Fall 2017-2018 Compiler Principles
Lecture 4: Parsing part 3

Roman Manevich
Ben-Gurion University of the Negev
Tentative syllabus

- Front End
  - Scanning
  - Top-down Parsing (LL)
  - Bottom-up Parsing (LR)

- Intermediate Representation
  - Operational Semantics
  - Lowering

- Optimizations
  - Dataflow Analysis
  - Loop Optimizations

- Code Generation
  - Register Allocation
  - Instruction Selection

Mid-term

Exam
Previously

• LR(0) parsing
  – Running the parser
  – Constructing transition diagram
  – Constructing parser table
  – Detecting conflicts
AGENDA

• SLR

• LR(1)

• LALR(1)

• Automatic LR parser generation

• Handling ambiguities
SLR PARSING
SRL (Simple LR) parsing

• **Observation:** a handle should not be reduced to non-terminal \( N \) if the next token cannot follow \( N \)

• A reduce item \( N \rightarrow \alpha \bullet \) is applicable only when the next token \( b \) is in FOLLOW(\( N \))
  - If \( b \) is not in FOLLOW(\( N \)) we just proved there is no terminating derivation \( S \rightarrow^* \beta Nb \) and thus it is safe to remove the reduce item from the conflicted state

• SLR differs from LR(0) only on ACTION table
  - Now a row in the parsing table may contain both shift actions and reduce actions and we need to consult the current token to decide which one to take
Exercise: What is FOLLOW(T)?

\[
\begin{align*}
S & \rightarrow E \, $ \\
E & \rightarrow T \\
E & \rightarrow E \, + \, T \\
T & \rightarrow \text{id} \\
T & \rightarrow ( \, E \, ) \\
T & \rightarrow \text{id}[E]
\end{align*}
\]
### SLR action table

<table>
<thead>
<tr>
<th>State</th>
<th>id</th>
<th>+</th>
<th>(</th>
<th>)</th>
<th>[</th>
<th>]</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>shift</td>
<td>shift</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>shift</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>shift</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>accept</td>
</tr>
<tr>
<td>3</td>
<td>shift</td>
<td>shift</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>E→E+T</td>
<td>E→E+T</td>
<td>E→E+T</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>r5</td>
<td>r5</td>
<td>r5</td>
<td>T→id</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>E→T</td>
<td>E→T</td>
<td>E→T</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>shift</td>
<td>shift</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>shift</td>
<td>shift</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>T→(E)</td>
<td>T→(E)</td>
<td>T→(E)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**SLR – use 1 token look-ahead**

**LR(0) – no look-ahead**

Lookahead token from the input

---

Lookahead token from the input

... as before...

T → id

T → id[E]
BEYOND SLR PARSING
Going beyond SLR

• Some common language constructs introduce conflicts even for SLR

(0) $S' \rightarrow S$
(1) $S \rightarrow L = R$
(2) $S \rightarrow R$
(3) $L \rightarrow * R$
(4) $L \rightarrow id$
(5) $R \rightarrow L$
shift/reduce conflict

(0) $S' \rightarrow S$
(1) $S \rightarrow L = R$
(2) $S \rightarrow R$
(3) $L \rightarrow * R$
(4) $L \rightarrow id$
(5) $R \rightarrow L$

- $S \rightarrow L \bullet = R$ vs. $R \rightarrow L \bullet$
- $\text{FOLLOW}(R)$ contains $=$
  - $S \rightarrow L = R \rightarrow * R = R$
- SLR cannot resolve conflict
Inputs requiring shift/reduce

• For the input id the rightmost derivation $S' \rightarrow S \rightarrow R \rightarrow L \rightarrow id$ requires reducing in $q2$

• For the input id = id
  $S' \rightarrow S \rightarrow L \rightarrow R \rightarrow L \rightarrow id \rightarrow id = id$
  requires shifting
LR(1) grammars

• In SLR: a reduce item $N \rightarrow \alpha \bullet$ is applicable only when the lookahead is in $\text{FOLLOW}(N)$
• But for a given context (state) are all tokens in $\text{FOLLOW}(N)$ indeed possible?
  – Not always
  – We can compute a context-sensitive (i.e., specific to a given state) subset of $\text{FOLLOW}(N)$ and use it to remove even more conflicts

• LR(1) keeps lookahead with each LR item
• Idea: a more refined notion of $\text{FOLLOW}$ computed per item
Hypothesis about $\alpha\beta$ being a possible handle: so far we’ve matched $\alpha$, expecting to see $\beta$ and after reducing $N$ we expect to see the token $t$
LR(1) items

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

$$S' \rightarrow S$$
$$S \rightarrow L = R$$
$$S \rightarrow R$$
$$L \rightarrow * R$$
$$L \rightarrow id$$

LR(0) items

- $[L \rightarrow \bullet id]$  
- $[L \rightarrow id \bullet]$  

LR(1) items

- $[L \rightarrow \bullet id, *]$  
- $[L \rightarrow \bullet id, =]$  
- $[L \rightarrow \bullet id, id]$  
- $[L \rightarrow \bullet id, \$]$  
- $[L \rightarrow id \bullet, *]$  
- $[L \rightarrow id \bullet, =]$  
- $[L \rightarrow id \bullet, id]$  
- $[L \rightarrow id \bullet, \$]$
Computing Closure for LR(1)

• For every $[A \rightarrow \alpha \bullet B\beta , c]$ 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 $S$
Back to the conflict

• Is there a conflict now?

\[
\begin{align*}
(S \rightarrow L \cdot = R, \ $) \\
(R \rightarrow L \cdot, \ $) \\
(S \rightarrow L = \cdot R, \ $) \\
(R \rightarrow \cdot L, \ $) \\
(L \rightarrow \cdot * R, \ $) \\
(L \rightarrow \cdot id, \ $)
\end{align*}
\]
Question

• What is SLR(1)?
LALR(1)
LALR(1)

- LR(1) tables have huge number of entries
- Often don’t need such refined observation (and cost)
- Idea: find states with the same LR(0) component and merge their lookaheads component as long as there are no conflicts
- LALR(1) not as powerful as LR(1) in theory but works quite well in practice
  - Merging may not introduce new shift-reduce conflicts, only reduce-reduce, which is unlikely in practice
```
(S' → S , $)
(S → L = R , $)
(S → R , $)
(L → * R , =)
(L → id , =)
(R → L , $)
(L → id , $)
(L → * R , $)

(S → R , $)
(R → L , $)
(L → id , =)
(L → * R , $)
(L → id , $)

(S → L = R , $)
(S → L · = R , $)
(R → L , $)
(L → id , $)
(L → id , =)
(R → * R , $)
(R → * R , =)
(L → id , $)
(R → L , $)
(L → * R , $)
(L → * R , =)
(R → L , $)
(L → id , $)
(L → * R , $)
(L → * R , =)
```
Left/Right- recursion

• At home: create a simple grammar with left-recursion and one with right-recursion
• Construct corresponding LR(0) parser
  – Any conflicts?
• Run on simple input and observe behavior
  – Attempt to generalize observation for long inputs
Example: non-LR(1) grammar

(1) \( S \rightarrow Y \ b \ c \ \$ \)
(2) \( S \rightarrow Z \ b \ d \ \$ \)
(3) \( Y \rightarrow a \)
(4) \( Z \rightarrow a \)

\[ \begin{align*}
S & \rightarrow \cdot \ Y \ b \ c, \ \$ \\
S & \rightarrow \cdot \ Y \ b \ c, \ \$ \\
Y & \rightarrow \cdot \ a, \ b \\
Z & \rightarrow \cdot \ a, \ b \\
\end{align*} \]

reduce-reduce conflict on lookahead ‘b’
Automated Parser Generation (via CUP)
High-level structure

Lexer spec

JFlex

Lexer.java

.javac

Lexical analyzer

text

tokens

(Token.java)

Parser spec

CUP

Parser.java

sym.java

Parser

AST

LANG.lex

LANG.cup

Tokenizer.java

Token.java
Expression calculator

expr → expr + expr
| expr - expr
| expr * expr
| expr / expr
| - expr
| ( expr )
| number

Goals of expression calculator parser:
• Is 2+3+4+5 a valid expression?
• What is the meaning (value) of this expression?
Syntax analysis with CUP

- CUP – parser generator
- Generates an LALR(1) Parser
- Input: spec file
- Output: a syntax analyzer
  - Can dump automaton and table

![Diagram](image-url)

**Diagram Notes:**
- **1.** Input file: `Parser spec` is processed by CUP.
- **2.** Output file: CUP generates a `.java` file
- **3.** Compilation: `javac` compiles the `.java` file.
- **4.** Output: A syntax analyzer is generated.
- **5.** The generated syntax analyzer can dump automaton and table.
CUP spec file

- Package and import specifications
- User code components
- Symbol (terminal and non-terminal) lists
  - Terminals go to `sym.java`
  - Types of AST nodes
- Precedence declarations
- The grammar
  - Semantic actions to construct AST
PARSING AMBIGUOUS GRAMMARS
Expression Calculator – 1st Attempt

terminal Integer NUMBER;
terminal PLUS, MINUS, MULT, DIV;
terminal LPAREN, RPAREN;

non terminal Integer expr;

expr ::= expr PLUS expr
   | expr MINUS expr
   | expr MULT expr
   | expr DIV expr
   | MINUS expr
   | LPAREN expr RPAREN
   | NUMBER

Symbol type explained later
Ambiguities

\[
\begin{align*}
\text{a + b * c} & \quad \text{a + b + c} \\
\text{a} + \text{b} & \quad \text{a} + \text{b} + \text{c} \\
\end{align*}
\]
Ambiguities as conflicts for LR(1)

Warning: *** Shift/Reduce conflict found in state #41
between Expr ::= Expr PLUS Expr (*)
and Expr ::= Expr (*) TIMES Expr
under symbol TIMES
Resolved in favor of shifting.

\[ a + b \times c \]

Warning: *** Shift/Reduce conflict found in state #41
between Expr ::= Expr PLUS Expr (*)
and Exp ::= Exp (*) PLUS Expr
under symbol PLUS
Resolved in favor of shifting.

\[ a + b + c \]
Expression Calculator – 2\textsuperscript{nd} Attempt

terminal Integer \textbf{NUMBER};
terminal PLUS, MINUS, MULT, DIV;
terminal LPAREN, RPAREN;
terminal UMINUS;
non terminal Integer expr;

precedence left PLUS, MINUS;
precedence left DIV, MULT;
precedence left UMINUS;

expr ::= expr PLUS expr
| expr MINUS expr
| expr MULT expr
| expr DIV expr
| \textbf{MINUS} expr \%prec UMINUS
| LPAREN expr RPAREN
| NUMBER

Increasing precedence

Contextual precedence
Parsing ambiguous grammars using precedence declarations

• Each terminal assigned with precedence
  – By default all terminals have lowest precedence
  – User can assign his own precedence
  – CUP assigns each production a precedence
    • Precedence of rightmost terminal in production
    • or user-specified contextual precedence

• On shift/reduce conflict resolve ambiguity by comparing precedence of terminal and production and decides whether to shift or reduce

• In case of equal precedences `left/right` help resolve conflicts
  – left means reduce
  – right means shift

• More information on precedence declarations in CUP’s manual
Resolving ambiguity (associativity)

precedence left PLUS

Warning: *** Shift/Reduce conflict found in state #41 between Expr ::= Expr PLUS Expr (*) and Expr ::= Expr (*) PLUS Expr under symbol PLUS Resolved in favor of shifting.

\[ a + b + c \]

\[ a + (b + c) \]
Resolving ambiguity (op. precedence)

precedence left PLUS
precedence left MULT

Warning: *** Shift/Reduce conflict found in state #41 between Expr ::= Expr PLUS Expr (\times) and Expr ::= Expr (\times) TIMES Expr under symbol TIMES Resolved in favor of shifting.

\( a + b \times c \)

\( a + b \times c \)
Resolving ambiguity (contextual)

precedence left MULT
MINUS expr %prec UMINUS

\[
\begin{align*}
\text{a} & \quad \text{b} \\
\text{a} & \quad \text{b}
\end{align*}
\]
Resolving ambiguity

terminal Integer NUMBER;
terminal PLUS, MINUS, MULT, DIV;
terminal LPAREN, RPAREN;
terminal UMINUS;

precedence left PLUS, MINUS;
precedence left DIV, MULT;
precedence left UMINUS;

expr ::= expr PLUS expr |
       expr MINUS expr |
       expr MULT expr |
       expr DIV expr |
       MINUS expr %prec UMINUS |
       LPAREN expr RPAREN |
       NUMBER ;

UMINUS never returned by scanner (used only to define precedence)

Rule has precedence of UMINUS
More CUP directives

- **precedence nonassoc NEQ**
  - Non-associative operators: `< > == !=` etc.
  - `1<2<3` identified as an error (semantic error?)

- **start non-terminal**
  - Specifies start non-terminal other than first non-terminal
  - Can change to test parts of grammar

- **Getting internal representation**
  - Command line options:
    - `-dump_grammar`
    - `-dump_states`
    - `-dump_tables`
    - `-dump`
import java_cup.runtime.*;

%%
%cup
%eofval{
    return new Symbol(sym.EOF);
}%eofval
NUMBER=[0-9]+
%%
<YYINITIAL>"+" { return new Symbol(sym.PLUS); }
<YYINITIAL>"-" { return new Symbol(sym.MINUS); }
<YYINITIAL>"*" { return new Symbol(sym.MULT); }
<YYINITIAL>"/" { return new Symbol(sym.DIV); }
<YYINITIAL>"(" { return new Symbol(sym.LPAREN); }
<YYINITIAL>")" { return new Symbol(sym.RPAREN); }
<YYINITIAL>{NUMBER} {
    return new Symbol(sym.NUMBER, new Integer(yytext()));
}
<YYINITIAL>\n { }
<YYINITIAL>. { }

Parser gets terminals from the scanner
Recap

• Package and import specifications and user code components

• Symbol (terminal and non-terminal) lists
  – Define building-blocks of the grammar

• Precedence declarations
  – May help resolve conflicts

• The grammar
  – May introduce conflicts that have to be resolved
ABSTRACT SYNTAX TREE CONSTRUCTION
Assigning meaning

expr ::= expr PLUS expr
      | expr MINUS expr
      | expr MULT expr
      | expr DIV expr
      | MINUS expr %prec UMINUS
      | LPAREN expr RPAREN
      | NUMBER

- So far, only validation
- Add Java code implementing semantic actions
Assigning meaning

non terminal Integer expr;

expr ::= expr::e1 PLUS expr::e2
{: RESULT = new Integer(e1.intValue() + e2.intValue()); :} \\
| expr::e1 MINUS expr::e2
{: RESULT = new Integer(e1.intValue() - e2.intValue()); :} \\
| expr::e1 MULT expr::e2
{: RESULT = new Integer(e1.intValue() * e2.intValue()); :} \\
| expr::e1 DIV expr::e2
{: RESULT = new Integer(e1.intValue() / e2.intValue()); :} \\
| MINUS expr::e1
{: RESULT = new Integer(0 - e1.intValue()); :} %prec UMINUS \\
| LPAREN expr::e1 RPAREN
{: RESULT = e1; :} \\
| NUMBER:n \\
{: RESULT = n; :} \\
;

• Symbol labels used to name variables
• RESULT names the left-hand side symbol
Abstract Syntax Trees

• More useful representation of syntax tree
  – Less clutter
  – Actual level of detail depends on your design

• Basis for semantic analysis

• Later annotated with various information
  – Type information
  – Computed values

• Technically – a class hierarchy of abstract syntax tree nodes
Parse tree vs. AST
AST hierarchy example
AST construction

• AST Nodes constructed during parsing
  – Stored in push-down stack

• Bottom-up parser
  – Grammar rules annotated with actions for AST construction
  – When node is constructed all children available (already constructed)
  – Node (RESULT) pushed on stack
AST construction

1 + (2) + (3)
expr + (2) + (3)
expr + (expr) + (3)
expr + (3)
expr + (expr)
expr

expr ::= expr:el PLUS expr:e2
    {: RESULT = new plus(e1,e2); :}
| LPAREN expr:e RPAREN
    {: RESULT = e; :}
| INT_CONST:i
    {: RESULT = new int_const(..., i); :}
Example of lists

terminal Integer NUMBER;
terminal PLUS,MINUS,MULT,DIV,LPAREN,RPAREN,SEMI;
terminal UMINUS;
non terminal Integer expr;
non terminal expr_list, expr_part;
precedence left PLUS, MINUS;
precedence left DIV, MULT;
precedence left UMINUS;

expr_list ::= expr_list expr_part
         | expr_part
       ;
expr_part ::= expr:e { : System.out.println("= " + e); :} SEMI
        ;
expr ::= expr PLUS expr
       | expr MINUS expr
       | expr MULT expr
       | expr DIV expr
       | MINUS expr %prec UMINUS
       | LPAREN expr RPAREN
       | NUMBER
     ;
Next lecture:

IR and Operational Semantics