Lecture 04 – Syntax analysis: top-down and bottom-up parsing

THEORY OF COMPIILATION

Eran Yahav

You are here

Compiler

Source text

Lexical Analysis

Syntax Analysis

Semantic Analysis

Inter. Rep. (IR)

Code Gen.

Executable code

Last week: from tokens to AST

You are here

Example

G = (V, T, P, S)

- V – non terminals
- T – terminals (tokens)
- P – derivation rules
- S – initial symbol

Example

S → S; S
S → id := E
E → id | E + E | E * E | ( E )
Last week: parsing

- A context free language can be recognized by a non-deterministic pushdown automaton
- Parsing can be seen as a search problem
  - Can you find a derivation from the start symbol to the input word?
  - Easy (but very expensive) to solve with backtracking
- We want efficient parsers
  - Linear in input size
  - Deterministic pushdown automata
  - We will sacrifice generality for efficiency

Chomsky Hierarchy

Grammar Hierarchy

LL(k) Parsers

- Manually constructed
  - Recursive Descent
- Generated
  - Uses a pushdown automaton
  - Does not use recursion
LL(k) parsing with pushdown automata

- Pushdown automaton uses
  - Prediction stack
  - Input stream
  - Transition table
    - nonterminals \times \text{tokens} \rightarrow \text{production alternative}
    - Entry indexed by nonterminal N and token t contains the alternative of N that must be predicated when current input starts with t

Two possible moves
- Prediction
  - When top of stack is nonterminal N, pop N, lookup table[N,t]. If table[N,t] is not empty, push table[N,t] on prediction stack, otherwise – syntax error
- Match
  - When top of prediction stack is a terminal T, must be equal to next input token t. If (t \equiv T\equiv T), pop T and consume t. If (t \not\equiv T\not\equiv T) syntax error

Parsing terminates when prediction stack is empty. If input is empty at that point, success. Otherwise, syntax error

Example transition table

<table>
<thead>
<tr>
<th>Nonterminals</th>
<th>not</th>
<th>true</th>
<th>false</th>
<th>and</th>
<th>or</th>
<th>xor</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIT</td>
<td></td>
<td>4</td>
<td>5</td>
<td></td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>OP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Simple Example

\[ \text{Input suffix} \quad \text{Stack content} \quad \text{Move} \]

\[ \text{aabbs} \quad A \to \text{aab} \mid c \]

- \[ \text{aabbs} \quad \text{aab} \quad \text{predict}(A,a) = A \to \text{aab} \]
- \[ \text{aabbs} \quad \text{aab} \quad \text{match}(a,a) \]
- \[ \text{aabbs} \quad \text{aab} \quad \text{predict}(A,a) = A \to \text{aab} \]
- \[ \text{aabbs} \quad \text{aab} \quad \text{match}(a,a) \]
- \[ \text{aabbs} \quad \text{aab} \quad \text{predict}(A,b) = \text{ERROR} \]
- \[ \text{aabbs} \quad \text{aab} \quad \text{match}(a,b) \]
- \[ \text{aabbs} \quad \text{aab} \quad \text{match}(b,b) \]
- \[ \text{aabbs} \quad \text{aab} \quad \text{match}(c,b) \]
- \[ \text{aabbs} \quad \text{aab} \quad \text{match}(s,s) \to \text{success} \]

Error Handling and Recovery

\[ x = a \ast (p+q \ast (-b \ast (r-s))) \]

- Where should we report the error?
- The valid prefix property
- Recovery is tricky
  - Heuristics for dropping tokens, skipping to semicolon, etc.
Error Handling in LL Parsers

\[ S \rightarrow a \mid b \]

<table>
<thead>
<tr>
<th>Input suffix</th>
<th>Stack content</th>
<th>Move</th>
</tr>
</thead>
<tbody>
<tr>
<td>bcs</td>
<td>$S$</td>
<td></td>
</tr>
<tr>
<td>bcs</td>
<td>$bS$</td>
<td></td>
</tr>
<tr>
<td>cs</td>
<td>$S$</td>
<td></td>
</tr>
</tbody>
</table>

- Result: infinite loop

```
 a  b  c
S  S->a  S->bS
```

Error Handling

- Requires more systematic treatment
- Enrichment
  - Acceptable-set method
  - Not part of course material

Summary so far

- Parsing
  - Top-down or bottom-up
- Top-down parsing
  - Recursive descent
  - LL(k) grammars
  - LL(k) parsing with pushdown automata
- LL(K) parsers
  - Cannot deal with left recursion
  - Left-recursion removal might result with complicated grammar

Bottom-up Parsing

- LR(K)
- SLR
- LALR

- All follow the same pushdown-based algorithm
- Differ on type of “LR Items”
LR Item

Input

Already matched

To be matched

\[ N \to \alpha \beta \]

Hypothesis about \( \alpha \beta \) being a possible handle, so far we've matched \( \alpha \), expecting to see \( \beta \)

LR Items

\[ N \to \alpha \beta \]
Shift Item

\[ N \to \alpha \beta \]
Reduce Item

Example

\[ Z \to \text{expr} \ E \ $ \]
\[ \text{expr} \to \text{term} \mid \text{expr} + \text{term} \]
\[ \text{term} \to \text{ID} \mid ( \text{expr} ) \]

Example:Parsing with LR Items

\[ Z \to \text{E} \ $ \]
\[ E \to \text{T} \mid E + T \]
\[ T \to \text{i} \mid ( \text{E} ) \]

\[ Z \to \bullet \ E \ $ \]
\[ E \to \bullet \ T \]
\[ \text{E} \to \bullet \ E + T \]
\[ T \to \bullet \ i \]
\[ T \to \bullet ( \text{E} ) \]

Why do we need these additional LR items?
Where do they come from?
What do they mean?

(just shorthand of the grammar on the top)
\( \varepsilon \)-closure

- Given a set \( S \) of LR(0) items
- If \( P \rightarrow \varepsilon \cdot N \) is in \( S \)
- then for each rule \( N \rightarrow \gamma \) in the grammar \( S \) must also contain \( N \rightarrow \varepsilon \gamma \)

\[ \varepsilon \text{-closure} \left( \{ Z \rightarrow \varepsilon \cdot E \} \right) = \{ E \rightarrow \varepsilon, E \rightarrow E + T, T \rightarrow \varepsilon, T \rightarrow \varepsilon (E) \} \]

Example: Parsing with LR Items

- Parse tree showing production rules for parsing with LR items.
- Reduce item! steps shown.

Example: Parsing with LR Items

- Parse tree showing production rules for parsing with LR items.
- Reduce item! steps shown.
Example: Parsing with LR Items

Z → E $  
Z → T | E + T  
T → i | (E)  

Z → E*$  
E → iT  
E → E + T  
T → i  
T → *(E)  

Example: Parsing with LR Items

Z → E $  
Z → T | E + T  
T → i | (E)  

Z → E*$  
E → iT  
E → E + T  
T → i  
T → *(E)  

Reduce item!
Example: Parsing with LR Items

Computing Item Sets

- Initial set
  - $Z$ is in the start symbol
  - $\varepsilon$-closure($\{Z \rightarrow \alpha \mid Z \rightarrow \alpha$ is in the grammar $\}$)

- Next set from a set $S$ and the next symbol $X$
  - $\text{step}(S,X) = \{ N \rightarrow \alpha X \beta \mid N \rightarrow \alpha \beta \}$ in the item set $S$
  - $\text{nextSet}(S,X) = \varepsilon$-closure($\text{step}(S,X)$)
LR(0) Automaton Example

LR Pushdown Automaton
- Two moves: shift and reduce
- Shift move
  - Remove first token from input
  - Push it on the stack
  - Compute next state based on GOTO table
  - Push new state on the stack
  - If new state is error – report error

GOTO/ACTION Tables

LR Pushdown Automaton $\Rightarrow$ $\Rightarrow$
- Reduce move
  - Using a rule $N \rightarrow \alpha$
  - Symbols in $\alpha$ and their following states are removed from stack
  - New state computed based on GOTO table (using top of stack, before pushing $N$)
  - $N$ is pushed on the stack
  - New state pushed on top of $N$
Are we done?

- Can make a transition diagram for any grammar
- Can make a GOTO table for every grammar
- Cannot make a deterministic ACTION table for every grammar
LR(0) Conflicts

- Any grammar with an $\varepsilon$-rule cannot be LR(0)
- Inherent shift/reduce conflict
  - $A \rightarrow \varepsilon$ – reduce item
  - $P \rightarrow \alpha\varepsilon\beta$ – shift item
  - $A \rightarrow \varepsilon$ can always be predicted from $P \rightarrow \alpha\beta$

SRL Grammars

- A handle should not be reduced to a non-terminal $N$ if the look-ahead is a token that cannot follow $N$
- A reduce item $N \rightarrow \alpha$ is applicable only when the look-ahead is in FOLLOW($N$)
- Differs from LR(0) only on the ACTION table

Back to the GOTO/ACTIONS tables

ACTION table determined only by transition diagram, ignores input
### SLR ACTION Table

<table>
<thead>
<tr>
<th>State</th>
<th>i</th>
<th>e</th>
<th>(</th>
<th>)</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>q0</td>
<td>shift</td>
<td>shift</td>
<td>shift</td>
<td></td>
<td></td>
</tr>
<tr>
<td>q1</td>
<td>shift</td>
<td></td>
<td></td>
<td>shift</td>
<td></td>
</tr>
<tr>
<td>q2</td>
<td></td>
<td></td>
<td>Z→E $</td>
<td></td>
<td></td>
</tr>
<tr>
<td>q3</td>
<td>shift</td>
<td>shift</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>q4</td>
<td>E→E+T</td>
<td>E→E+T</td>
<td>E→E+T</td>
<td></td>
<td></td>
</tr>
<tr>
<td>q5</td>
<td>T→i</td>
<td>T→i</td>
<td>T→i</td>
<td></td>
<td></td>
</tr>
<tr>
<td>q6</td>
<td>E→T</td>
<td>E→T</td>
<td>E→T</td>
<td></td>
<td></td>
</tr>
<tr>
<td>q7</td>
<td>shift</td>
<td>shift</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>q8</td>
<td>shift</td>
<td>shift</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>q9</td>
<td>T→(E)</td>
<td>T→(E)</td>
<td>T→(E)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note: In contrast, GOTO table is indexed by state and a grammar symbol from the stack.

### Look-ahead token from the input

Are we done?

Are we done?

- (0) S' → S
- (1) S → L * R
- (2) S → R
- (3) L → * R
- (4) L → id
- (5) R → L

### SLR ACTION Table

<table>
<thead>
<tr>
<th>State</th>
<th>i</th>
<th>e</th>
<th>(</th>
<th>)</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>q0</td>
<td>shift</td>
<td>shift</td>
<td>shift</td>
<td></td>
<td></td>
</tr>
<tr>
<td>q1</td>
<td>shift</td>
<td></td>
<td></td>
<td>shift</td>
<td></td>
</tr>
<tr>
<td>q2</td>
<td></td>
<td>Z→E $</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>q3</td>
<td>shift</td>
<td>shift</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>q4</td>
<td>E→E+T</td>
<td>E→E+T</td>
<td>E→E+T</td>
<td></td>
<td></td>
</tr>
<tr>
<td>q5</td>
<td>T→i</td>
<td>T→i</td>
<td>T→i</td>
<td>shift</td>
<td>T→i</td>
</tr>
<tr>
<td>q6</td>
<td>E→T</td>
<td>E→T</td>
<td>E→T</td>
<td></td>
<td></td>
</tr>
<tr>
<td>q7</td>
<td>shift</td>
<td>shift</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>q8</td>
<td>shift</td>
<td>shift</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>q9</td>
<td>T→(E)</td>
<td>T→(E)</td>
<td>T→(E)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Remember: GOTO table is indexed by state and a grammar symbol from the stack.

### SLR – use 1 token look-ahead

### LR(0) – no look-ahead

Are we done?
Shift/reduce conflict

- $S \rightarrow L \bullet R$ vs. $R \rightarrow L \bullet$
- FOLLOW($R$) contains $=$
- $S \Rightarrow L \Rightarrow R \Rightarrow R \Rightarrow L \Rightarrow R$
- SLR cannot resolve the conflict either

LR(1) Grammars

- In SLR: a reduce item $N \rightarrow \alpha \bullet$ is applicable only when the look-ahead is in FOLLOW($N$)
- But FOLLOW($N$) merges look-ahead for all alternatives for $N$
- LR(1) keeps look-ahead with each LR item
- Idea: a more refined notion of follows computed per item

LR(1) Item

- LR($1$) item is a pair
  - LR($0$) item
  - Look-ahead token
- Meaning
  - We matched the part left of the dot, looking to match the part on the right of the dot, followed by the look-ahead token.
- Example
  - The production $L \rightarrow \text{id}$ yields the following LR($1$) items

$\epsilon$-closure for LR(1)

- For every $[A \rightarrow \alpha \bullet B \beta \cdot c]$ in $S$
  - for every production $B \rightarrow \delta$ and every token $b$ in the grammar such that $b \in \text{FIRST}(\beta c)$
  - Add $[B \rightarrow \bullet \delta, b]$ to $S$
**LALR**

- LR tables have large number of entries
- Often don’t need such refined observation (and cost)
- LALR idea: find states with the same LR(0) component and merge their look-ahead component as long as there are no conflicts
- LALR not as powerful as LR(1)

**Summary**

- Bottom up
  - LR Items
  - LR parsing with pushdown automata
  - LR(0), SLR, LR(1) – different kinds of LR items, same basic algorithm
Next time

- Semantic analysis