$

- We know that the handle will appear on the top of the stack.
- But we still don't know when to shift and when to reduce.
  - For example, while parsing $\texttt{id}*\texttt{id}$, at the stage $T \mid *\texttt{id}$, we should not reduce using $E \rightarrow T$.
  - Intuitively, this is because $E*$ is never a prefix of a right-sentential form in the grammar.

Viable Prefix

- α is a *viable prefix* if there is an ω such that $\alpha \mid \omega$ is a state of a shift-reduce parser.
- A viable prefix does not extend past the right end of the handle.
  - The suffix of a viable prefix either is a handle, or it can be expanded into a handle by shifting.

Not all prefixes right-sentential forms are viable prefixes. Consider:

$$E \overset{rm}{\Longrightarrow} T \overset{rm}{\Longrightarrow} T * F \overset{rm}{\Longrightarrow} T * \texttt{id} \overset{rm}{\Longrightarrow} F * \texttt{id} \overset{rm}{\Longrightarrow} \texttt{id} * \texttt{id}$$

- While `id *` is a prefix of right-sential form, it is not a viable prefix as it does not appear on the shift-reduce stack.
  - It extends past the handle `id`.

Important fact about viable prefixes

For any grammar, the set of viable prefixes is a regular language.

- We show how to compute an automata that accepts viable prefixes.
- Such an automata can help automate shift-reduce decisions.
  - If the automata permits a transition on a symbol to another valid state (another viable prefix), I can shift as I can eventually find a handle.
  - Otherwise, I will have to reduce.

Items

We shall use the concept of items to help build the automata that recognizes viable prefixes.

An *item* is a production with a $\bullet$ somewhere on the RHS, denoting a focus point.

The $\bullet$ indicates how much of an item we have seen at a given state in the parse:

$[A \rightarrow \bullet XYZ]$	indicates that the parser is looking for a string that can be derived from XYZ
$[A \rightarrow XY \bullet Z]$	indicates that the parser has seen a string derived from XY and is looking for one derivable from Z

$A \rightarrow XYZ$ generates 4 items:

1. $[A \rightarrow \bullet XYZ]$
2. $[A \rightarrow X \bullet YZ]$
3. $[A \rightarrow XY \bullet Z]$
4. $[A \rightarrow XYZ\bullet]$

Intuition

- The problem in recognizing viable prefixes is that the stack has only bits and pieces of the rhs of productions.
  - If it had a complete rhs, we could reduce
- These bits and pieces are always prefixes of RHS of productions.

Consider:

$$\begin{array}{lcl} E & \rightarrow & E+T \mid E-T \mid T \\ T & \rightarrow & T*F \mid T/F \mid F \\ F & \rightarrow & \texttt{num} \mid \texttt{id} \end{array}$$

Consider the string `id*id`. While parsing, consider the state $T* \mid \texttt{id}$.

- $T*$ is a prefix of the RHS of $T \rightarrow T*F$.
- The corresponding item would be $T \rightarrow T*\bullet F$.

Generalization

- In general, the stack may have many prefixes of RHSs: $\mathit{Prefix}_1 \mathit{Prefix}_2 \ldots \mathit{Prefix}_n$
- Let Prefix_i be a prefix of RHS of $X_i \rightarrow \alpha_i$.
  - Prefix_i will eventually reduce to X_i.
  - The missing part of α_{i-1} starts with X_i, i.e. there is a production $X_{i-1} \rightarrow \mathit{Prefix}_{i-1} X_i \beta$ for some β.
- Recursively, $\mathit{Prefix}_{k+1} \ldots \mathit{Prefix}_n$ eventually reduces to the missing part of α_k.

Example

Consider:

$$\begin{array}{lcl} E & \rightarrow & E + T \mid E - T \mid T \\ T & \rightarrow & T * F \mid T / F \mid F \\ F & \rightarrow & \texttt{num} \mid \texttt{id} \end{array}$$

- Consider the string $\texttt{id} + \texttt{id} * \texttt{id}$.
  - $E + T * \mid \texttt{id}$ is a state of shift-reduce parse.
- From the top of the stack:
  - $T*$ is a prefix of $T \rightarrow T * F$
  - $E+$ is a prefix of $E \rightarrow E + T$
- We can consider the stack to contain a stack of items. From the top:
  - $T \rightarrow T * \bullet F$ – we've seen $T*$; hope to see F.
  - $E \rightarrow E + \bullet T$ – we've seen $E+$; hope to see T.

Recognizing Viable Prefixes

Idea: To recognize viable prefixes, we must

1. Recognize a sequence of partial RHS's of productions, such that
2. Each partial RHS can eventually reduce to part of the missing suffix of its predecessor in the sequence.

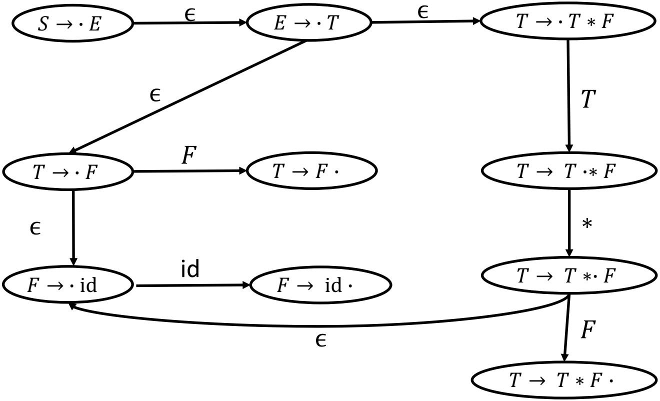
An NFA recognizing Viable Prefixes

1. Add a new start production $S' \to S$ to the grammar.
2. The NFA states are the items of the grammar.
   - The start state will be $S' \to \bullet S$
3. For item $E \to \alpha \bullet X\beta$, add transition $E \to \alpha \bullet X\beta \overset{X}{\rightsquigarrow} E \to \alpha X \bullet \beta$.
4. For item $E \to \alpha \bullet X\beta$ and production $X \to \gamma$, add transition $E \to \alpha \bullet X\beta \overset{\varepsilon}{\rightsquigarrow} X \to \bullet\gamma$.
5. Every state is an accepting state.

Example

Grammer G :

$$
\begin{array}{c|ccl}
1 & S & \rightarrow & E \\
2 & E & \rightarrow & E+T \\
3 & & | & E-T \\
4 & & | & T \\
5 & T & \rightarrow & T*F \\
6 & & | & T/F \\
7 & & | & F \\
8 & F & \rightarrow & \langle \mathrm{num} \rangle \\
9 & & | & \langle \mathrm{id} \rangle
\end{array}
$$



$S \rightarrow \cdot E$
ϵ
$E \rightarrow \cdot T$
ϵ
$T \rightarrow \cdot T * F$
ϵ
T
F
$T \rightarrow \cdot F$
$T \rightarrow F \cdot$
$T \rightarrow T \cdot * F$
ϵ
$*$
id
$F \rightarrow \cdot \mathrm{id}$
$F \rightarrow \mathrm{id} \cdot$
$T \rightarrow T * \cdot F$
ϵ
F
$T \rightarrow T * F \cdot$

Portion of the NFA for recognizing viable prefixes of G.

Recall: Shift-Reduce Parsing $\texttt{id}*\texttt{id}$

$\mid \texttt{id} \star \texttt{id}$	
$\texttt{id} \mid \star\ \texttt{id}$	Shift
$F \mid \star\ \texttt{id}$	Reduce by $F \rightarrow \texttt{id}$
$T \mid \star\ \texttt{id}$	Reduce by $T \rightarrow F$
$T \star \texttt{id} \mid$	Shift
$T \star F \mid$	Reduce by $F \rightarrow \texttt{id}$
$T \mid$	Reduce by $T \rightarrow T \star F$
$E \mid$	Reduce by $E \rightarrow T$

Recall: Shift-Reduce Parsing `id*id`

$|\,$id $\star$ id

id$\,|\,\star$ id

$F\,|\,\star$ id

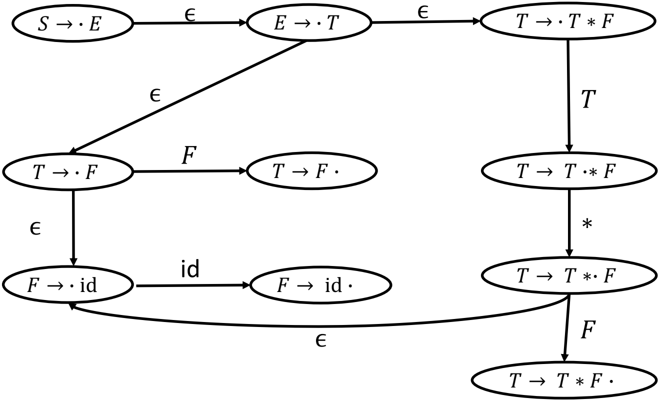
$T\,|\,\star$ id

$T\,\star$ id$\,|$

$T \star F\,|$

$T\,|$

$E\,|$



The NFA recognizes all the viable prefixes encountered during the parse.

DFA for recognizing viable prefixes

- We can convert the NFA to a DFA.
    - Each state will now be a set of items.
    - Transitions will be on a grammar symbol.
- The states of this DFA are called “canonical collection of items” or “canonical collection of LR(0) items”.
    - Each item that we have described so far is also called a LR(0) item.
- The Dragon Book defines procedures `CLOSURE` and `GOTO` to directly generate the DFA.
- This DFA is also sometimes called the Characteristic Finite State Machine (CFSM) of the grammar.

CLOSURE

Let I be a set of LR(0) items.

```
function CLOSURE($I$)
repeat
  if $[A \rightarrow \alpha \bullet B\beta] \in I$
    add $[B \rightarrow \bullet\gamma]$ to $I$
until no more items can be added to $I$
return $I$
```

Note that this is nothing but the ε-closure of states in the NFA.

GOTO

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
$[A \rightarrow \alpha X \bullet \beta]$ *such that* $[A \rightarrow \alpha \bullet X \beta] \in I$

GOTO(I,X) represents state after recognizing X in state I.

```
function GOTO($I,X$)
  let $J$ be the set of items $[A \rightarrow \alpha X \bullet \beta]$
    such that $[A \rightarrow \alpha \bullet X \beta] \in I$
  return CLOSURE($J$)
```

Building the LR(0) item sets

We start the construction with the item $[S' \rightarrow \bullet S\$]$, where

S' is the start symbol of the augmented grammar G'
S is the start symbol of G
\$ represents `EOF`

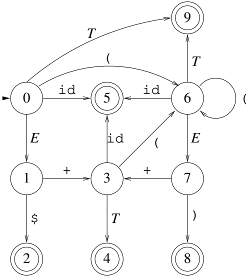
To compute the collection of sets of LR(0) items

```
function items($G'$)
  $s_0 \leftarrow$ CLOSURE($\{[S' \rightarrow \bullet S\$]\}$)
  $C \leftarrow \{s_0\}$
  repeat
    for each set of items $s \in C$
      for each grammar symbol $X$
        if GOTO($s,X$) $\neq \phi$ and GOTO($s,X$) $\notin C$
          add GOTO($s,X$) to $C$
  until no more item sets can be added to $C$
  return $C$
```

LR(0): Example

$$\begin{array}{c|ccl}
1 & S & \rightarrow & E\$ \\
2 & E & \rightarrow & E+T \\
3 & & \mid & T \\
4 & T & \rightarrow & \langle\text{id}\rangle \\
5 & & \mid & (E)
\end{array}$$

The corresponding DFA:



$$\begin{aligned}
I_0 : & S \rightarrow \bullet E\$ \\
& E \rightarrow \bullet E+T \\
& E \rightarrow \bullet T \\
& T \rightarrow \bullet\langle\text{id}\rangle \\
& T \rightarrow \bullet(E) \\
I_1 : & S \rightarrow E \bullet \$ \\
& E \rightarrow E \bullet +T \\
I_2 : & S \rightarrow E\$\bullet \\
I_3 : & E \rightarrow E+\bullet T \\
& T \rightarrow \bullet\langle\text{id}\rangle \\
& T \rightarrow \bullet(E)
\end{aligned}$$

$$\begin{aligned}
I_4 : & E \rightarrow E+T\bullet \\
I_5 : & T \rightarrow \langle\text{id}\rangle\bullet \\
I_6 : & T \rightarrow (\bullet E) \\
& E \rightarrow \bullet E+T \\
& E \rightarrow \bullet T \\
& T \rightarrow \bullet\langle\text{id}\rangle \\
& T \rightarrow \bullet(E) \\
I_7 : & T \rightarrow (E\bullet) \\
& E \rightarrow E \bullet +T \\
I_8 : & T \rightarrow (E)\bullet \\
I_9 : & E \rightarrow T\bullet
\end{aligned}$$

Valid Items

- Item $X \rightarrow \beta \bullet \gamma$ is *valid* for a viable prefix $\alpha\beta$ if

  $$S \Rightarrow^* \alpha X \omega \Rightarrow \alpha\beta\gamma\omega$$

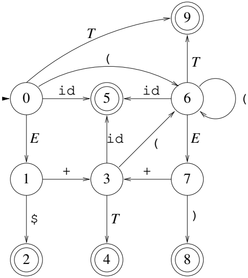
  by a right-most derivation.
- After parsing $\alpha\beta$, the valid items are the possible tops of the stack of items.

- Alternatively, an item I is valid for a viable prefix α if the DFA recognizing viable prefixes terminates on input α in a state s containing I.
  - The items in s describe what the top of the item stack might be after reading input α.

Valid Items: Example

$$\begin{array}{c|lcl} 1 & S & \to & E\$ \\ 2 & E & \to & E+T \\ 3 & & | & T \\ 4 & T & \to & \langle\text{id}\rangle \\ 5 & & | & (E) \end{array}$$

The corresponding DFA:



$$\begin{array}{ll} I_0: & S \to \bullet E\$ \\ & E \to \bullet E+T \\ & E \to \bullet T \\ & T \to \bullet\langle\text{id}\rangle \\ & T \to \bullet(E) \\ I_1: & S \to E\bullet\$ \\ & E \to E\bullet +T \\ I_2: & S \to E\$\bullet \\ I_3: & E \to E+\bullet T \\ & T \to \bullet\langle\text{id}\rangle \\ & T \to \bullet(E) \end{array}$$

$$\begin{array}{ll} I_4: & E \to E+T\bullet \\ I_5: & T \to \langle\text{id}\rangle\bullet \\ I_6: & T \to (\bullet E) \\ & E \to \bullet E+T \\ & E \to \bullet T \\ & T \to \bullet\langle\text{id}\rangle \\ & T \to \bullet(E) \\ I_7: & T \to (E\bullet) \\ & E \to E\bullet +T \\ I_8: & T \to (E)\bullet \\ I_9: & E \to T\bullet \end{array}$$

$T \to (\bullet E)$ is a valid item for (. Also for $E+(,((,E+(($.

Recall: Stack implementation of Shift-Reduce Parsing

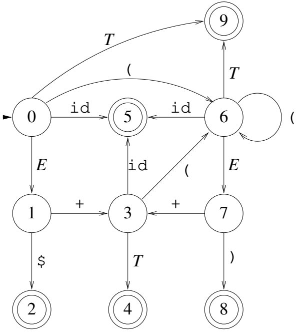
Shift-reduce parsers use a *stack* and an *input buffer*

1. initialize stack with $
2. Repeat until the top of the stack is the goal symbol and the input token is $
   - a) *find the handle*
     if we don't have a handle on top of the stack, *shift* an input symbol onto the stack
   - b) *prune the handle*
     if we have a handle $A \rightarrow \beta$ on the top of the stack, *reduce*
     - i) pop $\mid \beta \mid$ symbols off the stack
     - ii) push A onto the stack

Basic LR(0) Parsing

- Assume
  - stack contains α
  - next input symbol is a
  - DFA on input α terminates in state s
- Shift if s contains the item $X \to \beta \bullet a\omega$.
  - Equivalent to saying that state s has a transition labelled a.
- Reduce by $X \to \beta$ if s contains the item $X \to \beta\bullet$.
  - That is, pop $|\beta|$ symbols from the stack and push X.
- Accept if the stack contains S and input token in \$.
- Report an error if no shift/reduce moves are possible.

Example: Parsing `id + id`



$I_0 : S \to \bullet E\$$
$E \to \bullet E + T$
$E \to \bullet T$
$T \to \bullet\langle \text{id} \rangle$
$T \to \bullet(E)$
$I_1 : S \to E \bullet \$$
$E \to E \bullet + T$
$I_2 : S \to E\$\bullet$
$I_3 : E \to E + \bullet T$
$T \to \bullet\langle \text{id} \rangle$
$T \to \bullet(E)$

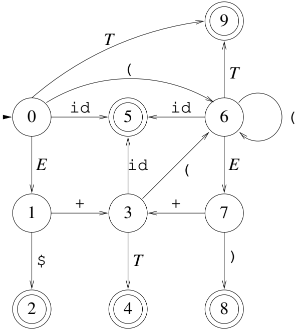
$I_4 : E \to E + T\bullet$
$I_5 : T \to \langle \text{id} \rangle\bullet$
$I_6 : T \to (\bullet E)$
$E \to \bullet E + T$
$E \to \bullet T$
$T \to \bullet\langle \text{id} \rangle$
$T \to \bullet(E)$
$I_7 : T \to (E\bullet)$
$E \to E \bullet + T$
$I_8 : T \to (E)\bullet$
$I_9 : E \to T\bullet$

| `id`+ `id`$	$\hat{\delta}(0, \varepsilon) = 0$	Shift `id`
`id`| + `id`$	$\hat{\delta}(0, \texttt{id}) = 5$	Reduce $T \to \texttt{id}$
T | + `id`$	$\hat{\delta}(0, T) = 9$	Reduce $E \to T$
E | + `id`$	$\hat{\delta}(0, E) = 1$	Shift +
E + | `id`$	$\hat{\delta}(0, E+) = 3$	Shift `id`
E + `id`| $	$\hat{\delta}(0, E + \texttt{id}) = 5$	Reduce $T \to \texttt{id}$
E + T | $	$\hat{\delta}(0, E + T) = 4$	Reduce $E \to E + T$
E | $	$\hat{\delta}(0, E) = 1$	Shift $
E $ |	$\hat{\delta}(0, E\$) = 2$	Accept

An Optimization

- Rerunning the automaton from the start state at each step is wasteful
  - Much of the work is repeated.

Example: Repeated Work in Basic LR(0) Parsing



$I_0 : S \rightarrow \bullet E\$$

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

$\quad E \rightarrow \bullet T$

$\quad T \rightarrow \bullet \langle \text{id} \rangle$

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

$I_1 : S \rightarrow E \bullet \$$

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

$I_2 : S \rightarrow E\$ \bullet$

$I_3 : E \rightarrow E + \bullet T$

$\quad T \rightarrow \bullet \langle \text{id} \rangle$

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

$I_4 : E \rightarrow E + T \bullet$

$I_5 : T \rightarrow \langle \text{id} \rangle \bullet$

$I_6 : T \rightarrow (\bullet E)$

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

$\quad E \rightarrow \bullet T$

$\quad T \rightarrow \bullet \langle \text{id} \rangle$

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

$I_7 : T \rightarrow (E \bullet)$

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

$I_8 : T \rightarrow (E) \bullet$

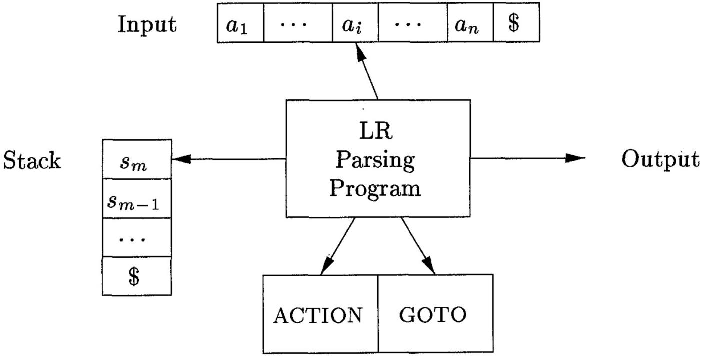
$I_9 : E \rightarrow T \bullet$

| id+ id$	$\hat{\delta}(0, \varepsilon) = 0$	Shift id
id| + id$	$\hat{\delta}(0, \text{id}) = 5$	Reduce $T \rightarrow$ id
T | + id$	$\hat{\delta}(0, T) = 9$	Reduce $E \rightarrow T$
E | + id$	$\hat{\delta}(0, E) = 1$	Shift +
E + | id$	$\hat{\delta}(0, E+) = 3$	Shift id
E + id| $	$\hat{\delta}(0, E + \text{id}) = 5$	Reduce $T \rightarrow$ id
E + T | $	$\hat{\delta}(0, E + T) = 4$	Reduce $E \rightarrow E + T$
E | $	$\hat{\delta}(0, E) = 1$	Shift $
E $ |	$\hat{\delta}(0, E\$) = 2$	Accept

An Optimization

- Rerunning the automaton from the start state at each step is wasteful
  - Much of the work is repeated.
- Instead, we can remember the state of the automaton for each prefix of the stack.
  - This state can be stored on the stack itself.
  - In fact, we will only store states on the stack now.
- Optimized LR(0) parsing algorithm uses two tables: ACTION and GOTO.
  - ACTION(i,a) is defined for every state i of the DFA and every terminal symbol a.
  - GOTO(i,A) is defined for every state i of the DFA and every non-terminal symbol A.

Model of an LR Parser



Constructing the LR(0) parsing table

1. construct the collection of sets of LR(0) items for the grammar
2. state i of the DFA is constructed from I_i
    1. $[A \to \alpha \bullet a\beta] \in I_i$ and $\texttt{GOTO}(I_i, a) = I_j$
       $\Rightarrow$ ACTION$[i,a] \leftarrow$ "*shift j*", $\forall a \neq \$$
    2. $[A \to \alpha\bullet] \in I_i, A \neq S'$
       $\Rightarrow$ ACTION$[i,a] \leftarrow$ "*reduce* $A \to \alpha$", $\forall a$
    3. $[S' \to S \bullet \$] \in I_i$
       $\Rightarrow$ ACTION$[i,\$] \leftarrow$ "*accept*",
3. $\texttt{GOTO}(I_i, A) = I_j$
   $\Rightarrow$ GOTO$[i,A] \leftarrow j$
4. set undefined entries in ACTION and GOTO to "*error*"
5. initial state of parser s_0 is $\texttt{CLOSURE}([S' \to \bullet S\$])$

LR Parsing Algorithm

The skeleton parser:

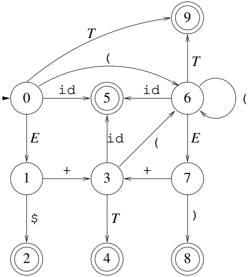
```
push $s_0$
token $\leftarrow$ next_token()
repeat forever
  s $\leftarrow$ top of stack
  if action[s,token] = "*shift* $s_i$" then
    push $s_i$
    token $\leftarrow$ next_token()
  else if action[s,token] = "*reduce* $A \rightarrow \beta$"
    then
    pop $|\beta|$ states
    s$'$ $\leftarrow$ top of stack
    push goto[s$'$,$A$]
  else if action[s, token] = "*accept*" then
    return
  else error()
```

"**How many** ops?": k shifts, l reduces, and 1 accept, where k is length of input string and l is length of reverse rightmost derivation

LR(0) Parsing Table: Example

$$\begin{array}{c|lcl}
1 & S & \rightarrow & E\$ \\
2 & E & \rightarrow & E+T \\
3 & & \mid & T \\
4 & T & \rightarrow & \langle\text{id}\rangle \\
5 & & \mid & (E)
\end{array}$$

The corresponding DFA:



$I_0 : S \rightarrow \bullet E\$$

$E \rightarrow \bullet E + T$

$E \rightarrow \bullet T$

$T \rightarrow \bullet\langle\text{id}\rangle$

$T \rightarrow \bullet(E)$

$I_1 : S \rightarrow E \bullet \$$

$E \rightarrow E \bullet + T$

$I_2 : S \rightarrow E\$\bullet$

$I_3 : E \rightarrow E + \bullet T$

$T \rightarrow \bullet\langle\text{id}\rangle$

$T \rightarrow \bullet(E)$

$I_4 : E \rightarrow E + T\bullet$

$I_5 : T \rightarrow \langle\text{id}\rangle\bullet$

$I_6 : T \rightarrow (\bullet E)$

$E \rightarrow \bullet E + T$

$E \rightarrow \bullet T$

$T \rightarrow \bullet\langle\text{id}\rangle$

$T \rightarrow \bullet(E)$

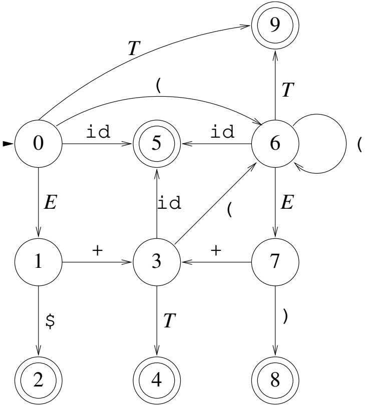
$I_7 : T \rightarrow (E\bullet)$

$E \rightarrow E \bullet + T$

$I_8 : T \rightarrow (E)\bullet$

$I_9 : E \rightarrow T\bullet$

LR(0) Parsing Table: Example



state	ACTION					GOTO		
	id	(	)	+	$	*S*	*E*	*T*
0	s5	s6	–	–	–	–	1	9
1	–	–	–	s3	acc	–	–	–
2	–	–	–	–	–	–	–	–
3	s5	s6	–	–	–	–	–	4
4	r2	r2	r2	r2	r2	–	–	–
5	r4	r4	r4	r4	r4	–	–	–
6	s5	s6	–	–	–	–	7	9
7	–	–	s8	s3	–	–	–	–
8	r5	r5	r5	r5	r5	–	–	–
9	r3	r3	r3	r3	r3	–	–	–

LR Parsing Algorithm: Parsing `id + id`

$$
\begin{array}{r|lcl}
1 & S & \rightarrow & E\$ \\
2 & E & \rightarrow & E + T \\
3 & & | & T \\
4 & T & \rightarrow & \langle \mathrm{id} \rangle \\
5 & & | & (E)
\end{array}
$$

Stack	**Input**
0	`id`+`id`$
05	+`id`$
09	+`id`$
01	+`id`$
013	`id`$
0135	$
0134	$
01	$

state	ACTION					GOTO		
	`id`	(	)	+	$	*S*	*E*	*T*
0	s5	s6	–	–	–	–	1	9
1	–	–	–	s3	acc	–	–	–
2	–	–	–	–	–	–	–	–
3	s5	s6	–	–	–	–	–	4
4	r2	r2	r2	r2	r2	–	–	–
5	r4	r4	r4	r4	r4	–	–	–
6	s5	s6	–	–	–	–	7	9
7	–	–	s8	s3	–	–	–	–
8	r5	r5	r5	r5	r5	–	–	–
9	r3	r3	r3	r3	r3	–	–	–

LR(0) Conflicts

- LR(0) has a reduce/reduce conflict if any state has two reduce items: $X \to \alpha \bullet$ and $Y \to \beta \bullet$.
    - Our running example of the simple expression grammar with just + and () does not have reduce-reduce conflicts.
- LR(0) has a shift/reduce conflict if any state has a reduce item and a shift item: $X \to \alpha \bullet$ and $Y \to \beta \bullet a\gamma$.
    - Our running example of the simple expression grammar does not have shift/reduce conflicts as well.

Conflicts in the ACTION table

LR(0) conflicts will manifest in the ACTION table as multiple entries for some cell.
Conflicts can be resolved through *lookahead*. Consider:

- $A \to \varepsilon \mid a\alpha$
  $\Rightarrow$ shift-reduce conflict
- `a:=b+c*d`
  requires lookahead to avoid shift-reduce conflict after shifting `c` (need to see $\star$ to give precedence over $+$)

LR parsing with lookahead

Three common techniques to build LR parsers with lookahead:

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

Here k indicates the number of lookahead symbols
We will study SLR(1), LR(1) and LALR(1).

Why study LR parsers?

- LR parsers can be constructed for virtually all context-free programming language constructs
- LR-parsing is the most general non-backtracking shift-reduce parsing method known. It is also one of the most efficient parsing methods.
- LR parsers detect an error as soon as possible in a left-to-right scan of the input
- LR grammars describe a proper superset of the languages recognized by predictive (i.e., LL) parsers

  LL(k): recognize use of a production $A \rightarrow \beta$ seeing first k symbols derived from β

  LR(k): recognize the handle β after seeing everything derived from β plus k lookahead symbols

Basic SLR(1) Parsing: Simple Lookahead LR

- Assume
  - stack contains α
  - next input symbol is a
  - DFA on stack α terminates in state s
- Shift if s contains the item $X \to \beta \bullet a\omega$.
  - Equivalent to saying that state s has a transition labelled a.
- Reduce by $X \to \beta$ if s contains the item $X \to \beta\bullet$ and $a \in$ FOLLOW(X).
  - That is, pop $|\beta|$ symbols from the stack and push X.
- Accept if the stack contains S and input token in \$.
- Report an error if no shift/reduce moves are possible.

Optimized SLR(1)

Add lookaheads after building LR(0) item sets
Constructing the SLR(1) parsing table:

1. construct the collection of sets of LR(0) items for G'
2. state i of the DFA is constructed from the item set I_i
    1. $[A \rightarrow \alpha \bullet a\beta] \in I_i$ and $\texttt{GOTO}(I_i, a) = I_j$
       $\Rightarrow$ ACTION$[i,a] \leftarrow$ "*shift* j", $\forall a \neq \$$
    2. $[A \rightarrow \alpha\bullet] \in I_i, A \neq S'$
       $\Rightarrow$ ACTION$[i,a] \leftarrow$ "*reduce* $A \rightarrow \alpha$", $\forall a \in$ FOLLOW(A)
    3. $[S' \rightarrow S \bullet \$] \in I_i$
       $\Rightarrow$ ACTION$[i,\$] \leftarrow$ "*accept*"
3. $\texttt{GOTO}(I_i, A) = I_j$
   $\Rightarrow$ GOTO$[i,A] \leftarrow j$
4. set undefined entries in ACTION and GOTO to "*error*"
5. initial state of parser s_0 is $\texttt{CLOSURE}([S' \rightarrow \bullet S\$])$

Example: A grammar that is not LR(0)

$$
\begin{array}{r|lcl}
1 & S & \rightarrow & E\$ \\
2 & E & \rightarrow & E+T \\
3 & & | & T \\
4 & T & \rightarrow & T*F \\
5 & & | & F \\
6 & F & \rightarrow & \langle \mathrm{id} \rangle \\
7 & & | & (E)
\end{array}
$$

	FOLLOW
E	$\{+,),\$\}$
T	$\{+,*,),\$\}$
F	$\{+,*,),\$\}$

$$
\begin{array}{ll}
I_0: & S \rightarrow \bullet E\$ \\
& E \rightarrow \bullet E+T \\
& E \rightarrow \bullet T \\
& T \rightarrow \bullet T*F \\
& T \rightarrow \bullet F \\
& F \rightarrow \bullet \langle \mathrm{id} \rangle \\
& F \rightarrow \bullet (E) \\
I_1: & S \rightarrow E \bullet \$ \\
& E \rightarrow E \bullet +T \\
I_2: & S \rightarrow E\$\bullet \\
I_3: & E \rightarrow E+\bullet T \\
& T \rightarrow \bullet T*F \\
& T \rightarrow \bullet F \\
& F \rightarrow \bullet \langle \mathrm{id} \rangle \\
& F \rightarrow \bullet (E) \\
I_4: & T \rightarrow F\bullet \\
I_5: & F \rightarrow \langle \mathrm{id} \rangle \bullet
\end{array}
\qquad
\begin{array}{ll}
I_6: & F \rightarrow (\bullet E) \\
& E \rightarrow \bullet E+T \\
& E \rightarrow \bullet T \\
& T \rightarrow \bullet T*F \\
& T \rightarrow \bullet F \\
& F \rightarrow \bullet \langle \mathrm{id} \rangle \\
& F \rightarrow \bullet (E) \\
I_7: & E \rightarrow T\bullet \\
& T \rightarrow T \bullet *F \\
I_8: & T \rightarrow T* \bullet F \\
& F \rightarrow \bullet \langle \mathrm{id} \rangle \\
& F \rightarrow \bullet (E) \\
I_9: & T \rightarrow T*F\bullet \\
I_{10}: & F \rightarrow (E)\bullet \\
I_{11}: & E \rightarrow E+T\bullet \\
& T \rightarrow T \bullet *F \\
I_{12}: & F \rightarrow (E\bullet) \\
& E \rightarrow E \bullet +T
\end{array}
$$

Shift/Reduce Conflicts

Example: But it is SLR(1)

state	ACTION						GOTO			
	+	*	id	(	)	$	S	E	T	F
0	–	–	s5	s6	–	–	–	1	7	4
1	s3	–	–	–	–	acc	–	–	–	–
2	–	–	–	–	–	–	–	–	–	–
3	–	–	s5	s6	–	–	–	–	11	4
4	r5	r5	–	–	r5	r5	–	–	–	–
5	r6	r6	–	–	r6	r6	–	–	–	–
6	–	–	s5	s6	–	–	–	12	7	4
7	r3	s8	–	–	r3	r3	–	–	–	–
8	–	–	s5	s6	–	–	–	–	–	9
9	r4	r4	–	–	r4	r4	–	–	–	–
10	r7	r7	–	–	r7	r7	–	–	–	–
11	r2	s8	–	–	r2	r2	–	–	–	–
12	s3	–	–	–	s10	–	–	–	–	–

Example: A grammar that is not SLR(1)

Consider:

$$
\begin{array}{lcl}
S & \rightarrow & L = R \\
 & \mid & R \\
L & \rightarrow & *R \\
 & \mid & \langle \text{id} \rangle \\
R & \rightarrow & L
\end{array}
$$

Its LR(0) item sets:

$$
\begin{array}{ll}
I_0 : & S' \rightarrow \bullet S\$ \\
 & S \rightarrow \bullet L = R \\
 & S \rightarrow \bullet R \\
 & L \rightarrow \bullet * R \\
 & L \rightarrow \bullet \langle \text{id} \rangle \\
 & R \rightarrow \bullet L \\
I_1 : & S' \rightarrow S \bullet \$ \\
I_2 : & S \rightarrow L \bullet = R \\
 & R \rightarrow L \bullet \\
I_3 : & S \rightarrow R \bullet \\
I_4 : & L \rightarrow \langle \text{id} \rangle \bullet
\end{array}
$$

$$
\begin{array}{ll}
I_5 : & L \rightarrow * \bullet R \\
 & R \rightarrow \bullet L \\
 & L \rightarrow \bullet * R \\
 & L \rightarrow \bullet \langle \text{id} \rangle \\
I_6 : & S \rightarrow L = \bullet R \\
 & R \rightarrow \bullet L \\
 & L \rightarrow \bullet * R \\
 & L \rightarrow \bullet \langle \text{id} \rangle \\
I_7 : & L \rightarrow * R \bullet \\
I_8 : & R \rightarrow L \bullet \\
I_9 : & S \rightarrow L = R \bullet
\end{array}
$$

Now consider I_2: $= \in \text{FOLLOW}(R)$ ($S \Rightarrow L = R \Rightarrow *R = R$)

I_2 has a shift/reduce conflict.

Consider parsing $*id = id$.

Example: A grammar that is not SLR(1)

Consider:

$$\begin{array}{lcl} S & \to & L = R \\ & \mid & R \\ L & \to & *R \\ & \mid & \langle \mathrm{id} \rangle \\ R & \to & L \end{array}$$

Its LR(0) item sets:

$I_0 : S' \to \bullet S\$$
$S \to \bullet L = R$
$S \to \bullet R$
$L \to \bullet * R$
$L \to \bullet \langle \mathrm{id} \rangle$
$R \to \bullet L$

$I_1 : S' \to S \bullet \$$

$I_2 : S \to L \bullet = R$
$R \to L \bullet$

$I_3 : S \to R \bullet$

$I_4 : L \to \langle \mathrm{id} \rangle \bullet$

$I_5 : L \to * \bullet R$
$R \to \bullet L$
$L \to \bullet * R$
$L \to \bullet \langle \mathrm{id} \rangle$

$I_6 : S \to L = \bullet R$
$R \to \bullet L$
$L \to \bullet * R$
$L \to \bullet \langle \mathrm{id} \rangle$

$I_7 : L \to *R \bullet$

$I_8 : R \to L \bullet$

$I_9 : S \to L = R \bullet$

While parsing $*id = id$, at the parse state $L \mid= id$, the correct option is to shift.

While parsing id, at the parse state $L \mid$, the correct option is to reduce by $R \to L$. Note that this is the only string where reduce is the correct option for item-set I_2.

LR(k) items

A LR(k) item is a pair $[\alpha, \beta]$, where

α is a production from G with a $\bullet$ at some position in the RHS, marking how much of the RHS of a production has already been seen

β is a lookahead string containing k symbols (terminals or `$`)

A LR(k) item $[A \rightarrow \alpha \bullet \beta, w]$ is valid for a viable prefix $\gamma\alpha$ iff

- there exists a rightmost derivation $S \Rightarrow_{rm}^{*} \gamma A x \Rightarrow_{rm} \gamma\alpha\beta x$ and
- $x = ww'$ or x is ε and w is \$.

LR(1) items

Will have the general form $[A \rightarrow \alpha \bullet \beta, a]$. What's the point of the lookahead symbols?

Choose correct reduction when there is a choice

- lookaheads are bookkeeping, unless item has $\bullet$ at right end:
  - in $[A \rightarrow X \bullet YZ, a]$, a has no direct use
  - in $[A \rightarrow XYZ\bullet, a]$, a is useful
- For item $[A \rightarrow XYZ\bullet, a]$, we will reduce only if the next input symbol is a.

closure1(I)

```
function closure1($I$)
repeat
  if $[A \rightarrow \alpha \bullet B\beta, a] \in I$
    add $[B \rightarrow \bullet\gamma, b]$ to $I$, where $b \in \text{FIRST}(\beta a)$
until no more items can be added to $I$
return $I$
```

Intuition:

- If $[A \rightarrow \alpha \bullet B\beta, a]$ is a valid item for viable prefix $\delta\alpha$, then $S \overset{rm}{\Longrightarrow}^{*} \delta A a x \overset{rm}{\Longrightarrow} \delta\alpha B\beta a x$.
- Suppose $\beta a x$ derives by. Then, for each of the productions of the form $B \rightarrow \gamma$, we have a derivation $S \overset{rm}{\Longrightarrow}^{*} \delta\alpha B b y \overset{rm}{\Longrightarrow} \delta\alpha\gamma b y$.
- This would imply that $[B \rightarrow \bullet\gamma, b]$ would be a valid item for viable prefix $\delta\alpha$ for all $b \in \text{FIRST}(\beta a)$. Note $\text{FIRST}(\beta a) = \text{FIRST}(\beta a x)$.

goto1(I)

Let I be a set of LR(1) items and X be a grammar symbol.
Then, GOTO1(I,X) is the closure of the set of all items

$$[A \rightarrow \alpha X \bullet \beta, a] \textit{ such that } [A \rightarrow \alpha \bullet X\beta, a] \in I$$

If I is the set of valid items for some viable prefix γ, then GOTO1(I,X) is the set of valid items for the viable prefix γX.
goto1(I,X) represents state after recognizing X in state I.

```
function goto1($I$,$X$)
  let $J$ be the set of items $[A \rightarrow \alpha X \bullet \beta, a]$
    such that $[A \rightarrow \alpha \bullet X\beta, a] \in I$
  return closure1($J$)
```

Building the LR(1) item sets for grammar G

We start the construction with the item $[S' \rightarrow \bullet S, \$]$, where

- S' is the start symbol of the augmented grammar G'
- S is the start symbol of G
- \$ represents `EOF`

To compute the collection of sets of LR(1) items

```
function items($G'$)
  $s_0 \leftarrow$ closure1($\{[S' \rightarrow \bullet S, \$]\}$)
  $C \leftarrow \{s_0\}$
  repeat
    for each set of items $s \in C$
      for each grammar symbol $X$
        if goto1$(s,X) \neq \phi$ and goto1$(s,X) \notin C$
          add goto1$(s,X)$ to $C$
  until no more item sets can be added to $C$
  return $C$
```

Constructing the LR(1) parsing table

Build lookahead into the DFA to begin with

1. construct the collection of sets of LR(1) items for G'
2. state i of the LR(1) machine is constructed from I_i
    1. $[A \rightarrow \alpha \bullet a\beta, b] \in I_i$ and $\texttt{goto1}(I_i, a) = I_j$
       $\Rightarrow$ ACTION$[i, a] \leftarrow$ "*shift j*"
    2. $[A \rightarrow \alpha\bullet, a] \in I_i, A \neq S'$
       $\Rightarrow$ ACTION$[i, a] \leftarrow$ "*reduce* $A \rightarrow \alpha$"
    3. $[S' \rightarrow S\bullet, \$] \in I_i$
       $\Rightarrow$ ACTION$[i, \$] \leftarrow$ "*accept*"
3. $\texttt{goto1}(I_i, A) = I_j$
   $\Rightarrow$ GOTO$[i, A] \leftarrow j$
4. set undefined entries in ACTION and GOTO to "*error*"
5. initial state of parser s_0 is $\texttt{closure1}([S' \rightarrow \bullet S, \$])$

Back to previous example ($\notin$ SLR(1))

$$\begin{array}{lcl} S & \rightarrow & L = R \\ & | & R \\ L & \rightarrow & *R \\ & | & \langle\text{id}\rangle \\ R & \rightarrow & L \end{array}$$

$$\begin{array}{lll}
I_0: & S' \rightarrow \bullet S, & \$ \\
 & S \rightarrow \bullet L = R, & \$ \\
 & S \rightarrow \bullet R, & \$ \\
 & L \rightarrow \bullet * R, & = \\
 & L \rightarrow \bullet\langle\text{id}\rangle, & = \\
 & R \rightarrow \bullet L, & \$ \\
 & L \rightarrow \bullet * R, & \$ \\
 & L \rightarrow \bullet\langle\text{id}\rangle, & \$ \\
I_1: & S' \rightarrow S\bullet, & \$ \\
I_2: & S \rightarrow L\bullet = R, & \$ \\
 & R \rightarrow L\bullet, & \$ \\
I_3: & S \rightarrow R\bullet, & \$ \\
I_4: & L \rightarrow * \bullet R, & = \$ \\
 & R \rightarrow \bullet L, & = \$ \\
 & L \rightarrow \bullet * R, & = \$ \\
 & L \rightarrow \bullet\langle\text{id}\rangle, & = \$
\end{array}$$

$$\begin{array}{lll}
I_5: & L \rightarrow \langle\text{id}\rangle\bullet, & = \$ \\
I_6: & S \rightarrow L = \bullet R, & \$ \\
 & R \rightarrow \bullet L, & \$ \\
 & L \rightarrow \bullet * R, & \$ \\
 & L \rightarrow \bullet\langle\text{id}\rangle, & \$ \\
I_7: & L \rightarrow *R\bullet, & = \$ \\
I_8: & R \rightarrow L\bullet, & = \$ \\
I_9: & S \rightarrow L = R\bullet, & \$ \\
I_{10}: & R \rightarrow L\bullet, & \$ \\
I_{11}: & L \rightarrow * \bullet R, & \$ \\
 & R \rightarrow \bullet L, & \$ \\
 & L \rightarrow \bullet * R, & \$ \\
 & L \rightarrow \bullet\langle\text{id}\rangle, & \$ \\
I_{12}: & L \rightarrow \langle\text{id}\rangle\bullet, & \$ \\
I_{13}: & L \rightarrow *R\bullet, & \$
\end{array}$$

I_2 no longer has shift-reduce conflict: reduce on \$, shift on $=$

Example: back to SLR(1) expression grammar

In general, LR(1) has many more states than LR(0)/SLR(1):

$$
\begin{array}{c|lcl}
1 & S & \to & E \\
2 & E & \to & E + T \\
3 & & | & T
\end{array}
\qquad
\begin{array}{c|lcl}
4 & T & \to & T * F \\
5 & & | & F \\
6 & F & \to & \langle \mathrm{id} \rangle \\
7 & & | & (E)
\end{array}
$$

LR(1) item sets:

I_0 :

$$
\begin{array}{ll}
S \to \bullet E, & \$ \\
E \to \bullet E + T, & + \$ \\
E \to \bullet T, & + \$ \\
T \to \bullet T * F, & * + \$ \\
T \to \bullet F, & * + \$ \\
F \to \bullet \langle \mathrm{id} \rangle, & * + \$ \\
F \to \bullet (E), & * + \$
\end{array}
$$

I_0' : shifting (

$$
\begin{array}{ll}
F \to (\bullet E), & * + \$ \\
E \to \bullet E + T, & +) \\
E \to \bullet T, & +) \\
T \to \bullet T * F, & * +) \\
T \to \bullet F, & * +) \\
F \to \bullet \langle \mathrm{id} \rangle, & * +) \\
F \to \bullet (E), & * +)
\end{array}
$$

I_0'' : shifting (

$$
\begin{array}{ll}
F \to (\bullet E), & * +) \\
E \to \bullet E + T, & +) \\
E \to \bullet T, & +) \\
T \to \bullet T * F, & * +) \\
T \to \bullet F, & * +) \\
F \to \bullet \langle \mathrm{id} \rangle, & * +) \\
F \to \bullet (E), & * +)
\end{array}
$$

Another example

Consider:

$$\begin{array}{c|lcl} 0 & S' & \rightarrow & S \\ 1 & S & \rightarrow & CC \\ 2 & C & \rightarrow & cC \\ 3 & & \mid & d \end{array}$$

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	–	–

LR(1) item sets:

$$\begin{array}{ll} I_0 : S' \rightarrow \bullet S, & \$ \\ \quad S \rightarrow \bullet CC, & \$ \\ \quad C \rightarrow \bullet cC, & cd \\ \quad C \rightarrow \bullet d, & cd \\ I_1 : S' \rightarrow S\bullet, & \$ \\ I_2 : S \rightarrow C \bullet C, & \$ \\ \quad C \rightarrow \bullet cC, & \$ \\ \quad C \rightarrow \bullet d, & \$ \\ I_3 : C \rightarrow c \bullet C, & cd \\ \quad C \rightarrow \bullet cC, & cd \\ \quad C \rightarrow \bullet d, & cd \end{array}$$

$$\begin{array}{ll} I_4 : C \rightarrow d\bullet, & cd \\ I_5 : S \rightarrow CC\bullet, & \$ \\ I_6 : C \rightarrow c \bullet C, & \$ \\ \quad C \rightarrow \bullet cC, & \$ \\ \quad C \rightarrow \bullet d, & \$ \\ I_7 : C \rightarrow d\bullet, & \$ \\ I_8 : C \rightarrow cC\bullet, & cd \\ I_9 : C \rightarrow cC\bullet, & \$ \end{array}$$

LALR(1) parsing

Define the *core* of a set of LR(1) items to be the set of LR(0) items derived by ignoring the lookahead symbols.
Thus, the two sets

- $\{[A \rightarrow \alpha_1 \bullet \alpha_2, \mathtt{a}], [B \rightarrow \beta_1 \bullet \beta_2, \mathtt{b}]\}$, and
- $\{[A \rightarrow \alpha_1 \bullet \alpha_2, \mathtt{c}], [B \rightarrow \beta_1 \bullet \beta_2, \mathtt{d}]\}$

have the same core.
Key idea:

> *If two sets of LR(1) items, I_i and I_j, have the same core, we can merge the states that represent them in the ACTION and GOTO tables.*

LALR(1) table construction

To construct LALR(1) parsing tables, we can insert a single step into the LR(1) algorithm

(1.5) For each core present among the set of LR(1) items, find all sets having that core and replace these sets by their union.
The goto function must be updated to reflect the replacement sets.

The resulting algorithm has large space requirements, as we still are required to build the full set of LR(1) items.

LALR(1) table construction

The revised (*and renumbered*) algorithm

1. construct the collection of sets of LR(1) items for G'
2. for each core present among the set of LR(1) items, find all sets having that core and replace these sets by their union (update the `goto1` function incrementally)
3. state i of the LALR(1) machine is constructed from I_i.
    1. $[A \to \alpha \bullet a\beta, b] \in I_i$ and $\texttt{goto1}(I_i, a) = I_j$ $\Rightarrow$ ACTION$[i,a] \leftarrow$ "*shift* j"
    2. $[A \to \alpha\bullet, a] \in I_i, A \neq S'$ $\Rightarrow$ ACTION$[i,a] \leftarrow$ "*reduce* $A \to \alpha$"
    3. $[S' \to S\bullet, \$] \in I_i \Rightarrow$ ACTION$[i,\$] \leftarrow$ "*accept*"
4. $\texttt{goto1}(I_i, A) = I_j \Rightarrow$ GOTO$[i,A] \leftarrow j$
5. set undefined entries in ACTION and GOTO to "*error*"
6. initial state of parser s_0 is $\texttt{closure1}([S' \to \bullet S, \$])$

Example

Reconsider:

$$\begin{array}{c|ccc} 0 & S' & \to & S \\ 1 & S & \to & CC \\ 2 & C & \to & cC \\ 3 & & \mid & d \end{array}$$

$$\begin{array}{rll} I_0 : & S' \to \bullet S, & \$ \\ & S \to \bullet CC, & \$ \\ & C \to \bullet cC, & cd \\ & C \to \bullet d, & cd \\ I_1 : & S' \to S\bullet, & \$ \\ I_2 : & S \to C \bullet C, & \$ \\ & C \to \bullet cC, & \$ \\ & C \to \bullet d, & \$ \end{array}$$

$$\begin{array}{rll} I_3 : & C \to c \bullet C, & cd \\ & C \to \bullet cC, & cd \\ & C \to \bullet d, & cd \\ I_4 : & C \to d\bullet, & cd \\ I_5 : & S \to CC\bullet, & \$ \end{array}$$

$$\begin{array}{rll} I_6 : & C \to c \bullet C, & \$ \\ & C \to \bullet cC, & \$ \\ & C \to \bullet d, & \$ \\ I_7 : & C \to d\bullet, & \$ \\ I_8 : & C \to cC\bullet, & cd \\ I_9 : & C \to cC\bullet, & \$ \end{array}$$

Merged states:

$$\begin{array}{rll} I_{36} : & C \to c \bullet C, & cd\$ \\ & C \to \bullet cC, & cd\$ \\ & C \to \bullet d, & cd\$ \\ I_{47} : & C \to d\bullet, & cd\$ \\ I_{89} : & C \to cC\bullet, & cd\$ \end{array}$$

state	ACTION			GOTO	
	c	d	$	S	C
0	s36	s47	–	1	2
1	–	–	acc	–	–
2	s36	s47	–	–	5
36	s36	s47	–	–	8
47	r3	r3	r3	–	–
5	–	–	r1	–	–
89	r2	r2	r2	–	–

In the last lecture

- LR(1) parsing and LALR(1) parsing.
  - LR(1): Embed lookahead symbol in the items themselves. Improves upon SLR(1) parsing by allowing more grammars to be parsed.
  - LALR(1): Merge states of LR(1) DFA. Improves upon LR(1) parsing by reducing the time/space complexity of parsing table generation.
- Question: What can we say about the sizes of the SLR(1) parsing table and LALR(1) parsing table?
  - They will be the same.
  - Essentially, LR(1) item-sets with the same core correspond to a unique LR(0) itemset.

Example: LR(1) Itemsets

$$
\begin{array}{lcl}
S & \rightarrow & L = R \\
 & | & R \\
L & \rightarrow & *R \\
 & | & \langle \mathrm{id} \rangle \\
R & \rightarrow & L
\end{array}
$$

$$
\begin{array}{lll}
I_0 : & S' \rightarrow \bullet S, & \$ \\
 & S \rightarrow \bullet L = R, & \$ \\
 & S \rightarrow \bullet R, & \$ \\
 & L \rightarrow \bullet * R, & = \\
 & L \rightarrow \bullet \langle \mathrm{id} \rangle, & = \\
 & R \rightarrow \bullet L, & \$ \\
 & L \rightarrow \bullet * R, & \$ \\
 & L \rightarrow \bullet \langle \mathrm{id} \rangle, & \$ \\
I_1 : & S' \rightarrow S \bullet, & \$ \\
I_2 : & S \rightarrow L \bullet = R, & \$ \\
 & R \rightarrow L \bullet, & \$ \\
I_3 : & S \rightarrow R \bullet, & \$ \\
I_4 : & L \rightarrow * \bullet R, & = \$ \\
 & R \rightarrow \bullet L, & = \$ \\
 & L \rightarrow \bullet * R, & = \$ \\
 & L \rightarrow \bullet \langle \mathrm{id} \rangle, & = \$
\end{array}
$$

$$
\begin{array}{lll}
I_5 : & L \rightarrow \langle \mathrm{id} \rangle \bullet, & = \$ \\
I_6 : & S \rightarrow L = \bullet R, & \$ \\
 & R \rightarrow \bullet L, & \$ \\
 & L \rightarrow \bullet * R, & \$ \\
 & L \rightarrow \bullet \langle \mathrm{id} \rangle, & \$ \\
I_7 : & L \rightarrow * R \bullet, & = \$ \\
I_8 : & R \rightarrow L \bullet, & = \$ \\
I_9 : & S \rightarrow L = R \bullet, & \$ \\
I_{10} : & R \rightarrow L \bullet, & \$ \\
I_{11} : & L \rightarrow * \bullet R, & \$ \\
 & R \rightarrow \bullet L, & \$ \\
 & L \rightarrow \bullet * R, & \$ \\
 & L \rightarrow \bullet \langle \mathrm{id} \rangle, & \$ \\
I_{12} : & L \rightarrow \langle \mathrm{id} \rangle \bullet, & \$ \\
I_{13} : & L \rightarrow * R \bullet, & \$
\end{array}
$$

Example: LALR(1) Itemsets

$$
\begin{array}{rcl}
S & \rightarrow & L = R \\
 & \mid & R \\
L & \rightarrow & *R \\
 & \mid & \langle \mathrm{id} \rangle \\
R & \rightarrow & L
\end{array}
$$

$$
\begin{array}{lll}
I_0 : & S' \rightarrow \bullet S, & \$ \\
 & S \rightarrow \bullet L = R, & \$ \\
 & S \rightarrow \bullet R, & \$ \\
 & L \rightarrow \bullet * R, & = \\
 & L \rightarrow \bullet \langle \mathrm{id} \rangle, & = \\
 & R \rightarrow \bullet L, & \$ \\
 & L \rightarrow \bullet * R, & \$ \\
 & L \rightarrow \bullet \langle \mathrm{id} \rangle, & \$ \\
I_1 : & S' \rightarrow S \bullet, & \$ \\
I_2 : & S \rightarrow L \bullet = R, & \$ \\
 & R \rightarrow L \bullet, & \$ \\
I_3 : & S \rightarrow R \bullet, & \$ \\
I_{4,11} : & L \rightarrow * \bullet R, & = \$ \\
 & R \rightarrow \bullet L, & = \$ \\
 & L \rightarrow \bullet * R, & = \$ \\
 & L \rightarrow \bullet \langle \mathrm{id} \rangle, & = \$
\end{array}
$$

$$
\begin{array}{lll}
I_{5,12} : & L \rightarrow \langle \mathrm{id} \rangle \bullet, & = \$ \\
I_6 : & S \rightarrow L = \bullet R, & \$ \\
 & R \rightarrow \bullet L, & \$ \\
 & L \rightarrow \bullet * R, & \$ \\
 & L \rightarrow \bullet \langle \mathrm{id} \rangle, & \$ \\
I_{7,13} : & L \rightarrow * R \bullet, & = \$ \\
I_{8,10} : & R \rightarrow L \bullet, & = \$ \\
I_9 : & S \rightarrow L = R \bullet, & \$
\end{array}
$$

Has the same number of states as LR(0) DFA of the grammar

LALR(1) Conflicts

Can we always merge states with the same core? Can it create new conflicts?

- Merging LR(1) states with the same core cannot create a new shift/reduce conflict.
  - For contradiction, suppose after merging, the state contains items $[A \rightarrow \alpha \bullet, a]$ and $[B \rightarrow \beta \bullet a\gamma, b]$.
  - Then, one of the original states before merging must have the items $[A \rightarrow \alpha \bullet, a]$ and $[B \rightarrow \beta \bullet a\gamma, c]$, since all original states must have the same core.
  - This indicates a shift-reduce conflict in the original LR(1) state.
- Note that merging LR(1) states can create new reduce/reduce conflicts.
  - Example 4.58 in the Dragon Book.

LALR(1) Conflicts

Can we always merge states with the same core? Can it create new conflicts?

- However, merging LR(1) states can create new reduce/reduce conflicts.
  - For example, consider LR(1) itemsets $\{[A \to \alpha\bullet, a], [B \to \beta\bullet, b]\}$ and $\{[A \to \alpha\bullet, b], [B \to \beta\bullet, a]\}$.
  - After merging, the LALR(1) itemset would be $\{[A \to \alpha\bullet, ab], [B \to \beta\bullet, ab]\}$.
  - There is a reduce/reduce conflict on both a and b.
  - The Dragon Book contains a detailed example illustrating the above scenario (Section 4.7.4, Example 4.58).

More efficient LALR(1) construction

Observe that we can:

- represent I_i by its *basis* or *kernel*:
  items that are either $[S' \rightarrow \bullet S, \$]$
  or do not have $\bullet$ at the left of the RHS
- compute *shift*, *reduce* and *goto* actions for state derived from I_i directly from its kernel

This leads to a method that avoids building the complete canonical collection of sets of LR(1) items

Self reading: Section 4.7.5 Dragon book

Ambiguous Grammars and LR Parsing

Ambiguous grammars are neither LR(k), SLR(k) or LALR(k) for any k.

- In general, we call a grammar LR(k) if there are no conflicts in any of the LR(k) item-sets of the grammar. That is, we can parse any string in the language of the grammar using a LR(k) parser without encountering any conflicts.
- Similar definitions for SLR(k) and LALR(k).

The role of precedence

Precedence and associativity can be used to resolve shift/reduce conflicts in ambiguous grammars.

- lookahead with higher precedence ⇒ *shift*
- same precedence, left associative ⇒ *reduce*

Advantages:

- more concise, albeit ambiguous, grammars
- shallower parse trees ⇒ fewer reductions

Classic application: expression grammars

The role of precedence: Example

With precedence and associativity, we can use:

$$E \to E + E \mid E * E \mid (E) \mid \langle \text{id} \rangle \mid \langle \text{num} \rangle$$

This eliminates useless reductions (*single productions*) but causes shift/reduce conflicts.

- In particular, the LR(0) DFA for this grammar will contain a state with the items $E \to E + E \bullet$, $E \to E \bullet + E$ and $E \to E \bullet * E$.
- This shift/reduce conflict cannot be resolved by SLR(k), LR(k) or LALR(k).
- Since $*$ takes precedence over $+$, shift if the next symbol is $*$.
- For enforcing left-associativity, reduce if the next symbol is $+$.

Error recovery in shift-reduce parsers

The problem

- encounter an error entry in the parsing table for the current state and next symbol
- No shift/reduce action defined

Approaches to Syntax Error Recovery, from simple to complex:

- *Panic Mode*: Discard tokens until a synchronizing token is found
- *Error Productions*: specify in the grammar known common mistakes
- *Automatic local or global correction*: try token insertion or deletions

Parsers typically use a combination of these techniques to handle different kinds of errors.

Panic Mode Recovery

Panic mode error recovery: We want to *parse* the rest of the file
Restarting the parser

- find a restartable state on the stack
- move to a consistent place in the input
- print an informative message to `stderr` (*line number*)

Typically, this involves popping from the stack until a state s with GOTO on non-terminal A is defined KC: where does A come from?. Then, discard input symbols until $a \in$ FOLLOW(A) is found. Resume by pushing GOTO(s, A) on the stack.

A would be non-terminals representing major program pieces, such as expression, statement, block.

Recovery using Error Productions

- Specify in the grammar known common mistakes.
- Essentially, parse and identify errors for smooth recovery.
- Example:
  - Error: The program contains $5x$ instead of $5 * x$.
  - Add the production $E \rightarrow EE$.

Recovery by Automatic Local or Global Correction

- Find a correct ‘nearby’ program by token insertions or deletions.
- Either by exhaustive search or by the context.
- Example
  - For the expression grammar, in the parsing state $E \rightarrow \bullet E + E$, the next token should be a $\langle \mathrm{id} \rangle$.
  - Suppose the next input token is $+$ or $*$.
  - The parser inserts $\langle \mathrm{id} \rangle$ in the input implicitly, by pushing the state $E \rightarrow E \bullet + E$ on the stack.
  - For more details, refer to Example 4.68 in the Dragon Book.

Left versus right recursion

Right Recursion:

- needed for termination in predictive (LL) parsers
- requires more stack space in LR parsers
- right associative operators

Left Recursion:

- works fine in bottom-up (LR) parsers
- limits required stack space
- left associative operators

Rule of thumb:

- right recursion for top-down parsers
- left recursion for bottom-up parsers

Left recursive grammar:

$$\begin{aligned} E \rightarrow &\quad E + T | E \\ T \rightarrow &\quad T * F | F \\ F \rightarrow &\quad (E) \mid Int \end{aligned}$$

After left recursion removal

$$\begin{aligned} E \rightarrow &\quad TE' \\ E' \rightarrow &\quad +TE' | \varepsilon \\ T \rightarrow &\quad FT' \\ T' \rightarrow &\quad *FT' | \varepsilon \\ F \rightarrow &\quad (E) \mid Int \end{aligned}$$

Parse the string 3 + 4 + 5

Parsing review

- *Recursive descent*
  A recursive descent parser directly encodes a grammar into a series of mutually recursive procedures.
- LL(k)
  An LL(k) parser must be able to recognize the use of a production after seeing only the first k symbols of its right hand side.
- LR(k)
  An LR(k) parser must be able to recognize the occurrence of the right hand side of a production after having seen all that is derived from that right hand side with k symbols of lookahead.

Grammar hierarchy

- LR(k+1) $>$ LR(k)
- LR(k) $>$ LALR(k) $>$ SLR(k) $>$ LR(0)
- LL(k+1) $>$ LL(k)
- LR(k) $>$ LL(k)