Bottom-Up Parsing Algorithms

Lecture Notes by
Profs. Alex Aiken and George Necula (UCB)

Outline

• More about handles
• Viable prefixes: A building block for recognizing handles
• Computing viable prefixes
• Simple LR parsing (SLR)

Review

• \( \alpha \beta \) is a handle of \( \alpha \beta \omega \) in a rightmost derivation if
  \[ S \Rightarrow^* \alpha X \omega \rightarrow \alpha \beta \omega \]
• Handles always appear at the top of the stack
  - To the right of the rightmost non-terminal.
  - Thus shift and reduce moves are sufficient.

• To implement shift-reduce parsing, we must detect handles.
• But even if there are symbols on the stack that can be reduced, that doesn’t mean there is a handle.

Example: Reduction w/o a Handle

• Recall our favorite grammar:
  \[
  E \rightarrow T \ast E | T \\
  T \rightarrow \text{int} \ast T | \text{int} | (E)
  \]
• Now the step
  \[ T \ast \text{int} + \text{int} \rightarrow \text{int} \ast \text{int} + \text{int} \]
  is not part of any rightmost derivation.
• Thus, int is not a handle of \( \text{int} \ast \text{int} + \text{int} \).

Notes on Handles

• Every handle has some production rhs on top of the stack. But a production rhs on top of the stack is not necessarily a handle.
• In other words, whether a given stack has handle can depend on the contents of the entire stack.
• Unique Handles: If a grammar is unambiguous, then every step of a rightmost derivation has a unique handle.

Viable Prefixes

• It is not obvious how to detect handles.
• At each step the parser sees only the stack, not the entire input.
  \[ \alpha \] is a viable prefix if there is an \( \omega \) such that \( \alpha \omega \) is a state of a shift-reduce parser.
Huh?

- What does this mean? A few things:
  - A viable prefix does not extend past the right end of the handle.
  - It’s a viable prefix because it is a prefix of the handle.
  - A viable prefix can be extended to a sentential form by adding terminals to the right.
  - As long as a parser has viable prefixes on the stack, no parsing error has been detected.

Important Fact #3

Important Fact #3 about bottom-up parsing:

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

- This fact is non-obvious.
- We show how to compute automata that accept viable prefixes.

Items

- An item is a production with a "." somewhere on the rhs.
- The items for \( T \rightarrow (E) \) are:
  - \( T \rightarrow (E) \)
  - \( T \rightarrow (E.) \)
  - \( T \rightarrow (E.) \)
- The only item for \( X \rightarrow \varepsilon \) is \( X \rightarrow . \).
- Items are often called "LR(0) items".

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.

Example

Consider the input (int)

- Then \( (E) \) is a state of a shift-reduce parse.
- \( (E) \) is a prefix of the rhs of \( T \rightarrow (E) \)
  - Will be reduced after the next shift.
- Item \( T \rightarrow (E.) \) says that so far we have seen \( E \) of this production and hope to see .

Generalization

- The stack may have many prefixes of rhs's:
  \( \text{Prefix}_1, \text{Prefix}_2, \ldots, \text{Prefix}_{n-1}, \text{Prefix}_n \)
- Let \( \text{Prefix}_i \) be a prefix of rhs of \( X_i \rightarrow \alpha_i \)
  - Now \( \text{Prefix}_i \) is a prefix of \( \alpha_i \)
  - Which eventually reduces to \( X_i \)
  - Which should be a prefix of the missing part of \( \alpha_{n+1} \)
- Recursively, \( \text{Prefix}_{n+1}, \text{Prefix}_n, \ldots, \text{Prefix}_1 \) eventually reduces to the missing part of \( \alpha_i \).
An Example

Consider the string (int * int):

(int * int) is a state of a shift-reduce parse.

"(" is a prefix of the rhs of T → (E)

"ε" is a prefix of the rhs of E → T

"int *" is a prefix of the rhs of T → int * T

An Example (Cont.)

The "stack of items"

T → (.E)
E → .T
T → int *.T

Says

We've seen "(" of T → (E)
We've seen ε of E → T
We've seen int * of T → int * T

Recognizing Viable Prefixes

Idea: To recognize viable prefixes, we must

- Recognize a sequence of partial rhs's of productions, where
- Each sequence can eventually reduce to part of the missing suffix of its predecessor.

NFA Recognizing Viable Prefixes

1. Add a dummy production S' → S to G
2. The NFA states are the items of G
   - Including the extra production
3. For item E → α.Xβ add transition
   E → α.Xβ ® X
4. For item E → α.Xβ and production X → γ add
   E → α.Xβ ® X → γ
5. Every state is an accepting state
6. Start state is S' → S

Translation to the DFA

CS780(Prasad) L134ALG 13
CS780(Prasad) L134ALG 14
CS780(Prasad) L134ALG 15
CS780(Prasad) L134ALG 16
CS780(Prasad) L134ALG 17
CS780(Prasad) L134ALG 18
Valid Items

Item $X \rightarrow \beta \gamma$ is valid for a viable prefix $\alpha \beta \omega$ if

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

by a right-most derivation.

**Intuition:** The valid items are the prefixes of productions we might see after $\alpha$

**Items Valid for a Prefix**

An item $I$ is valid for a viable prefix $\alpha$ if the DFA recognizing viable prefixes terminates on input $\alpha$ in a state containing $I$.

Valid Items Example

- An item is often valid for many prefixes.
- **Example:** The item $T \rightarrow (E)$ is valid for prefixes

  \[
  \begin{aligned}
  ( & ) & \\
  ( & ) & \\
  ( & ) & \\
  \ldots
  \end{aligned}
  \]

Lingo

The states of the DFA are

- “canonical collections of items”
- “canonical collections of LR(0) items”

(There are other ways of constructing LR(0) items.)

SLR Parsing

- LR = “Left-to-right scan”
- SLR = “Simple LR”

**Idea:** Assume

- stack contains $\alpha \beta$
- next input is $t$
- DFA on input $\alpha \beta$ terminates in state $s$

SLR Moves

- **Reduce** by $X \rightarrow \beta$ if
  - $s$ contains item $X \rightarrow \beta$
  - $t \in \text{Follow}(X)$
- **Shift** if
  - $s$ has a transition labeled $t$

If there are conflicts under these rules, the grammar is not SLR.

The rules amount to a heuristic for detecting handles.

The SLR grammars are those where the heuristics detect exactly the handles.
Naïve SLR Parsing Algorithm

1. Let $M$ be DFA for viable prefixes of $G$.
2. Let $[x_1...x_n]$ be the initial configuration.
3. Repeat until configuration is $S|$$
   \hspace{1cm}$
   • Let $\alpha | \omega$ be current configuration.
   \hspace{1cm}
   • Run $M$ on current stack $\alpha$.
   \hspace{1cm}
   • If $M$ rejects $\alpha$, report parsing error.
   \hspace{1cm}
   • Stack $\alpha$ is not a viable prefix.
   \hspace{1cm}
   • If $M$ accepts $\alpha$ with items $I$, let $a$ be next input.
   \hspace{1cm}
   • Shift if $X \rightarrow \beta. a \gamma \in I$.
   \hspace{1cm}
   • Reduce if $X \rightarrow \beta. \in I$ and $a \in \text{Follow}(\alpha)$.
   \hspace{1cm}
   • Report parsing error if neither applies.

Notes

• If there is a conflict in the last step, grammar is not SLR($k$).
• $k$ is the amount of lookahead.
  - In practice, $k = 1$.

SLR Example

<table>
<thead>
<tr>
<th>Configuration</th>
<th>DFA</th>
<th>Halt State</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{int} \star \text{int}$</td>
<td>1</td>
<td>shift</td>
<td></td>
</tr>
</tbody>
</table>

SLR Example

<table>
<thead>
<tr>
<th>Configuration</th>
<th>DFA</th>
<th>Halt State</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{int} \star \text{int}$</td>
<td>1</td>
<td>shift</td>
<td></td>
</tr>
<tr>
<td>$\text{int} \star \text{int}$</td>
<td>3</td>
<td>not in Follow($T$)</td>
<td>shift</td>
</tr>
</tbody>
</table>
### SLR Example

**Configuration DFA Halt State**

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>int * int$ 1</td>
<td>shift</td>
</tr>
<tr>
<td>int * int$ 3</td>
<td>* not in Follow(T) shift</td>
</tr>
<tr>
<td>int * int$ 11</td>
<td>shift</td>
</tr>
<tr>
<td>int * int</td>
<td>$ 3 ∈ Follow(T) red. T→int</td>
</tr>
<tr>
<td>T</td>
<td>$ 5 ∈ Follow(T) red. E→T</td>
</tr>
</tbody>
</table>
SLR Example

Configuration  DFA Halt State  Action

| int * int$ | 1 | shift |
| int | * int$ | 3 | not in Follow(T) | shift |
| int * | int$ | 11 | shift |
| int * int |$ | 3 | ∈ Follow(T) | red. T→int |
| int * T |$ | 4 | ∈ Follow(T) | red. T→int*T |
| T |$ | 5 | ∈ Follow(T) | red. E→T |
| E |$ | accept |

Notes

• Skipped using extra start state S' in this example to save space on slides.
• Rerunning the automaton at each step is wasteful
  - Most of the work is repeated.

An Improvement

• Remember the state of the automaton on each prefix of the stack.
• Change stack to contain pairs
  ( Symbol, DFA State )

An Improvement (Cont.)

• For a stack
  ( sym_1, state_1 ) ... ( sym_n, state_n )
  state_n is the final state of the DFA on sym_1 ... sym_n

• Detail: The bottom of the stack is ( any, start )
  where
  - any is any dummy symbol
  - start is the start state of the DFA

Goto Table

• Define Goto[i,A] = j if state_i →\^ state_j

• Goto is just the transition function of the DFA
  - One of two parsing tables.

Action Table

For each state s_i and terminal a

• If s_i has item X → α.a$ and Goto[i,a] = j then
  action[i,a] = shift j

• If s_i has item X → α, and a ∈ Follow(X) and X = S
  then action[i,a] = reduce X → α

• If s_i has item S' → S, then action[i,\$] = accept

• Otherwise, action[i,a] = error
SLR Parsing Algorithm

Let I = w$ be initial input
Let j = 0
Let DFA state I have item S' → .S
Let stack = (dummy, 1)
repeat
  case action[top_state(stack), I[j]] of
    shift k: push (I[j+1], k)
    reduce X → A:
      pop |A| pairs,
      push (X, Goto[X, top_state(stack)])
    accept: halt normally
    error: halt and report error

Notes on SLR Parsing Algorithm

• Note that the algorithm uses only the DFA states and the input
  - The stack symbols are never used!

• However, we still need the symbols for semantic actions.