Associativity

Right recursion produces right associativity.
Treewalk evaluation produces “wrong” sequence.
**Associativity**

Right recursion produces right associativity.
Left recursion results in left associativity.
This is the "natural" associativity.
Left recursion versus right recursion

Right recursion: \( E ::= \text{id} + E \)

- needed for termination in predictive parsers
- requires more stack space
- right associative operators

Left recursion: \( E ::= E + \text{id} \)

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

Rule of thumb:

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

⇒ a simpler expression grammar

\[
<\text{expr}> ::= <\text{expr}> * <\text{expr}>
\]

\[
| <\text{expr}> / <\text{expr}>
\]

\[
| <\text{expr}> + <\text{expr}>
\]

\[
| <\text{expr}> - <\text{expr}>
\]

\[
| ( <\text{expr}> )
\]

\[
| -<\text{expr}>
\]

\[
| \text{id}
\]

\[
| \text{num}
\]
Error recovery in LL(1) parsers

Key notion:

- for each non-terminal, construct a set of terminals on which the parser can synchronize.
- When an error occurs looking for $A$, scan until an element of $\text{SYNCH}(A)$ is found, then pop $A$ and continue.

Building $\text{SYNCH}$:

1. $a \in \text{FOLLOW}(A) \Rightarrow a \in \text{SYNCH}(A)$
2. place keywords that start statements in $\text{SYNCH}(A)$
3. add symbols in $\text{FIRST}(A)$ to $\text{SYNCH}(A)$

If we can’t match a terminal on the top of stack:

1. pop the terminal
2. print a message saying the terminal was inserted
3. continue the parse
Error recovery in shift-reduce parsers

The problem

- encounter an invalid token
- bad pieces of tree hanging from stack
- incorrect entries in symbol table

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 (line number)

For a general discussion, see J.J. Horning’s What the compiler should tell the user in Compiler Construction, An Advanced Course, Springer-Verlag, 1974.
Error recovery in yacc

Yacc’s error mechanism

- designated token error
- valid in any production
- error shows synchronization points

When an error is discovered

- pops the stack until error is legal
- skips input tokens until it matches 3 tokens
- error productions can have actions

This mechanism is fairly general

See §7 of Yacc: yet another compiler-compiler

by Stephen C. Johnson
Error recovery in *yacc*

```
stmt_list : stmt
           | stmt_list ; stmt
```

_can be augmented with error_

```
stmt_list : stmt
           | error
           | stmt_list ; stmt
```

_this should_

- throw out the erroneous statement
- synchronize at ".;" or "end"
- invoke yyerror("syntax error")

Other “natural” places for errors

- all the “lists”
- missing parentheses or brackets
- extra operator or missing operator
$LR(1)$

$SLR(1)$ parsers may not be able to parse some $LR$ grammars.

Problem is that lookahead information is added to $LR(0)$ parser at the end of construction.

We can get more powerful parser by keeping track of lookahead information in the states of the parser.

If, in a single left-to-right scan, we can construct a reverse rightmost derivation, while using at most a single token lookahead to resolve ambiguities, then the grammar is $LR(1)$

Of course, we would like a more formal definition. Unfortunately, that requires some more notation.
**LR(1) grammars**

Given these definitions, we can formally define an LR(1) grammar.

An augmented grammar\(^\dagger\) \(G\) is LR(1) if the three conditions

1. \(\text{Start} \Rightarrow^* \alpha Aw \Rightarrow^* \alpha \beta w\),
2. \(\text{Start} \Rightarrow^* \gamma Bx \Rightarrow^* \alpha \beta y\),
3. \(\text{FIRST}(w) = \text{FIRST}(y)\)

imply that \(\alpha Ay = \gamma Bx\)

(That is, \(\alpha = \gamma\), \(A = B\), and \(x = y\))

To extend this to LR\((k)\) grammars, we define \(\text{FIRST}_k(\alpha)\) as the leading \(k\) symbols that begin strings derived from \(\alpha\)

The definition extends naturally by changing rule 3

\(^\dagger\) 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 \(S' ::= S\)
LR\((k)\) items

The table construction algorithms use LR\((k)\) items to represent the set of possible states in a parse.

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

\(\alpha\) is a production from \(G\) with a \(\bullet\) at some position in the rhs

\(\beta\) is a lookahead string containing \(k\) symbols (terminals or \texttt{eof})

What about LR\((1)\) items?

- example LR\((1)\) item: \([A ::= X \bullet Y Z, a]\)
- LR\((1)\) items have lookahead strings of length 1
- several LR\((1)\) items may have the same core

\([A ::= X \bullet Y Z, a]\)
\([A ::= X \bullet Y Z, b]\)

we represent this as

\([A ::= X \bullet Y Z, \{a, b\}]\)
\textbf{LR(1) lookahead}

What’s the point of all these lookahead symbols?

- carry them along to allow us to choose correct reduction when there is any choice
- lookaheads are bookkeeping, unless item has \( \bullet \) at right end.
  - in \([A ::= X \bullet YZ, a]\), \(a\) has no direct use
  - in \([A ::= X YZ\bullet, a]\), \(a\) is useful
- allows use of grammars that are not \textit{uniquely invertible}\footnote{G is \textit{uniquely invertible} if no two productions have the same \textit{rhs}}

Recall, the \textit{SLR(1)} construction uses \textit{LR(0)} items!

\textit{The point}

For \([A ::= \alpha\bullet, a]\) and \([B ::= \alpha\bullet, b]\), we can decide between reducing to \(A\) and to \(B\) by looking at limited right context!
**Canonical LR(1) items**

The canonical collection of LR(1) items:

- set of items derivable from \([S' ::= \cdot S, \text{eof}]\)

- set of all items that can derive the final configuration

Essentially,

- each set in the canonical collection of sets of LR(1) items represents a state in an NFA that recognizes viable prefixes.

- Grouping together is really the subset construction, §3.6

To construct the canonical collection we need two functions:

- closure\((I)\)

- goto\((I, X)\)
LR(1) closure

Given an item \([A ::= \alpha \bullet B\beta, a]\), 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 \([B ::= \bullet \gamma, b]\) in the closure).

To compute closure(\(I\))

```plaintext
function closure(I)
    repeat
        new_item ← false
        for each item \([A ::= \alpha \bullet B\beta, a] \in I\),
            each production \(B ::= \gamma \in G'\),
            and each terminal \(b \in \text{FIRST(}\beta a\text{)}\),
            if \([B ::= \bullet \gamma, b] \not\in I\) then
                add \([B ::= \bullet \gamma, b]\) to I
                new_item ← true
            endif
    until (new_item = false)
    return I
```

Aho, Sethi, and Ullman, Figure 4.38
**LR(1) goto**

Let $I$ be a set of $LR(1)$ items and $X$ be a grammar symbol.

Then, $\text{goto}(I, X)$ is the closure of the set of all items

$$ [A ::= \alpha X \bullet \beta, a] \text{ such that } [A ::= \alpha \bullet X \beta, a] \in I $$

If $I$ is the set of valid items for some viable prefix $\gamma$, then $\text{goto}(I, X)$ is the set of valid items for the viable prefix $\gamma X$.

$\text{goto}(I, X)$ represents state after recognizing $X$ in state $I$.

To compute $\text{goto}(I, X)$

```
function goto(I, X)
    J ← set of items $[A ::= \alpha X \bullet \beta, a]$
    such that $[A ::= \alpha \bullet X \beta, a] \in I$
    J' ← closure(J)
    return J'
```

Aho, Sethi, and Ullman, Figure 4.38
Collection of sets of \( LR(1) \) items

We start the construction of the collection of sets of \( LR(1) \) items with the item \( [S' ::= \bullet S, \text{eof}] \), where

- \( S' \) is the start symbol of the augmented grammar \( G' \)
- \( S \) is the start symbol of \( G \), and
- \( \text{eof} \) is the right end of string marker

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

\[
\text{procedure items}(G')
\]

\[
C \leftarrow \{\text{closure}([S' ::= \bullet S, \text{eof}])\}
\]

repeat

- new_item \leftarrow false

for each set of items \( I \) in \( C \) and each grammar symbol \( X \) such that
  - \( \text{goto}(I, X) \neq \emptyset \) and
  - \( \text{goto}(I, X) \notin C \)
  - add \( \text{goto}(I, X) \) to \( C \)

new_item \leftarrow true

endfor

until (new_item = false)

Aho, Sethi, and Ullman, Figure 4.38
LR(1) table construction

The Algorithm

1. construct the collection of sets of LR(1) items for $G'$.

2. State $i$ of the parser is constructed from $I_i$.
   (a) if $[A := \alpha \cdot a\beta, b] \in I_i$ and $\text{goto}(I_i, a) = I_j$, then set $\text{action}[i, a]$ to "shift $j". (a must be a terminal)
   (b) if $[A := \alpha \cdot , a] \in I_i$, then set $\text{action}[i, a]$ to "reduce $A := \alpha$".
   (c) if $[S' := S\cdot, \text{eof}] \in I_i$, then set $\text{action}[i, \text{eof}]$ to "accept".

3. If $\text{goto}(I_i, A) = I_j$, then set $\text{goto}[i, A]$ to $j$.

4. All other entries in $\text{action}$ and $\text{goto}$ are set to "error"

5. The initial state of the parser is the state constructed from the set containing the item $[S' := \cdot S, \text{eof}]$.

Aho, Sethi, and Ullman, Algorithm 4.10
Example

The Grammar

1) \( \text{goal} ::= \text{expr} \)
2) \( \text{expr} ::= \text{term} + \text{expr} \)
3) \( \text{term} ::= \text{term} \)
4) \( \text{term} ::= \text{factor} * \text{term} \)
5) \( \text{factor} ::= \text{id} \)
6) \( \text{id} \)

<table>
<thead>
<tr>
<th>( \text{ACTION} )</th>
<th>( \text{GOTO} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{id} + * \text{eof}</td>
<td>\text{expr} \text{term} \text{factor}</td>
</tr>
<tr>
<td>( S_0 ) s4</td>
<td>1 2 3</td>
</tr>
<tr>
<td>( S_1 )</td>
<td>— — —</td>
</tr>
<tr>
<td>( S_2 ) — s5 — r3</td>
<td>— — —</td>
</tr>
<tr>
<td>( S_3 ) — r5 s6 r5</td>
<td>— — —</td>
</tr>
<tr>
<td>( S_4 ) — r6 r6 r6</td>
<td>— — —</td>
</tr>
<tr>
<td>( S_5 ) s4</td>
<td>7 2 3</td>
</tr>
<tr>
<td>( S_6 ) s4</td>
<td>— 8 3</td>
</tr>
<tr>
<td>( S_7 )</td>
<td>— — r2</td>
</tr>
<tr>
<td>( S_8 ) — r4</td>
<td>— — —</td>
</tr>
</tbody>
</table>
Example

Step 1

\[ I_0 \leftarrow \{ [g ::= \bullet e, e\text{of}] \} \]
\[ I_0 \leftarrow \text{closure}(I_0) \]
\[ \{ [g ::= \bullet e, e\text{of}], [e ::= \bullet t + e, e\text{of}], \]
\[ [e ::= \bullet t, e\text{of}], [t ::= \bullet f \ast t, +], \]
\[ [t ::= \bullet f \ast t, e\text{of}], [f ::= \bullet f, +], \]
\[ [t ::= \bullet f, e\text{of}], [f ::= \bullet id, +], \]
\[ [f ::= \bullet id, e\text{of}] \} \]

Iteration 1

\[ I_1 \leftarrow \text{goto}(I_0, e) \]
\[ I_2 \leftarrow \text{goto}(I_0, t) \]
\[ I_3 \leftarrow \text{goto}(I_0, f) \]
\[ I_4 \leftarrow \text{goto}(I_0, \text{id}) \]

Iteration 2

\[ I_5 \leftarrow \text{goto}(I_2, +) \]
\[ I_6 \leftarrow \text{goto}(I_3, \ast) \]

Iteration 3

\[ I_7 \leftarrow \text{goto}(I_5, e) \]
\[ I_8 \leftarrow \text{goto}(I_6, t) \]
Example

$I_0$:  $[g ::= \bullet e, \text{eof}]$,  $[e ::= \bullet t + e, \text{eof}]$,  $[e ::= \bullet t, \text{eof}]$,  $[t ::= \bullet f * t, \{+, \text{eof}\}]$,  $[t ::= \bullet f, \{+, \text{eof}\}]$,  $[f ::= \bullet \text{id}, \{+, \text{eof}\}]$

$I_1$:  $[g ::= e \bullet, \text{eof}]$

$I_2$:  $[e ::= t \bullet, \text{eof}]$,  $[e ::= t \bullet + e, \text{eof}]$

$I_3$:  $[t ::= f \bullet, \{+, \text{eof}\}]$,  $[t ::= f \bullet * t, \{+, \text{eof}\}]$

$I_4$:  $[f ::= \text{id} \bullet, \{+, *, \text{eof}\}]$

$I_5$:  $[e ::= t + \bullet e, \text{eof}]$,  $[e ::= \bullet t + e, \text{eof}]$,  $[e ::= \bullet t, \text{eof}]$,  $[t ::= \bullet f * t, \{+, \text{eof}\}]$,  $[t ::= \bullet f, \{+, \text{eof}\}]$,  $[f ::= \bullet \text{id}, \{+, *, \text{eof}\}]$

$I_6$:  $[t ::= f * \bullet t, \{+, \text{eof}\}]$,  $[t ::= \bullet f * t, \{+, \text{eof}\}]$,  $[t ::= \bullet f, \{+, \text{eof}\}]$,  $[f ::= \bullet \text{id}, \{+, *, \text{eof}\}]$

$I_7$:  $[e ::= t + e \bullet, \text{eof}]$

$I_8$:  $[t ::= f * t \bullet, \{+, \text{eof}\}]$