LR(0) Drawbacks

Consider the unambiguous augmented grammar:

0.) S → E $
1.) E → T + E
2.) E → T
3.) T → x

If we build the LR(0) DFA table, we find that there is a shift-reduce conflict.

This arises because the reduce rule was too naïve for LR(0). It put the reduce action for all terminals, when realistically, we would only reduce when we see something in the Follow set of the production we are reducing by.

Simple LR (SLR)

New algorithm for adding reduce actions into an SLR table:

\[
R ← \{\} \\
\text{for each state } I \text{ in } T \\
\quad \text{for each item } A → α • β \text{ in } I \\
\quad \quad \text{for each token } X \text{ in } \text{Follow}(A) \\
\quad \quad \quad R ← R \cup \{(I, X, A → α)\} \\
\] //i.e., \(M[I, X] = \text{rn}\)

SLR is actually useful as it is powerful enough to deal with many programming language constructs we'd wish to use.

However, SLR parsers cannot handle constructs such as pointer dereferences in C.

LR(1)

An LR(1) item consists of:

- A grammar production,
- A right-hand-side position (\(\cdot\)), and
- A lookahead symbol.

An LR(1) item \(A → α•β, x\) indicates that the sequence \(α\) is on top of the stack, and at the head of the input is a string derivable from \(βx\).

LR(1) Items

\[
\text{Closure}(I) = \text{repeat} \\
\quad \text{for any item } (A → α•β, z) \text{ in } I \\
\quad \quad \text{for any production } X → γ \\
\quad \quad \quad \text{for any } w ∈ \text{First}(βz) \\
\quad \quad \quad \quad I ← I \cup \{(X → γ • w, z)\} \\
\quad \text{until } I \text{ does not change} \\
\quad \text{return } I \\
\text{Goto}(I, X) = J ← \{\} \\
\quad \text{for any item } (A → α•β, z) \text{ in } I \\
\quad \quad \text{add } (A → α•β, z) \text{ to } J \\
\quad \text{return Closure}(J).
\]

LR(1) Start State and Reduce

The start state is the closure of the item \(S → S \$ \$, \$\), where the lookahead symbol \$ will not matter, because \$ will never be shifted.

Reduce actions are chosen by this algorithm:

\[
R ← \{\} \\
\text{for each state } I \text{ in } T \\
\quad \text{for each item } (A → α•β, z) \text{ in } I \\
\quad \quad R ← R \cup \{(I, z, A → α)\} \\
\]

SLR Parse

\[
\begin{array}{|c|c|c|c|c|c|}
\hline
\text{Stack} & \text{Input} & \text{Action} & \text{X} & \text{S} & \text{E} & \text{T} \\
\hline
S1 & x + x + x & \text{Shift 5} & 1 & a5 & g2 & g3 \\
S1 + 5 & x + x + x & \text{Reduce T → x} & 2 & a & 3 & a4 & a2 \\
S1 + 4 & x + x & \text{Shift 5} & 4 & a5 & g6 & g3 \\
S1 + 3 & x + x & \text{Reduce T → x} & 5 & r3 & r3 \\
S1 + 2 & x + x & \text{Shift 4} & 6 & r1 \\
S1 + 1 & x + x & \text{Shift 5} & 7 & r3 & r3 \\
S1 + 0 & x + x & \text{Reduce T → x} & 8 & r1 \\
S1 + 3 & x + x & \text{Reduce E → T} & 9 & r5 \\
S1 + 2 & x + x & \text{Reduce E → T} & 10 & r5 \\
S1 + 1 & x + x & \text{Reduce E → T} & 11 & r5 \\
S1 + 0 & x + x & \text{Reduce E → T} & 12 & r5 \\
S1 + 9 & \text{Accept} & & & & & \\
\hline
\end{array}
\]

LR(1) Items

\[
\begin{align*}
\text{Closure}(I) &= \text{repeat} \\
& \quad \text{for any item } (A → α•β, z) \text{ in } I \\
& \quad \quad \text{for any production } X → γ \\
& \quad \quad \quad \text{for any } w ∈ \text{First}(βz) \\
& \quad \quad \quad \quad I ← I \cup \{(X → γ • w, z)\} \\
& \quad \text{until } I \text{ does not change} \\
& \quad \text{return } I \\
\text{Goto}(I, X) &= J ← \{\} \\
& \quad \text{for any item } (A → α•β, z) \text{ in } I \\
& \quad \quad \text{add } (A → α•β, z) \text{ to } J \\
& \quad \text{return Closure}(J).
\end{align*}
\]
LR(1) Drawbacks

The number of states is now potentially huge.

Every SLR(1) grammar is LR(1) but LR(1) has more states than SLR(1) – orders of magnitude differences

LALR(1) – look ahead LR parsing method

- Used in practice because most syntactic structure can be represented by LALR (not true for SLR)
- Same number of states as SLR
- Can be constructed by merging states with the same core

Construction on LALR Parser

One solution:

- construct LR(1) items
- merge states ignoring lookahead
- if no conflicts, you have a LALR parser

Inefficient because of building LR(1) items are expensive in time and space

Efficient construction of LALR parsers

- avoid construction of LR(1) items
- construction states containing only LR(0) kernel items
- compute look ahead for the kernel items
- predict actions using kernel items

LALR vs. LR Parsing

LALR languages are not natural

- They are an efficiency hack on LR languages

Any reasonable programming language has a LALR(1) grammar

LALR(1) has become a standard for programming languages and for parser generators

A Hierarchy of Grammar Classes

Summary

Regular languages were insufficient for programming language constructs. While entire programs are not context free, the grammars of most programming languages are.

Approaches to CFG parsers usually limit the grammar productions in some way to reduce the cost of parsing.

Top-down parsers try to parse from the start symbol to the sentence:

- Recursive descent parsers may backtrack and get stuck in left recursion
- LL(1) works without backtracking, but requires left-factored languages

More powerful parsers can be built bottom-up: LR(k)

- LL(0) is insufficient, but SLR can be useful.
- LR(1) is huge but LALR(1) is powerful and fast enough for most programming language constructs today.

Generating Parsers
Using Parser Generator

Most common parser generators are LALR(1)

A parser generator constructs a LALR(1) table and reports an error when a table entry is multiply defined

- A shift and a reduce – report shift/reduce conflict
- Multiple reduces – report reduce/reduce conflict
- An ambiguous grammar will generate conflicts
- Must resolve conflicts

Shift/Reduce Conflicts

Typically due to ambiguities in the grammar

Classic example: the dangling else

\[ S \rightarrow \text{if } E \text{ then } S \text{ if } E \text{ then } S \text{ else } S \text{ if } E \text{ then } S \text{ else } S \text{ ...} \]

will have DFA state containing

\[ [S \rightarrow \text{if } E \text{ then } S, \text{ else}] \]
\[ [S \rightarrow \text{if } E \text{ then } S \text{ else } S, \text{ x}] \]

If else follows then we can shift or reduce.

The default (YACC, bison, Java CUP, etc.) resolution is to shift

- The default behavior is correct in this case

More Shift/Reduce Conflicts

Consider the ambiguous grammar:

\[ E \rightarrow E \cdot E \mid E \ast E \mid \text{int} \]

we will have the states containing:

\[ [E \rightarrow E \cdot E, +] \]
\[ [E \rightarrow E \ast E, +] \]
\[ ... \]

We have a shift/reduce conflict on input +

- we need to reduce ( \ast \text{ is higher than } \cdot \text{ )}
- Solution: specify the precedence of \ast \text{ and } \cdot

Dangling Else

Back to our dangling else example

\[ [S \rightarrow \text{if } E \text{ then } S, \text{ else}] \]
\[ [S \rightarrow \text{if } E \text{ then } S \text{ else } S, \text{ x}] \]

can eliminate conflict by declaring else with higher precedence than then

But this starts to look like “hacking the tables”

Best to avoid overuse of precedence declarations or you will end with unexpected parse trees.

Reduce/Reduce Conflicts

Usually due to ambiguity in the grammar

Example: a sequence of identifiers

\[ S \rightarrow \varepsilon \mid \text{id} \mid \text{id} S \]

There are two parse trees for the string id

\[ S \rightarrow \text{id} \]
\[ S \rightarrow \text{id} S \rightarrow \text{id} \]

How does this confuse the parser?

Reduce/Reduce Conflicts

Consider the states

\[ [S \rightarrow \varepsilon, \text{id}] \]
\[ [S \rightarrow \varepsilon, \text{id} S] \]
\[ [S \rightarrow \text{id} S, \varepsilon] \]
\[ [S \rightarrow \text{id} S, S] \]

Reduce/reduce conflict on input “idS”

\[ S' \rightarrow S \rightarrow \text{id} \]
\[ S' \rightarrow S \rightarrow \text{id} S \rightarrow \text{id} \]

Better to rewrite the grammar:

\[ S \rightarrow \varepsilon \mid \text{id} S \]
Semantic Actions

Semantic actions are implemented for LR parsing
- keep attributes on the semantic stack – parallel syntax stack
- on shift a, push attribute for a on semantic stack
- on reduce $X \rightarrow \alpha$
  - pop attributes for $\alpha$
  - compute attribute for $X$
  - push it on the semantic stack

Create AST
- Bottom up
- Creating leaf node for tokens and create internal nodes from subtrees.

Error Recovery

Error detected when parser consults parsing table
- empty entry
- Canonical LR will not make a single reduction before announcing an error
- SLR and LALR parser may make several reductions before announcing an error but not shift an erroneous symbol on the stack

Simple error recovery
- continue to scan down the stack until a state S with a goto on a particular non-terminal $A$ is found
- zero or more input symbols are discarded until a symbol "$a" is found that can follow $A$
- goto(S, $A$) put on stack and parsing continues

Choice of $A$ – non-terminal representing major program piece
- e.g. if $A$ is 'stmt' the 'a' may be 'end' or ';

Java CUP

Java Based Constructor of Useful Parsers (CUP).

CUP is a system for generating LALR parsers from simple specifications.

It serves the same role as the widely used program YACC and in fact offers most of the features of YACC.

However, CUP is written in Java, uses specifications including embedded Java code, and produces parsers which are implemented in Java.

Lexer

```java
import java_cup.runtime.Symbol;
%
class ExprLex
tcup
timplements sym
tline
tcolumn
%
private void error(String message) {
    System.err.println("Error at line "+(yyline+1)+", column "+(yycolumn+1)+" : "+message);
}%
int = 0 | -?[1-9]\[0-9\]*
new_line = \r|\n|\r\n|\z;
white_space = \{new_line\} | \[ \t\f\]%
%%
/* keywords */
"+" { return new Symbol(PLUS, yyline+1, yycolumn+1); }
"-" { return new Symbol(MINUS, yyline+1, yycolumn+1); }
"*" { return new Symbol(TIMES, yyline+1, yycolumn+1); }
"/" { return new Symbol(DIV, yyline+1, yycolumn+1); }
"(" { return new Symbol(LPAREN, yyline+1, yycolumn+1); }
")" { return new Symbol(RPAREN, yyline+1, yycolumn+1); }
"{int}" {return new Symbol(INT, yyline+1, yycolumn+1, new
    Integer.parseInt(yytext())); }
"{new_line}" { return new Symbol(EOL, yyline+1, yycolumn+1); }
"{white_space}" { /* ignore */ }
/* error fallback */
./\[m { error("Illegal character <"+ yytext()+">";
}%
```

Parser

```java
import java_cup.runtime.Symbol;
%
/* Preliminaries to use the scanner. */

parse code {}
parser code {}
public ExprParser(ExprLex lex) { super(lex); lexer = lex; }};
/* Terminal tokens returned by lexer. */
/terminal TIMES, DIV;
terminal PLUS, MINUS;
terminal LPAREN, RPAREN;
terminal EOL;
terminal Integer INT;
non terminal line_list;
non terminal line;
non terminal expr;
```
Parser

precedence left PLUS, MINUS;
precedence left TIMES, DIV;

start with line_list;

line_list ::= line_list:list line |

line ::= expr:result EOL |

expr ::= expr:l PLUS expr:r |

expr:l MINUS expr:r |

expr:l DIV expr:r |

expr:l TIMES expr:r |

LPAREN expr:e RPAREN |

INT:i |

Driver Program

import java_cup.runtime.Symbol;

public class Calc {
    public static void main(String[] args) {
        if(args.length != 1) {
            System.err.println("Usage: java Calc file.asm");
            System.exit(1);
        }
        Symbol parse_tree = null;
        try {
            ExprParser parser = new ExprParser(new ExprLex(
                new java.io.FileInputStream(args[0])));
            parse_tree = parser.parse();
        } catch (java.io.IOException e) {
            System.err.println("Unable to open file: " + args[0]);
        } catch (Exception e) {
            e.printStackTrace(System.err);
        }
    }
}

Build

jflex ExprLex.flex
java -jar java-cup-11a.jar -interface -parser ExprParser ExprParser.cup
javac Calc.java

Output

------- CUP 0.11a beta 20060608 Parser generation summary -------
  0 errors and 0 warnings
  10 terminals, 3 non-terminals, and 11 productions declared,
  producing 19 unique parse states.
  0 terminals declared but not used.
  0 non-terminals declared but not used.
  0 productions never reduced.
  0 conflicts detected (0 expected).
  Code written to "ExprParser.java" and "sym.java".

Run

Test.txt:
3+4
3 * 4 - 2
(3+2) * -2

java Calc test.txt
Result: 7
Result: 10
Result: -10
Ambiguity

expr:l PLUS expr:r |
expr:l MINUS expr:r |
expr:l DIV expr:r |
expr:l TIMES expr:r |
...

If we remove:
precedence left PLUS, MINUS;
precedence left TIMES, DIV;

Shift/Reduce Conflicts

Error

Error: *** More conflicts encountered than expected -- parser generation aborted

-------- CUP v0.11a beta 20060608 Parser generation summary --------
1 error and 16 warnings
18 terminals, 3 non-terminals, and 11 productions declared,
producing 19 unique parse states.
0 terminals declared but not used.
0 non-terminals declared but not used.
0 productions never reduced.
16 conflicts detected (0 expected).
No code produced.

-------------------------------------------------------- (v0.11a beta 20060608)