Lecture 7 Outline

- Assignment 3
- Continue with bottom up parsing
- LR parsing

reading Assignment: Section 4.7

Assignment 3: Develop Syntax Analyzer

- a parser generator
  - Bison = Construction of Useful Parsers
- C++ successor to yacc (for C); also cup (for Java)
  - a standard UNIX parser generator
  - yacc = Yet Another Compiler Compiler

```
xxx.y
  parser specification
```

```
Bison
```

```
parser.c
```
Input to Bison

• The specification includes:
  – optional c declarations (e.g., #include <ctype.h>)
  – optional user code
  – terminal and nonterminal declarations
  – optional precedence and associativity declarations
  – grammar rules with associated actions

Declarations of terminals and nonterminals

• all terminals and nonterminals used by your grammar must be declared

```plaintext
%token CLASS
%token CLASS ELSE FI
%token arrow "=>"
%token <symbol> TYPEID
%type <program> program
%type <classes> class_list
```

• all terminals with a value must declare their type
• non terminals with a value must also declare their type
Precedence Declarations

• You can resolve an ambiguous grammar like
  \[ E \rightarrow E + E \mid E - E \mid E \cdot E \mid E / E \]
• by rewriting the grammar
  – introduce new nonterminals
• or using precedence declarations
  \%
  \left symbols...
  \right <type> symbols...
  \nonassoc symbol

  – recall that precedence declarations resolve shift/reduce conflicts

Grammar Rules

• the heart of the specification
  – (Optional) Declaration of start symbol
    \%
    \start start_symbol
    First nonterminal by default
  – Syntax of Grammar rules
    result: rule1-components...
    | rule2-components...
    ...
  ;
  – Recursive rules
  – Actions
Outline of Bison grammar files

%{
C declarations
%
Bison declarations
%
%
Grammar rules
%
%
Additional C code
Comments enclosed in /* ... */ may appear in any of the sections

%token CLASS 258 IF 261 IN 262
%token ASSIGN 280 NOT 281
%type <program> program
%type <classes> class_list

%left '-' '+'
%left '*' '/'
%left LET_STMT
%
program : class_list
         { ast_root = program($1); }
         ;
expr    : expr '+' expr
         { $$ = plus($1,$3); }
         ;
%

Consider the ambiguous grammar:

\[ E \rightarrow E \ast E \mid E + E \mid (E) \mid \text{int} \]

One successful shift-reduce parse:

<table>
<thead>
<tr>
<th>sentential form</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>int*int+int</td>
<td>shift</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>E*E+#+int</td>
<td>reduce E \rightarrow E*E</td>
</tr>
<tr>
<td>E#+int</td>
<td>shift</td>
</tr>
<tr>
<td>E+#int</td>
<td>shift</td>
</tr>
<tr>
<td>E+int#</td>
<td>reduce E \rightarrow \text{int}</td>
</tr>
<tr>
<td>E+E#</td>
<td>reduce E \rightarrow E+E</td>
</tr>
<tr>
<td>E#</td>
<td></td>
</tr>
</tbody>
</table>

Another successful shift reduce parse - conflict!

| ...               |            |
| E*E#int           | shift      |
| E*E+#int          | shift      |
| E*E+int#          | reduce E \rightarrow \text{int} |
| E*E+E#            | reduce E \rightarrow E+E |
| E*E#              | reduce E \rightarrow E*E |
| E#                |            |
In the first step shown, we can either shift or reduce by \[ E \rightarrow E^*E \]

• Choice because of precedence of + and *

• Same problem with association of * and +

• Can always rewrite ambiguous grammars of this sort to encode precedence and association in the grammar

• Sometimes result in convoluted grammars

• Tools have other means to encode precedence and association

But must get rid of conflicts!

Know what a handle is but not clear how to detect them

At any step, parser sees only stack, not whole input

**Definition:** \( \alpha \) is a viable prefix if

1. There is a \( w \) where \( \alpha w \) is a right sentential form
2. \( \alpha \# w \) is a configuration of a shift-reduce parser

Alternatively: A prefix of a rightmost derived sentential form is viable if it does not extend past the right end of the handle.

A prefix is viable because it can be extended by adding terminals to form a valid (rightmost derived) sentential form.

As long as the parser has viable prefixes on the stack, on parsing error has not been detected.

**Types of bottom up parsers:**

• simple precedence
• operator precedence
• LR family
LR Parsers - family of parsers

LR (k) - L- Left to right
R - Rightmost derivation in reverse
k elements of look ahead

Attractive
1. LR - powerful ~ virtually all language constructs
2. Efficient
3. LL(k) ⊂ LR (k)
4. LR parsers can detect an error as soon as it is possible to do so in a left-right scan
5. Automatic technique to generate one - YACC, Bison

See how bottom up parsers work

3 types of LR parsers

SLR - simple LR - easiest to implement
- not as powerful
Canonical LR - most powerful
- expensive to implement
LALR - look ahead LR
- in between the 2 previous ones in power and overhead

Overall parsing algorithm is the same - table is different
LR parsers

- LR parser, each reduction needed for parse is detected on the basis of
  - left context
  - reducible phrase
  - k terminals symbols of look ahead

- LL parser
  - left context
  - first k symbols of what right hand side derive (combined phrase and what’s to right of phrase)

3 types of LR parsers

SLR - simple LR - easiest to implement
  - not as powerful
Canonical LR - most powerful
  - expensive to implement
LALR - look ahead LR
  - in between the 2 previous ones in power and overhead

Overall parsing algorithm is the same - table is different
Each state summarizes the information contained in the stack below it — what has been seen so far:

\[(S_0X_1S_1X_1\ldots X_mS_m\#a_ia_{i+1}\ldots a_n\#) \rightarrow X_1\ldots X_m a_i \ldots a_n\] — right sentential

State at the top of the stack and current input — index into parsing table to determine whether to shift or reduce:

**Parse table: Action & Goto**

- \(\text{action}[S_m,a_i] \rightarrow\)
  - shift S: where S is a state
  - reduce by a grammar production \(\alpha \rightarrow \beta\)
  - accept
  - error
- go to[S_m, grammar symbol] go to another state

Assume \(S_0X_1S_1X_2S_2\ldots \ldots X_mS_m\#a_ia_{i+1}\ldots \#\)

right sentential form \(X_jX_2\ldots X_m a_i a_{i+1}\ldots \)
1. Action\([S_m, a_i]\) is shift input and go to state \(S\) 
   \((S_0X_1...X_mS_m\#a_i...\#\$)\)
2. Action\([S_m, a_i]\) is reduce \(A \rightarrow \beta\) 
   | \(\beta\) | = \(r\), pop off \(2r\) symbols 
   \((S_0X_1...X_{m-r}S_{m-r}A \#a_i\#a_{i+1}...\$)\)
   where \(S = \text{go to } [S_{m-r}, A]\)
   output generated after reduce tree
3. action\([S_m, a_i]\) = accept - parsing is complete
4. action\([S_m, a_i]\) = error

Bottom up only use 2 operations:
Shift and Reduce on stack#input

Stack#input
Shift:
ExT#abc\(\Rightarrow\) ExTa#bc

Reduce:
ExTa#bc\(\Rightarrow\) ExF#bc
• LR parsers
  Can tell handle by looking at stacktop [grammar symbol, state] and k input symbols - finite state automaton.

  In practice $k \leq 1$

  How to construct LR parse table from grammar.

  First construct SLR parser

  LR & LALR are augmented basic SLR techniques

• 2 phases to construct table

  1. Build deterministic finite state automation to go from state to state.

  2. Build table from DFA

Each state - how do we know from grammar where we are in the parse. Production already seen.

---

Notion of an LR(0) item (0 look ahead)

An item is a production with a distinguished position on the right hand side - position indicates how much of the production already seen.

Example

$$S \to a \, B \, S$$ is a production

Items for the production:

$$S \to . \, a \, B \, S$$

$$S \to a. \, B \, S$$

$$S \to a \, B \, . \, S$$

$$S \to a \, B \, S.$$ called LR(0) items -

Basic idea - construct a DFA that recognizes the viable prefixes

Group items into sets - state of SLR
Construction of LR(0) items & SLR parsing table for Grammar G:

1. Create augmented grammar $G'$
   
   $G : S \rightarrow \alpha | \beta$
   
   $G' : S' \rightarrow S$
   
   $S \rightarrow \alpha | \beta$

   What else is needed?
   
   • $A \rightarrow c.d E$ - indicate new state by consuming symbol d: need go to function
   
   • $A \rightarrow cd. E$ - what are all possible things to see - all possible derivations from $E$? Add strings derivable from $E$ - closure function
   
   • $A \rightarrow cdE. = reduce to A and go to another state$

2. Compute functions closure and go to
   
   - will be used to determine the action and go to parts of the parsing table
   
   Closure - essentially defines what is expected
   
   Go to - moves from one state to another by consuming symbol
Closure (I) where I is a set of items - form states

Let N be non-terminal

- if distinguished point is in front of N then add each production for that N & put distinguished point at the beginning of the rhs

\[ A \rightarrow \alpha \quad B \beta \text{ is in } I \Rightarrow \text{we expect to see a string derivable from } B \]

\[ B \rightarrow \gamma \text{ is added to the closure, where } B \rightarrow \gamma \text{ is a production} \]

Apply rule until nothing is added

---

Example:

\[
\begin{align*}
S & \rightarrow E \\
E & \rightarrow E + T \\
E & \rightarrow T \\
T & \rightarrow id \mid (E) \\
\end{align*}
\]

Assume I = \{S \rightarrow .E\}

Closure I = \{S \rightarrow .E \\
E \rightarrow .E + T \\
E \rightarrow .T \\
T \rightarrow .id \\
T \rightarrow .(E) \} \]
Two kinds of items

- kernel items - includes $S \rightarrow .S$ and all items where point not at left end
- non-kernel items - items with points at left end can always add - not keep around

Closure of kernel items

- go to(I, X), where X is a grammar symbol,
  I - set of items
  shift action moves from one state to another by absorbing single symbol. Successor states will contain each item with distinguished point advanced by 1 grammar symbol
  if $A \rightarrow \alpha . X \beta$ is in I then closure of $A \rightarrow \alpha X . \beta$ is added to go to(I, X)

Sets are viable prefixes if

if $\gamma$ is a viable prefix for I
then $\gamma X$ is a viable prefix to go to(I, X)

Example

go to (I, ( ) = closure (T -> (.E))}
Procedure items (C), C is set of items - state begin

\[ C := \{ \text{closure (} S' \rightarrow .S \} \] 
repeat
for each set of items I in C and each grammar symbol X such that go to(I,X) is not empty and not in C do
  add go to (I,X) to C
until no more sets can added