Bottom-Up Syntax Analysis

Kristoffer H. Rose   Eva Rose
krisrose@cs.nyu.edu

Compiler Construction
CSCI-GA.2130-001/Spring 2014
NYU Courant Institute

May 5, 2014
Outline

1. Parsers (recap)
2. LR(0)
3. LR Parsing
4. LR(1) and LALR(1)
Second compilation phase

source program

Tokens

Lexical Analysis → Syntax Analysis

Tree

Semantic Analysis → Symbol Table

Tree

Intermediate Representation Generator

IR

Optimizer

IR

Code Generator

IR

Machine-Dependent Code Optimizer

target machine code
Outline

1. Parsers (recap)
2. LR(0)
3. LR Parsing
4. LR(1) and LALR(1)
A Left Recursive Grammar

\[ E \rightarrow E + T \mid T \]
\[ T \rightarrow T \ast F \mid F \]
\[ F \rightarrow (E) \mid \text{id} \]  

(4.1)
A Left Recursive Grammar

\[ E \rightarrow E + T \mid T \]
\[ T \rightarrow T \ast F \mid F \]
\[ F \rightarrow (E) \mid \text{id} \]  

(4.1)

Bottom-Up Construction
Parser Categories

L  Left (to right)
R  (use) Rightmost derivation
k  look-ahead count

LR(0)  Easy parsers—include SLR, LALR.
LR(1)  More powerful, less practical:
   ▶ most general non-backtracking method known
   ▶ detects errors early*
   ▶ handles grammar superset of predictive parsers
   ▶ hard to hand-code
   ▶ hard to debug
   ▶ *error recovery problematic

Kristoffer H. Rose  Eva Rose 
krisrose@cs.nyu.edu
Bottom-Up Syntax Analysis  May 5, 2014  6/29
Parser Categories

L  Left (to right)
R  (use) Rightmost derivation
k  look-ahead count

LR(0)  Easy parsers—include SLR, LALR.

LR(1)  More powerful, less practical:
►  most general non-backtracking method known
►  detects errors early*
►  handles grammar superset of predictive parsers
►  hard to hand-code
►  hard to debug
►  *error recovery problematic
Parser Categories

L  Left (to right)
R  (use) Rightmost derivation
k  look-ahead count

LR(0)  Easy parsers—include SLR, LALR.
LR(1)  More powerful, less practical:
   ▶  most general non-backtracking method known
   ▶  detects errors early*
   ▶  handles grammar superset of predictive parsers
      ▶  hard to hand-code
      ▶  hard to debug
      ▶  *error recovery problematic
Parser Categories

L  Left (to right)
R  (use) Rightmost derivation
k  look-ahead count

LR(0)  Easy parsers—include SLR, LALR.
LR(1)  More powerful, less practical:
  ▶  most general non-backtracking method known
  ▶  detects errors early*
  ▶  handles grammar superset of predictive parsers
  ▶  hard to hand-code
  ▶  hard to debug
  ▶  *error recovery problematic
Outline

1. Parsers (recap)
2. LR(0)
3. LR Parsing
4. LR(1) and LALR(1)
LR(k) automaton:

- 2 basic moves: shift token, reduce production
- 2 other moves: accept, error

Concepts:

- An LR(0) item is a rule with a dot at some position: $A \rightarrow .BCD, A \rightarrow B.CD, \ldots$
- An LR(0) state is a set of items.
- Canonical LR(0) collection: FA used to make parsing decisions
**LR(k)**

**LR(k) automaton:**
- 2 basic moves: *shift* token, *reduce* production
- 2 other moves: *accept*, *error*

**Concepts:**
- An *LR(0) item* is a rule with a dot at some position: $A \rightarrow .BCD$, $A \rightarrow B.CD$, ...
- An *LR(0) state* is a set of items.
- *Canonical LR(0) collection*: FA used to make parsing decisions
Closure Construction

For Grammar $G$ need—

- Augmented grammar $G'$
  - Just add rules $S' \rightarrow S$
- closure function for set of items $I$, closure($I$):
  1. add $I$ to closure($I$).
  2. if $A \rightarrow \alpha. B \beta$ in closure($I$) then add $B \rightarrow .\gamma$ (for each possible $\gamma$)
- GOTO function
Example

1. $E' \rightarrow E$
2. $E \rightarrow E + T$
3. $E \rightarrow T$
4. $T \rightarrow T \ast F$
5. $T \rightarrow F$
6. $F \rightarrow (E)$
7. $F \rightarrow \text{id}$
Construction (I)

\[ I = \{ [E' \rightarrow .E] \} \]

\[ I_0 = \text{closure}(I) \]

\[ = \{ [E' \rightarrow .E], [E \rightarrow .E + T], [E \rightarrow .T], [T \rightarrow .T*F], [T \rightarrow .F], [F \rightarrow .(E)], [F \rightarrow .id] \} \]
Construction (II)

For items $I$, $\forall x$ in a set of symbols following dot.

$$GOTO(I, x) = \text{closure}(\{ [A \rightarrow \alpha x.\beta] \mid [A \rightarrow \alpha.s\beta] \in I \})$$

gives

$I_1 = GOTO(I_0, E) = \{ [E' \rightarrow E.], [E' \rightarrow E. + T] \}$

$I_2 = GOTO(I_0, T) = \{ [E \rightarrow T.], [T \rightarrow T. * F] \}$

$I_3 = GOTO(I_0, F) = \{ [T \rightarrow F.] \}$

$I_4 = GOTO(I_0, () = \{ [F \rightarrow (.E)], [E \rightarrow .E + T], [E \rightarrow .T],$

$$[T \rightarrow .T * F], [T \rightarrow .F], [F \rightarrow .(E)], [F \rightarrow .id] \}$

$I_5 = GOTO(I_0, id) = \{ [F \rightarrow id.] \}$
Algorithm

initialize $C = \{ \text{closure}([S' \rightarrow .S]) \}$
repeat for all $I \in C$, for all $x$,
  if $\text{GOTO}(I, x) \neq \emptyset$ and $\text{GOTO}(I, x) \notin C$
    add $\text{GOTO}(I, x)$ to $C$

In practice find repeating items, *i.e.*, $\text{GOTO}(I_4, T) = I_2$
  - A **final** item is one with the . at the end.
  - If all final items are in states by themselves then the grammar is $LR(0)$,
  - otherwise there is a shift-reduce or reduce-reduce conflict in $LR(0)$. 
Use **Follow Sets** to decide when to reduce: $SLR(1)$. If they overlap we can look 2 symbols ahead, *etc.*
Outline

1. Parsers (recap)
2. LR(0)
3. LR Parsing
4. LR(1) and LALR(1)
Automaton

- 2 basic moves: **shift**, **reduce**
- 2 other moves: **accept**, **error**
# Shift-Reduce

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>$id_1</td>
<td>shift $id_2 $</td>
</tr>
<tr>
<td>$id_1</td>
<td>$id_2</td>
<td>reduce by $F \rightarrow id$</td>
</tr>
<tr>
<td>$F</td>
<td>$id_2</td>
<td>reduce by $T \rightarrow F$</td>
</tr>
<tr>
<td>$T</td>
<td>$id_2</td>
<td>shift $id_1 \ast id_2$</td>
</tr>
<tr>
<td>$T \ast$</td>
<td>$id_2$</td>
<td>shift $id_1 \ast id_2$</td>
</tr>
<tr>
<td>$T \ast id_2$</td>
<td>$id_2$</td>
<td>reduce by $F \rightarrow id$</td>
</tr>
<tr>
<td>$T \ast F$</td>
<td>$id_2$</td>
<td>reduce by $T \rightarrow T \ast F$</td>
</tr>
<tr>
<td>$T$</td>
<td>$id_2$</td>
<td>reduce by $E \rightarrow T$</td>
</tr>
<tr>
<td>$E$</td>
<td>$id_2$</td>
<td>accept $id_1 \ast id_2$</td>
</tr>
</tbody>
</table>
LR components

- Input “pointer.”
- Driver.
- Action/GOTO table.
- Stack.
Configuration

Stack  $s_0 \ldots s_m$ (both productions and tokens)
Input  $a_1 \ldots a_n$
Moves...

**shift** \( j \) **consume symbol** \( a_i \), **push** \( s_j \).

![Diagram](image)

**reduce** \( A \rightarrow \beta \) **pop** \(|\beta|\) **states from stack**

**push** \( \text{GOTO}(s_{m-|\beta|}, A) \) **on stack**.

**With** \( A \rightarrow .BCD \):

![Diagram with states](image)
Moves...

shift \( j \) consume symbol \( a_i \), push \( s_j \).

reduce \( A \rightarrow \beta \) pop \( |\beta| \) states from stack
push GOTO\((s_m - |\beta|, A)\) on stack.
With \( A \rightarrow .BCD \):

```
B -> C -> D
```
Example.
Remark

Do you see the bottom-up?
Universal LR Driver

push S0 onto empty stack
read first input token
repeat:
  let s = state on top of stack
  if action(s,a) = shift t:
    push t onto stack
    a := next symbol
  else if action(s,a) = reduce A → β
    pop |β| states off stack
    let t be new top of stack
    push GOTO(t,A) onto stack
    synthesize attributes for A → β
  else if action(s,a) = accept
    break
else
  call error recovery
Outline

1. Parsers (recap)
2. LR(0)
3. LR Parsing
4. LR(1) and LALR(1)
LR(1) Items

- Extend LR(0) items with look-ahead: $[A \rightarrow \alpha \beta, a]$
- Start with $[S' \rightarrow .S, \dollar]$
- Example: $[E' \rightarrow .E, \dollar]$

Closure:
- $[E \rightarrow .E + T, \dollar, +], [E \rightarrow .T, \dollar, +], [T \rightarrow .T^*F, \dollar, +, *], ...$
LR(1) Items

- Extend LR(0) items with **look-ahead**: \([A \to \alpha.\beta, a]\)
- Start with \([S' \to .S, \$]\)
- Example: \([E' \to .E, \$]\)
  closure:
  \([E \to .E+T, \$, +], [E \to .T, \$, +], [T \to .T*F, \$, +, *], \ldots\)
LR(1) Items

- Extend LR(0) items with **look-ahead:** \([A \rightarrow \alpha.\beta, a]\)
- Start with \([S' \rightarrow .S, \$]\)
- Example: \([E' \rightarrow .E, \$]\)

**Closure:**

\[
\begin{align*}
E & \rightarrow .E + T, \$, + \\
E & \rightarrow .T, \$, + \\
T & \rightarrow .T^*F, \$, +, *
\end{align*}
\]
Algorithm (with Lookahead)

closure(I):
  repeat for all \([A\rightarrow\alpha.B\beta,a] \in I\),
    for all \(B\rightarrow\gamma\) in G',
      for all \(b\) in FIRST(\(\beta a\))
        add \([B\rightarrow\gamma,b]\) to I
  until no more items can be added

goto(I,x):
  \(J = \emptyset\)
  for all \([A\rightarrow\alpha.x\beta,a] \in I\)
    add \([A\rightarrow\alpha x.\beta,a]\) to J
  return closure(J)
Remarks

Unlike LR(0) → SLR step no need to deal with ambiguous states of LR(1)

For LALR(1), systematically merge certain LR(1) states ⇒ compact tables

To build LALR from LR(0) consider path that led to it ⇒ efficient table construction
Remarks

- Unlike LR(0)→SLR step no need to deal with ambiguous states of LR(1)
- For LALR(1), systematically *merge* certain LR(1) states ⇒ compact tables
- To build LALR from LR(0) consider *path that led to it* ⇒ efficient table construction
Remarks

- Unlike LR(0) → SLR step no need to deal with ambiguous states of LR(1)
- For LALR(1), systematically *merge* certain LR(1) states ⇒ compact tables
- To build LALR from LR(0) consider *path that led to it* ⇒ efficient table construction
Summary

- LR(k) parsers are also called **bottom-up** parsers in contrast to LL(k) **top-down** parsers.
- LR(0) is efficient, LR(k > 0) is not.
- SLR and LALR improve LR(0) in important ways but remain efficient.
- SLR(k) and LALR(k) are both less than LR(k).
- The main source of conflicts are **shift-reduce** and **reduce-reduce** conflicts; resolving in nonstandard ways allows for (some) ambiguous grammars.
**Summary**

- LR($k$) parsers are also called **bottom-up** parsers in contrast to LL($k$) **top-down** parsers.
- LR(0) is efficient, LR($k > 0$) is not.
- SLR and LALR improve LR(0) in important ways but remain efficient.
- SLR($k$) and LALR($k$) are both less than LR($k$).
- The main source of conflicts are shift-reduce and reduce-reduce conflicts; resolving in nonstandard ways allows for (some) ambiguous grammars.
Summary

- LR($k$) parsers are also called **bottom-up** parsers in contrast to LL($k$) **top-down** parsers.
- LR(0) is efficient, LR($k > 0$) is not.
- SLR and LALR improve LR(0) in important ways but remain efficient.
- SLR($k$) and LALR($k$) are both less than LR($k$).
- The main source of conflicts are shift-reduce and reduce-reduce conflicts; resolving in nonstandard ways allows for (some) ambiguous grammars.
Summary

- LR(k) parsers are also called **bottom-up** parsers in contrast to LL(k) **top-down** parsers.
- LR(0) is efficient, LR(k > 0) is not.
- SLR and LALR improve LR(0) in important ways but remain efficient.
- SLR(k) and LALR(k) are both less than LR(k).
- The main source of conflicts are shift-reduce and reduce-reduce conflicts; resolving in nonstandard ways allows for (some) ambiguous grammars.
Summary

- LR($k$) parsers are also called **bottom-up** parsers in contrast to LL($k$) **top-down** parsers.
- LR(0) is efficient, LR($k > 0$) is not.
- SLR and LALR improve LR(0) in important ways but remain efficient.
- SLR($k$) and LALR($k$) are both less than LR($k$).
- The main source of conflicts are **shift-reduce** and **reduce-reduce** conflicts; resolving in nonstandard ways allows for (some) ambiguous grammars.
Questions?

krisrose@cs.nyu.edu

- HACS did not recover but pr2 is restarted.
- Because of GSAS conflict we swap final and pr3:
  - Final is May 12.
  - pr3 will be due week of May 19 (and reduced).
Questions?

krisrose@cs.nyu.edu

- HACS did not recover but pr2 is restarted.
- Because of GSAS conflict we swap final and pr3:
  - Final is May 12.
  - pr3 will be due week of May 19 (and reduced).