LR Parsing: LALR(1) Parsers; Lex and Yacc

CS 440: Programming Languages and Translators, Fall 2020

A. LR(1) Parsers Generally Not Practical

- In practice, for the kinds of languages we want for programming languages, a full LR(1) parser’s CFSM is too large. Instead of one item \( A \rightarrow \beta \cdot \) to reduce, as in an SLR(1) parser, we get \( A \rightarrow \beta \cdot, c \) where the number of symbols \( c \in \Sigma \) to use depends on how and how many times \( A \) is used in the rhs of a production in the grammar. This holds for every nonterminal in the grammar (except for \( S' \)).
- So the situation is that an SLR(1) parser for a grammar (if one exists) is in practice likely to detect syntax errors later than we’d like. At the other extreme, a full LR(1) parser for the same grammar detects syntax errors as soon as possible (under some notion of "possible"), but the parser is too large.
- Ideally, we’d like a parser with of a size more like an SLR(1) parser that’s more refined than an SLR(1) parser (i.e., it detects errors earlier and/or avoids shift/reduce or reduce/reduce conflicts that make a grammar SLR(1)). Since it will have fewer states than a full LR(1) parser, however, we will have live with some amount of later error detection.

LALR(1) Parsers Use Merged LR(1) States

- Relative to an LR(1) parser, an LALR(1) parser ("Look-Ahead Left-to-Right" parser, pronounced "ell-a-ell-r") uses states that can merge multiple LR(1) parser states.
- Definition: The core of an LR(1) item or state is the LR(0) part of the item or state. (i.e., we drop the lookahead symbol).
- To get the LALR(1) parser states, we use the LR(1) parser states but merge states that have the same core.
  - Example 1: We can merge states \( \{ A \rightarrow \beta \cdot, c \} \) and \( \{ A \rightarrow \beta \cdot, d \} \) to get \( \{ A \rightarrow \beta \cdot, c; A \rightarrow \beta \cdot, d \} \).
  - Example 2: Take the states \( s_1 = \{ A \rightarrow \cdot, a, b \}, s_2 = \{ A \rightarrow \cdot, a, c \}, \) and \( s_3 = \{ A \rightarrow \cdot, a, e/f \} \). We can’t merge \( s_1 \) and \( s_2 \) because their cores are different; we can’t merge \( s_1 \) (or \( s_2 \)) with \( s_3 \) because their cores are of different sizes. (Item \( A \rightarrow \cdot, a, e/f \) is a shorthand for the two items \( A \rightarrow \cdot, a, e \) and \( A \rightarrow \cdot, a, f \), so \( s_3 \) has a size of 2.)
- Unfortunately, merging two LR(1) states can introduce conflicts. If this happens, the grammar is LR(1) but not LALR(1).
  - Example 3: Take states \( s_1 = \{ A \rightarrow \beta \cdot, c; B \rightarrow \gamma \cdot \delta, d \} \) and \( s_2 = \{ A \rightarrow \beta \cdot, d; B \rightarrow \gamma \cdot \delta, c \} \).
    Neither of them separately has a conflict, but merging them creates a state \( \{ A \rightarrow \beta \cdot, c/d; B \rightarrow \gamma \cdot \delta, c/d \} \), which has both shift/reduce and reduce/reduce conflicts.
- Next is an example of how LALR(1), LR(1), and SLR(1) parsers for the same grammar can differ.
Example: A Grammar with Different LR(1), LALR(1), and SLR(1) Parsers

The LR(1) and LALR(1) Parsers

• Figure 1 shows the LR(1) and LALR(1) CFSMs for the grammar 1: $S' \rightarrow \cdot S \; ; \; 2: S \rightarrow \cdot A \; a \; A \; b \; A \; c; \; 3: S \rightarrow \cdot B \; c; \; 4: A \rightarrow \cdot c; \; 5: B \rightarrow \cdot c$. (To save space, the "s" part of shift operations is omitted.)
• Highlighting shows how two of the LR(1) rows are merged when going to the LALR(1) parser.
  • In the LR(1) CFSM, items 4.1, b and 4.1, c are in separate states. In the LALR(1) CFSM, they are merged into one state, 4.1, b/c.
  • Item 4.1, a is part of a core with 5.1, b, so that state does not get merged with the others.

<table>
<thead>
<tr>
<th>LR(1) and LALR(1) Parsers</th>
<th>Action</th>
<th>GoTo</th>
</tr>
</thead>
<tbody>
<tr>
<td>Note</td>
<td>Items</td>
<td>a</td>
</tr>
<tr>
<td>*</td>
<td>1.0, $\emptyset: S' \rightarrow \cdot S ; ; ; \emptyset$</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>2.0, $: S \rightarrow \cdot A ; a ; A ; b ; A ; c; ; :$</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>3.0, $: S \rightarrow \cdot B ; c; ; :$</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>4.0,a: A \rightarrow \cdot c, a</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>5.0,b: B \rightarrow \cdot c, c</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>1.1, $\emptyset: S' \rightarrow S ; \cdot ; ; \emptyset$</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>2.1, a: S \rightarrow A ; \cdot A ; a ; A ; b ; A ; c; ; :$</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>2.2, $: S \rightarrow A ; a ; \cdot A ; b ; A ; c; ; :$</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>4.0,b: A \rightarrow \cdot c, b</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>2.3, $: S \rightarrow A ; a ; A ; \cdot b ; A ; c; ; :$</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>2.4, $: S \rightarrow A ; a ; A ; b ; A ; c; ; :$</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>4.0,c: A \rightarrow \cdot c, c</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>2.5, $: S \rightarrow A ; a ; A ; b ; A ; \cdot c; ; :$</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>2.6, $: S \rightarrow A ; a ; A ; b ; A ; c ; \cdot; ; :$</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>3.1, $: S \rightarrow B ; \cdot c, ; :$</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>3.2, $: S \rightarrow B ; c ; \cdot; ; :$</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>4.1,c: A \rightarrow \cdot c, a</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>5.1,c: B \rightarrow \cdot c, c</td>
<td></td>
</tr>
<tr>
<td>LR(1)</td>
<td>4.1,b: A \rightarrow c ; \cdot, ; b</td>
<td></td>
</tr>
<tr>
<td>LR(1)</td>
<td>4.1,c: A \rightarrow c ; \cdot, ; c</td>
<td></td>
</tr>
<tr>
<td>LALR(1)</td>
<td>4.1,b/c: A \rightarrow c ; \cdot, ; b/c</td>
<td></td>
</tr>
</tbody>
</table>

Figure 1: LR(1) and LALR(1) CFSMs

Different Behavior by the LALR(1) and LR(1) Parsers

• Let’s compare the merged LALR(1) state $\{A \rightarrow c \; \cdot, \; b/c\}$ with the unmerged LR(1) states $\{A \rightarrow c \; \cdot, \; b\}$ and $\{A \rightarrow c \; \cdot, \; c\}$.
• In the correct input $c \; a \; c \; b \; c \; c \; \cdot \; :$, state $\{A \rightarrow c \; \cdot, \; b\}$ checks for the required $b$, so it notes an error in $c \; a \; c \; \subseteq c \; \ldots$ when we try to reduce the second $c$ (indicated by the underline). In the LALR(1)
parser, we are in state \( \{ A \rightarrow c \cdot, b/c \} \), so we do the reduction, go to state \( \{ S \rightarrow A a A \cdot b A c, \} \) and find an error because we can’t shift the third \( c \) in \( c a c c \) ...

- Similarly, \( \{ A \rightarrow c \cdot, c \} \) checks for the required \( c \) so it notes an error in \( c a c b c b \$ \) when we try to reduce the third \( c \). In the LALR(1) parser, we’re in state \( \{ A \rightarrow c \cdot, b/c \} \), do the reduction, go to \( \{ S \rightarrow A a A b A \cdot c, \} \) and find an error because we can’t shift the second \( b \) in \( c a c b c b \$ \).

**The SLR(1) Parser**

- Figure 2 shows the SLR(1) CFSM for the same grammar (1: \( S \rightarrow \cdot S \$ \); 2: \( S \rightarrow \cdot A a A b A c \); 3: \( S \rightarrow \cdot B c \); 4: \( A \rightarrow \cdot c \); 5: \( B \rightarrow \cdot c \)). The shift operations are unchanged but there are changes to the reductions.

- The SLR(1) parser reduces 4.1: \( A \rightarrow c \cdot \) if the next symbol is in \( \text{Follow}(A) = \{ a, b, c, \} \).

- So we add reductions (shown in red) to state 4.1: \( A \rightarrow c \cdot \) on next symbol = \( a \) or \( \$ \).

- We also add reductions to state \( \{ 4.1: A \rightarrow c \cdot; 5.1: B \rightarrow c \cdot \} \) on next symbol = \( b \), \( c \), or \( \$ \). This produces a reduce/reduce error: On next symbol \( b \), we can reduce 4.1: \( A \rightarrow c \cdot \); or 5.1: \( B \rightarrow c \cdot \). Therefore the grammar is not SLR(1).

<table>
<thead>
<tr>
<th><strong>SLR(1) Parser</strong></th>
<th><strong>Action</strong></th>
<th><strong>GoTo</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Items</strong></td>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>1.0: ( S' \rightarrow \cdot S $ )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.0: ( S \rightarrow \cdot A a A b A c )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.0: ( S \rightarrow \cdot B c )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.0: ( A \rightarrow \cdot c )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.0: ( B \rightarrow \cdot c )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.1: ( S' \rightarrow S \cdot $ )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.1: ( S \rightarrow A \cdot a A b A c )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.2: ( S \rightarrow A a A \cdot A b A c )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.0: ( A \rightarrow \cdot c )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.3: ( S \rightarrow A a A \cdot b A c )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.4: ( S \rightarrow A a A b \cdot A c )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.0: ( A \rightarrow \cdot c )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.5: ( S \rightarrow A a A b A \cdot c )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.6: ( S \rightarrow A a A b A c \cdot )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.1: ( S \rightarrow B \cdot c )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.2: ( S \rightarrow B c \cdot )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.1: ( A \rightarrow c \cdot )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.1: ( B \rightarrow c \cdot )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.1: ( A \rightarrow c \cdot )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Figure 2: The SLR(1) CFSM*
B. Lex and Yacc

• **Lex** a program that generates lexical analyzers (= "lexers") using regular expressions to describe the structure of tokens. Lex exists in many different versions, flex being a common one (it's free and open source).

• Lex implements output lexer using C, but there are versions of lex that produce lexers in many different languages (such as ones for OCaml and Python).

• A Lex input file has three sections, separated by %% lines. (The lex code below is for a lexer that counts and then reports the number of blanks in the input.)
  
  • **Definition section**
    
    - Code that typically define or declare things used in the lexer code. E.g.,
    
    ```
    %{
    #include <stdio.h>
    int blanks = 0; // how many blanks read?
    int upper = 0; // how many upper case letters read?
    int lower = 0; // how many lower case letters read?
    %}
    
    %
  ```

  • **Rules section**
    
    - Includes the definitions of tokens as given by regular expressions.
    - Also includes "action" code to execute when a token is identified. (E.g., to take a string of digits and calculate the corresponding natural number.)
    
    ```
    [ ]  {  blanks++; }
    [A-Z]  { upper++; }
    [a-z]  { lower++; }
    .  { } // dot = ignore everything else
    \n  { } // even ignore newline
  ```

  • **C Code section:** Code to implement the actions in the rules section.
    
    ```
    int main() {
    yylex();
    printf("%d blanks, %d uppercase, %d lowercase\n",
           blanks, upper, lower);
    }
    ```

• Running llex (varies by environment)
  
  - `> lex filename.l` -- builds lexer, puts it in file lex.yy.c
  - `> gcc lex.yy.c -ll` -- link with lex library (this varies by environment)
• *yacc* is a program that generates compilers (*yacc* = "Yet Another Compiler-compiler").
• As with lex, there exist different versions, and for different languages.
• A yacc input file has three sections, separated by `%` lines. (The lex code below is for a lexer that counts and then reports the number of blanks in the input.)
  • Definition section
    • Code that typically define or declare things used in the yacc action code.
    • If we’re using a lex-generated lexer, we include some interface information.
    • E.g.,
      ```
      %{ 
      #include <stdio.h>
      #include <stdlib.h>   /* for exit */
      int yylex();                 /* lexer routine */
      void yyerror(void *);  /* parsing error handler */
      %}
      /* The lexer should return NUMBER when it recognizes one */
      %token NUMBER
      %
      • Rules section
    • Includes the definitions of grammar rules (context-free)
    • Like lex, includes "action" code to execute when a rule is recognized.
    • E.g., here’s input to recognize a sequence of lines, with one number per line.
    • It prints "Done!" on end-of-input.
    • The $1 stands for "the value associated with the first item of this rule."
    • If we wanted a line to be associated with a value, we’d assign $$ = value;
      ```
      lines : line lines
      |  { printf("Done!\n"); exit(0); } 
      line : NUMBER \n'  { printf("%d\n", $1); }
      ;
      %
      • C Code section: Lexer code and code to support actions
      ```
      #include "lex.yy.c" /* Assuming we’re using lex to generate lexer */
Here's a small lexer based on one from the PDB for natural numbers.

/* Very simple lexer that recognizes natural numbers, 
passes parentheses, +, *, and \n to parser, 
ignores all other characters. 
Usage: > lex thisfile.l */

int yylval; /* This actually is the default */
%
#include <stdlib.h>
%
/* natnum = natural number */
natnum \[0-9\]+ 
%
{natnum} { yylval = atoi(yytext); 
    return NUMBER; /* token NUMBER defined in yacc file */ 
 }

\[\]+\*\n\] \{ return yytext[0]; \} 
. \{ } /* ignore everything else (blanks, letters, etc). */ 
%
/* No C code needed */

And here's a small yacc file for a 2-operator calculator (again, based on one from the PDB). It's designed to go with the lexer above for numbers.

In the declaration code section

- The yylex function header declaration references the lexer defined by the lex file above.
- The yyerror function header declaration is for an error message routine that the yacc code will use. The void * pointer lets yyerror more easily take a pointer to any kind of value.
- The %token NUMBER declaration tells yacc to create the declaration of NUMBER used by the lex file.

In the implementation code section

- The #include of lex.yy.c inserts the lexer code generated by lex, so that the yacc code can access it.

/* Yacc input for calculator with natural numbers and + and */
Usage: yacc thisfile.y ; gcc y.tab.c -ly -ll ; a.out
%include <stdio.h>
%include <stdlib.h>    /* for exit */
int yylex();           /* lexer routine */
void yyerror(void *);  /* parsing error handler */
%
%token NUMBER
%
lines   : line lines
         | { printf("done!\n"); exit(0); }
line    : expr \n'       { printf("%d\n", $1); }
expr    : expr '+' term     { $$ = $1 + $3; }
| term
term    : term '*' factor   { $$ = $1 * $3; }
| factor
factor  : '(' expr ')'      { $$ = $2; }
| NUMBER
%
#include "lex.yy.c"   /* Lexer code */