Chapter 4 S.A. II (Bottom-up Parsing)

I. Introduction

II. Operator Precedence Parser

III. LR Parsers

(a) Simple LR parser (SLR)
(b) Canonical LR parser (CLR)
(c) Look ahead LR parser (LALR)

IV. Error handling

V. Symbol table

I. Introduction

Top-down parsing problem: we have to change the grammar to remove left recursion. Solution: use bottom-up parser.

How? Example: Expression grammar. (begin from bottom terminal symbols to look for starting symbol).

1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow id \mid (E)$

String: $a + b$ using reduction.

Example 2 $a + b * c$

Q: how does the parser know which reduction to take?

A: use handle.

String: $a + b * c$

1. Do RMD

   $E \rightarrow E + T$
   $\quad \rightarrow E + T + F$
   $\quad \rightarrow E + T * id$
   $\quad \rightarrow E + F * id \rightarrow E + id * id \rightarrow T + id * id \rightarrow F + id * id \rightarrow id + id * id$
2. Construct the tree from the bottom up of the right-most-derivation.

Handle is a RHS of a production which reduced to get the preceding step of RMD.
Formal definition: For $\beta$ to be a handle of the sentential form $\alpha\beta\omega$, we must have:
1. $\alpha A\omega \rightarrow \alpha\beta\omega \ (A\rightarrow\beta)$
2. $S\rightarrow\alpha A\omega$ with RMD.

II. Operator Precedence Parser
- Works for just simple grammar (such as expressions)
- How handles work
- Table driven

(i) Parsing Table
   Example:
   1. $E\rightarrow E + E$
   2. $E\rightarrow E * E$
   3. $E\rightarrow id$

<table>
<thead>
<tr>
<th>Stack Symbol</th>
<th>Input Symbol</th>
<th>+</th>
<th>*</th>
<th>id</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>+</td>
<td>&gt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&gt;</td>
</tr>
<tr>
<td></td>
<td>*</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&lt;</td>
<td>&gt;</td>
</tr>
<tr>
<td>id</td>
<td></td>
<td>&gt;</td>
<td>&gt;</td>
<td></td>
<td>&gt;</td>
</tr>
<tr>
<td>$</td>
<td></td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td></td>
</tr>
</tbody>
</table>

(ii) Driver:
- Push $\$ onto the stack
- append $\$ at the end of the string
repeat
   - let $t = \text{tos terminal symbol and } x = \text{input symbol}$
   - Find Table[$t$, $x$]
   - if no entry then error
   - else if $t < x$ then push $<$ and $x$ onto the stack
   - else begin /* we have a handle */
     - reduce the handle delimited by $<$ and $>$
     - if no RHS matches then error
     - else push the LHS onto the stack
   end
until (($t = \$ and $x = \$) or error)
(iii) Example: $a + b * c$

<table>
<thead>
<tr>
<th>Stack</th>
<th>Comparison</th>
<th>Input</th>
<th>Production Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td></td>
<td>$a + b * c$</td>
<td></td>
</tr>
<tr>
<td>$&lt;a$</td>
<td>$&gt;$</td>
<td>$b * c$</td>
<td>3. $E \to id$</td>
</tr>
<tr>
<td>$E$</td>
<td>$&lt;$</td>
<td>$b * c$</td>
<td></td>
</tr>
<tr>
<td>$&lt;E+$</td>
<td>$&lt;$</td>
<td>$b * c$</td>
<td></td>
</tr>
<tr>
<td>$&lt;E+&lt;b$</td>
<td>$&gt;$</td>
<td>$b * c$</td>
<td>3. $E \to id$</td>
</tr>
<tr>
<td>$&lt;E+E$</td>
<td>$&lt;$</td>
<td>$b * c$</td>
<td></td>
</tr>
<tr>
<td>$&lt;E+E*$</td>
<td>$&lt;$</td>
<td>$b * c$</td>
<td></td>
</tr>
<tr>
<td>$&lt;E+E*&lt;c$</td>
<td>$&gt;$</td>
<td>$&lt;E+E*$</td>
<td>2. $E \to E\cdot E$</td>
</tr>
<tr>
<td>$&lt;E+E*E$</td>
<td>$&gt;$</td>
<td>$&lt;E+E*$</td>
<td>1. $E \to E+E$</td>
</tr>
<tr>
<td>$E$</td>
<td></td>
<td>$&lt;$</td>
<td>3. $E \to id$</td>
</tr>
<tr>
<td>$&lt;E+E*&lt;c$</td>
<td>$&gt;$</td>
<td>$&lt;E+E*&lt;c$</td>
<td></td>
</tr>
<tr>
<td>$&lt;E+E*E$</td>
<td>$&gt;$</td>
<td>$&lt;E+E*E$</td>
<td></td>
</tr>
<tr>
<td>$E$</td>
<td></td>
<td>$&lt;$</td>
<td></td>
</tr>
<tr>
<td>$&lt;E+E*E$</td>
<td>$&gt;$</td>
<td>$&lt;E+E*E$</td>
<td></td>
</tr>
<tr>
<td>$E$</td>
<td></td>
<td>$&lt;$</td>
<td></td>
</tr>
<tr>
<td>$&lt;E+E*$</td>
<td>$&gt;$</td>
<td>$&lt;E+E*$</td>
<td></td>
</tr>
<tr>
<td>$E$</td>
<td></td>
<td>$&lt;$</td>
<td></td>
</tr>
<tr>
<td>$&lt;E+E$</td>
<td>$&gt;$</td>
<td>$&lt;E+E$</td>
<td></td>
</tr>
<tr>
<td>$E$</td>
<td></td>
<td>$&lt;$</td>
<td></td>
</tr>
<tr>
<td>$&lt;E+$</td>
<td>$&lt;$</td>
<td>$&lt;E+$</td>
<td></td>
</tr>
<tr>
<td>$E$</td>
<td></td>
<td>$&lt;$</td>
<td></td>
</tr>
<tr>
<td>$E$</td>
<td></td>
<td>$&lt;$</td>
<td>Finished.</td>
</tr>
</tbody>
</table>

III. LR parsers (Left-to-right-scan; Right-most-derivation)
- SLR (Simple LR)
- CLR (Canonical LR)
- LALR (Look Ahead LR)

(a) SLR
Still table driven.
(i) Introduction to the table.
(See the handout)
(ii) Driver
- Place $ at the end of the string
- Push state 0 onto the stack
- Repeat
  - let $q_m$ be the current state (Top of stack) and i be the token.
  - let $x = Table[q_m, i]$;
  - case $x$ of
    - $S_q_n$ : shift (i) onto the stack and enter to the state $q_n$ (shift $q_n$)
    - $R_r$ : reduce by production $n$ by popping $2*$ (the number of RHS symbols).
      - let $q_i$ be the top-of-stack state
      - push LHS N onto the stack
      - push $q_k = Table[q_i, N]$;
    ACCT : Finished.
    Empty : Error
- Until (Finished or Error)
Example: a+b

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Rule/Action</th>
<th>Production used</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a+b$</td>
<td>S5</td>
<td></td>
</tr>
<tr>
<td>0a5</td>
<td>+b$</td>
<td>R6</td>
<td>6. F→id</td>
</tr>
<tr>
<td>0F3</td>
<td>+b$</td>
<td>R4</td>
<td>4. T→F</td>
</tr>
<tr>
<td>0T2</td>
<td>+b$</td>
<td>R2</td>
<td>2. E→T</td>
</tr>
<tr>
<td>0E1</td>
<td>+b$</td>
<td>S6</td>
<td></td>
</tr>
<tr>
<td>0E1+6</td>
<td>b$</td>
<td>S5</td>
<td></td>
</tr>
<tr>
<td>0E1+6b5</td>
<td>$</td>
<td>R6</td>
<td>6. F→id</td>
</tr>
<tr>
<td>0E1+6F3</td>
<td>$</td>
<td>R4</td>
<td>4. T→F</td>
</tr>
<tr>
<td>0E1+6T9</td>
<td>$</td>
<td>R1</td>
<td>1. E→E+T</td>
</tr>
<tr>
<td>0E1</td>
<td></td>
<td>Acct</td>
<td></td>
</tr>
</tbody>
</table>

To create a SLR parsing table, we need to introduce three new concepts:

(1) Item:
An item is a production with a place marker “.” inserted somewhere in its RHS in “[”, “]”. Example:
[E→E+T] (initial item), [E→E+T], [E→E+T], [E→E+T] (completed item).
Given a production rule, you can create a set of items.

(2) Closure(I)
If I is a set of items, then the Closure(I) is given by:
(i) $I \in \text{Closure}(I) \rightarrow \text{basic set}.$
(ii) if $[A→\alpha.B\beta]$ is in the Closure(I), and $B→\gamma$ is a production, then, $[B→\gamma]$ is in Closure(I).
Example:
Closure(I = \{[E→E+T]\}) = \{[E→E+T], [E→E+T], [T→T*F], [T→F], [F→id], [F→(E)]\}.
Closure(I = \{[E→E+T]\}) = \{[E→E+T]\}.

(3) Transition N(I, x)
If I is a set of items, and x is any grammar symbol (terminal or non-terminal), then N(I, x) is defined to be the closure of all items $[A→\alpha.x.B\beta]$ such that $[A→\alpha.x.B\beta]$ is in I.
Example:
N(I=\{[E→E+T], [E→E+T]\}, x = E) = \{[E→E+T]\}.
N(I=\{[E→E+T], [F→id]\}, x = *) = \{ \}.

* Build the Parsing Table:
1. E→E+T 4. T→F
2. E→T 5. F→id
3. T→T*F 6. F→(E)

3 Steps:
(1) Augment the grammar
(2) Construct the transition sets and transition table
(3) Write out the table
Step (1) Augment the grammar: Add one more production: $E' \to E$

Step (2)

(i) Construct transition sets:
Define $I_0 = \text{Closure}([E' \to E, E \to E + T, E \to T, T \to T\ast F, T \to F, F \to \text{id}, F \to (E)])$

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

$I_2 = N(I_0, T) = \{ [E \to T.], [T \to T.\ast F] \}$

$I_3 = N(I_0, F) = \{ [T \to F.], [F \to \text{id}] \}$

$I_4 = N(I_0, ( ) = \{ [F \to (E)], [E \to E + T], [E \to T], [T \to T\ast F], [T \to F], [F \to \text{id}], [F \to (E)] \}$

$I_5 = N(I_0, \text{id}) = \{ [F \to \text{id}.] \}$

$I_6 = N(I_1, +) = \ldots$

(ii) Transition Table (Please see the handout)

Step (3) Write out the table

(i) Change the entry n in the action part to $S_n$.

(ii) If a state contains a completed item, then for each symbol in the follow set of LHS, put the reduction rule.

(iii) If a state $q$ contains the item $[E' \to E.]$, then the action for $[q, \$]$ is ACCT.

(ii) Example:
$I_2 = N(I_0, T) = \{ [E \to T.], [T \to T.\ast F] \}$

$[E \to T.]$ is a completed item.

Follow($E$) = {$\$, +, $\}$, $E \to T$ is production #2.

So, put R2 to $\$, +, $\$, columns for state 2.

Another example:
1. $S \to E=E$
2. $S \to \text{id}$
3. $E \to E+\text{id}$
4. $E \to \text{id}$

Following the steps to build the table.

(i) 0. $S' \to S$

(ii) Transition sets:
$I_0 = \text{closure}([S' \to S]) = \{ [S' \to S], [S \to E=E], [S \to \text{id}], [E \to E+\text{id}], [E \to \text{id}] \}$

$I_1 = N(I_0, S) = \{ [S' \to S.] \}$

$I_2 = N(I_0, E) = \{ [S \to E=E], [E \to E+\text{id}] \}$

$I_3 = N(I_0, \text{id}) = \{ [S \to \text{id}.], [E \to \text{id}.] \}$

$I_4 = N(I_2, =) = \{ [S \to E=E], [E \to E+\text{id}], [E \to \text{id}] \}$

$I_5 = N(I_2, +) = \{ [E \to E+\text{id}] \}$

$I_6 = N(I_4, E) = \{ [S \to E=E], [E \to E+\text{id}] \}$

$I_7 = N(I_4, \text{id}) = \{ [E \to \text{id}.] \}$

$I_8 = N(I_5, \text{id}) = \{ [E \to E+\text{id}.] \}$

$N(I_6, +) = \{ [E \to E+.\text{id}] \}$
(iii) Write out the table:
Follow(S) = {$}; Follow(E) = {=, +, $}

<table>
<thead>
<tr>
<th></th>
<th>=</th>
<th>id</th>
<th>+</th>
<th>$</th>
<th>S</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>S3</td>
<td></td>
<td></td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>acct</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>S4</td>
<td></td>
<td>S5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>R4</td>
<td></td>
<td>R4</td>
<td>R2/R4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>S7</td>
<td></td>
<td></td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>S8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>S5</td>
<td></td>
<td>R1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>R4</td>
<td></td>
<td>R4</td>
<td>R4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>R3</td>
<td></td>
<td>R3</td>
<td>R3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Conflict (more than one values in a cell)*
Reasons:
1. Grammar is ambiguous
2. The technique is not powerful enough.
Distinguish these 2 reasons:
Parse a string that will use the conflict cell, if both values work, then the conflict is caused by the ambiguous grammar; if just one value works, then the conflict is caused by the 2nd reason (technique is not powerful enough).
Previous example:
Conflict in cell [3, $]
Parse an identifier.

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action/Production</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a$</td>
<td>S3</td>
</tr>
<tr>
<td>0a3</td>
<td>$</td>
<td>R2 (S$\rightarrow$ id)</td>
</tr>
<tr>
<td>0S1</td>
<td>#</td>
<td>acct</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action/Production</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a$</td>
<td>S3</td>
</tr>
<tr>
<td>0a3</td>
<td>$</td>
<td>R4 (E$\rightarrow$ id)</td>
</tr>
<tr>
<td>0E2</td>
<td>$</td>
<td>Empty (Error)</td>
</tr>
</tbody>
</table>

So, the conflict is caused by the parsing technique.

(b) CLR (Canonical LR)
It’s a more powerful method than SLR.
One more concept: CLR item.
SLR item: [E$\rightarrow$.E+T]
CLR item: [E$\rightarrow$.E+T; l], E$\rightarrow$.E+T is the core; and l is the look ahead set (a set of tokens).
Example:
[E$\rightarrow$.E+T; {+}], [E$\rightarrow$.E+T; {=}].

3 modifications:
1. Our starting state $I_0 = \text{Closure}([E' \rightarrow .E; \{\}$ ]).

2. For each item, $[A \rightarrow \alpha.B\gamma; l]$ in the closure set do:
   
   For every production $B \rightarrow \beta$ do
   
   Create an initial item $[B \rightarrow \beta; t]$ with $t = \bigcup_{x \in l} \text{First}(\gamma x)$
   
   If the cores are the same, merge the items with same core.

3. Reduce for completed items in each state for tokens in LAS (look ahead set)
   
   rather than the follow set.

Example:
1. $S \rightarrow E=E$  
2. $S \rightarrow \text{id}$  
3. $E \rightarrow E+\text{id}$  
4. $E \rightarrow \text{id}$

(Please see the handout, and we will discuss it in class)

CLR can solve the conflict problem, but it creates 11 states in this example. Comparing
with SLR the disadvantage of CLR is that the machine it creates is bigger (more states).
The advantage is it is more powerful than SLR (can solve conflict.)

(c) LALR (Look-Ahead LR)
It’s the best LR parser.
It can resolve the conflict and minimize the number of states.
2 ways to construct a LALR. We only consider the easy way here.
- Create CLR states (sets)
- Merge states that have the same core.

Some tools (for UNIX) for compiler constructing:
- YACC: Yet Another Compiler Compiler, a parser generator (generates LALR).
  Input: rules; output: parser.
- LEX: (Lexical Analyzer generator). Output C program.
DOS version:
BISON (DOS YACC)
FLEX (DOS LEX)

CFL Hierarchy

All CFL

Ambiguous Unambiguous

Non-Deterministic Deterministic

LALR

LL SLR

Operator Precedence

Suggested Homework Assignments for Chapter 4:
4.5 a, c; 4.8, 4.12
IV. Error Handling
(i) Give meaningful error message
(ii) Recovery from the error

(i) Error message:
- Location (such as line #)
- Token, Lexeme
- Type of error
- What tokens are valid

(ii) Recovery (Example: x = a b;)
1. Skip Tokens until something that can be recognized, such as “;”, then go on.
2. Fake token.

V. Symbol Table
Database for symbols in the source.
Example:
Procedure xyz
    var x: integer; /* Put x into symbol table */
    begin
        x = … /* Look up symbol table */

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Length</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>int</td>
<td>…</td>
<td>…</td>
</tr>
<tr>
<td>y</td>
<td>real</td>
<td>…</td>
<td>…</td>
</tr>
</tbody>
</table>

Two operations for symbol table: insert and look up.
1. Organization of symbol table
   i. Array: allocate fixed size memory block.
      Adv.: Simple organization, less pointer overhead
      Disadv.: Search: $\Theta(n)$, n is the size of table. Not flexible.
   ii. Linked List:
      Adv: Dynamic Allocation
      Disadv.: Search: $\Theta(n)$, pointer overhead.
   iii. Binary Tree
      Adv.: Search: $\Theta(lg n)$, Dynamic allocation
      Disadv.: More pointer overhead.
   iv. Hash Table (Array + Linked list)
      Adv.: Search: $\Theta(\lambda)$, $\lambda$ is average size of the buckets, it depends on the hash function.
      Dynamic.
      Disadv.: Pointer overhead. But better than Linked-list.

2. Scope Issues (of the symbols)
   - no scope (all global)
   - All local (no global)
   - Nested (Local + global)
Compiler uses dynamic symbol table (Shrinks and grows) to handle the scope issue. That is, if we enter a nesting, then the table grows; if we leave the nesting, the table shrinks.

Example:

Procedure xyz
var x, y, z:integer;
begin
  ...
  procedure abc
var x:real;
begin
  ...
    x = ...;
  ...
    end;
  x = ...;
  ...
end.

leaving nested:

search

real
x
y
z

integer
x
y
z

integer