CS 480

Translators (Compilers)

weeks 4: yacc, LR parsing

Instructor: Liang Huang

(some slides courtesy of David Beazley and Zhendong Su)
ONLY on HWI cases (60% of grade)
ply.yacc preliminaries

• ply.yacc is a module for creating a parser

• Assumes you have defined a BNF grammar

```plaintext
assign  : NAME EQUALS expr
expr    : expr PLUS term
         | expr MINUS term
         | term
term    : term TIMES factor
         | term DIVIDE factor
         | factor
factor  : NUMBER

# compare with (ambiguity):
expr    : expr PLUS expr
         | expr TIMES expr
         | NUMBER
```
import ply.yacc as yacc
import mylexer  # Import lexer information
tokens = mylexer.tokens  # Need token list

def p_assign(p):
    '''assign : NAME EQUALS expr'''

def p_expr(p):
    '''expr : expr PLUS term
            | expr MINUS term
            | term'''

def p_term(p):
    '''term : term TIMES factor
            | term DIVIDE factor
            | factor'''

def p_factor(p):
    '''factor : NUMBER'''

yacc.yacc()  # Build the parser
ply.yacc example

import ply.yacc as yacc
import mylexer            # Import lexer information
tokens = mylexer.tokens

# Need token list

def p_assign(p):
    '''assign : NAME EQUALS expr'''

def p_expr(p):
    '''expr : expr PLUS term
            | expr MINUS term
            | term'''

def p_term(p):
    '''term : term TIMES factor
            | term DIVIDE factor
            | factor'''

def p_factor(p):
    '''factor : NUMBER'''

yacc.yacc()            # Build the parser
import ply.yacc as yacc
import mylexer            # Import lexer information
tokens = mylexer.tokens   # Need token token list

def p_assign(p):
    '''assign : NAME EQUALS expr'''

def p_expr(p):
    '''expr : expr PLUS term
           | expr MINUS term
           | term'''

def p_term(p):
    '''term : term TIMES factor
           | term DIVIDE factor
           | factor'''

def p_factor(p):
    '''factor : NUMBER'''

yacc.yacc()            # Build the parser

Note: Name doesn't matter as long as it starts with p_
import ply.yacc as yacc
import mylexer  # Import lexer information
tokens = mylexer.tokens  # Need token list

def p_assign(p):
    '''assign : NAME EQUALS expr'''

def p_expr(p):
    '''expr : expr PLUS term
            | expr MINUS term
            | term'''

def p_term(p):
    '''term : term TIMES factor
            | term DIVIDE factor
            | factor'''

def p_factor(p):
    '''factor : NUMBER'''

yacc.yacc()  # Build the parser

docstrings contain grammar rules from BNF
ply.yacc example

import ply.yacc as yacc
import mylexer            # Import lexer information
tokens = mylexer.tokens   # Need token list

def p_assign(p):
    '''assign : NAME EQUALS expr'''

def p_expr(p):
    '''expr : expr PLUS term
            | expr MINUS term
            | term'''

def p_term(p):
    '''term : term TIMES factor
            | term DIVIDE factor
            | factor'''

def p_factor(p):
    '''factor : NUMBER'''

yacc.yacc()            # Build the parser using introspection
ply.yacc parsing

• yacc.parse() function

```python
yacc.yacc()    # Build the parser
...
data = "x = 3*4+5*6"
yacc.parse(data)  # Parse some text
```

• This feeds data into lexer
• Parses the text and invokes grammar rules
A peek inside

• PLY uses LR-parsing. LALR(1)
• AKA: Shift-reduce parsing
• Widely used parsing technique
• Table driven
Bottom-Up Parsing

• Bottom-up parsing is more general than top-down parsing
  – And just as efficient
  – Builds on ideas in top-down parsing
  – Preferred method in practice

• Also called LR parsing
  – L means that tokens are read left to right
  – R means that it constructs a rightmost derivation
An Introductory Example

• LR parsers
  - Don’t need left-factored grammars, and
  - Can handle left-recursive grammars

• Consider the following grammar

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

  - Why is this not LL(1)?

• Consider the string: \text{int} + ( \text{int} ) + ( \text{int} )
The Idea

- LR parsing \textit{reduces} a string to the start symbol by \textit{inverting productions}:

\[
\text{str} \leftarrow \text{input string of terminals}
\]

\[
\text{repeat}
\]

- Identify $\beta$ in str such that $A \rightarrow \beta$ is a production (i.e., $\text{str} = \alpha \beta \gamma$)
- Replace $\beta$ by $A$ in str (i.e., str becomes $\alpha A \gamma$)

\[
\text{until str} = S
\]
A Bottom-up Parse in Detail (1)

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

\[ \text{int} + (\text{int}) + (\text{int}) \]
A Bottom-up Parse in Detail (2)

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

\[ \text{int} + (\text{int}) + (\text{int}) \]

\[ E + (\text{int}) + (\text{int}) \]
A Bottom-up Parse in Detail (3)

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

\[
\begin{align*}
\text{int} & + (\text{int}) + (\text{int}) \\
E & + (\text{int}) + (\text{int}) \\
E & + (E) + (\text{int})
\end{align*}
\]
A Bottom-up Parse in Detail (4)

\[
\begin{align*}
\text{int} & + (\text{int}) + (\text{int}) \\
E & + (\text{int}) + (\text{int}) \\
E & + (E) + (\text{int}) \\
E & + (\text{int}) \\
\end{align*}
\]
A Bottom-up Parse in Detail (5)

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

\[
\text{int} + (\text{int}) + (\text{int})
\]
\[
E + (\text{int}) + (\text{int})
\]
\[
E + (E) + (\text{int})
\]
\[
E + (\text{int})
\]
\[
E + (E)
\]
A Bottom-up Parse in Detail (6)

A rightmost derivation in reverse

(always rewrite the rightmost nonterminal in each step)
Important Fact #1

Important Fact #1 about bottom-up parsing:

An LR parser traces a rightmost derivation in reverse
Where Do Reductions Happen

Important Fact #1 has an interesting consequence:
- Let $\alpha\beta\gamma$ be a step of a bottom-up parse
- Assume the next reduction is by $A \rightarrow \beta$
- Then $\gamma$ is a string of terminals!

Why? Because $\alpha A \gamma \rightarrow \alpha \beta \gamma$ is a step in a right-most derivation
Notation

- Idea: Split the string into two substrings
  - Right substring (a string of terminals) is as yet unexamined by parser
  - Left substring has terminals and non-terminals

- The dividing point is marked by a ▶
  - The ▶ is not part of the string

- Initially, all input is unexamined: ▶x_1x_2 \ldots x_n
Shift-Reduce Parsing

- Bottom-up parsing uses only two kinds of actions:
  
  Shift

  Reduce
Shift

**Shift:** Move ➤ one place to the right
- Shifts a terminal to the left string

\[ E + (\text{int }) \Rightarrow E + (\text{int} \; \text{➤}) \]
Reduce

Reduce: Apply a production in reverse at the right end of the left string
- If $E \rightarrow E + (E)$ is a production, then

$$E + (E + (E) \Rightarrow) \Rightarrow E + (E \Rightarrow)$$
Shift-Reduce Example

- $\text{int} + (\text{int}) + (\text{int})$

  \text{Shift}
Shift-Reduce Example

1. int + (int) + (int)$  shift
2. int $ + (int) + (int)$  red. E → int

int  +  (  int  )  +  (  int  )
Shift-Reduce Example

int + (int) + (int)$  shift
int + (int) + (int)$  red. E → int
E + (int) + (int)$  shift 3 times
Shift-Reduce Example

► int + (int) + (int)$   shift

int ▶ + (int) + (int)$   red. E → int

E ▶ + (int) + (int)$   shift 3 times

E + (int ▶ ) + (int)$   red. E → int

E

/ 

int + ( int ) + ( int )
Shift-Reduce Example

- \( \text{int} \rightarrow + (\text{int}) + (\text{int}) \) $ shift$
- \( \text{int} \rightarrow + (\text{int}) + (\text{int}) \) $ red. E \rightarrow \text{int}$
- \( E \rightarrow + (\text{int}) + (\text{int}) \) $ shift \text{ 3 times}$
- \( E \rightarrow + (\text{int}) + (\text{int}) \) $ red. E \rightarrow \text{int}$
- \( E \rightarrow + (\text{int}) + (\text{int}) \) $ shift$

```
E
/  /
/  /
int  +  (  int  )  +  (  int  )
```
Shift-Reduce Example

- \texttt{int + (int) + (int)$} shift
- \texttt{int $\rightarrow$ + (int) + (int)$} red. E $\rightarrow$ int
- \texttt{E $\rightarrow$ + (int) + (int)$} shift 3 times
- \texttt{E $\rightarrow$ + (int) + (int)$} red. E $\rightarrow$ int
- \texttt{E $\rightarrow$ + (E $\rightarrow$ ) + (int)$} shift
- \texttt{E $\rightarrow$ + (E $\rightarrow$ ) + (int)$} red. E $\rightarrow$ E + (E)
Shift-Reduce Example

\[ \text{int} + (\text{int}) + (\text{int}) \]

- **shift**

\[ \text{int} \rightarrow + (\text{int}) + (\text{int}) \]

- **red. E \rightarrow int**

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

- **shift 3 times**

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

- **red. E \rightarrow int**

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

- **shift**

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

- **red. E \rightarrow E + (E)\]

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

- **shift 3 times**
Shift-Reduce Example

- int + (int) + (int)$ shift
- int $+ (int) + (int)$ red. E → int
- E $+ (int) + (int)$ shift 3 times
- E + (int $+ (int)$ red. E → int
- E + (E $+ (int)$ shift
- E + (E $+ (int)$ red. E → E + (E
- E $+ (int)$ shift 3 times
- E + (int $+ (int)$ red. E → int

```
Shift-Reduce Example

- int + (int) + (int)$ shift
- int $+ (int) + (int)$ red. E → int
- E $+ (int) + (int)$ shift 3 times
- E + (int $+ (int)$ red. E → int
- E + (E $+ (int)$ shift
- E + (E $+ (int)$ red. E → E + (E
- E $+ (int)$ shift 3 times
- E + (int $+ (int)$ red. E → int
```
Shift-Reduce Example

\[
\begin{align*}
&\text{shift} & &\text{red. } E \rightarrow \text{int} \\
&\text{shift 3 times} & &\text{red. } E \rightarrow \text{int} \\
&\text{shift} & &\text{red. } E \rightarrow E + (E) \\
&\text{shift 3 times} & &\text{red. } E \rightarrow \text{int} \\
&\text{shift} & &
\end{align*}
\]
Shift-Reduce Example

- $\text{int} + (\text{int}) + (\text{int})$: shift
- $\text{int} \rightarrow + (\text{int}) + (\text{int})$: red. $E \rightarrow \text{int}$
- $E \rightarrow + (\text{int}) + (\text{int})$: shift 3 times
- $E + (\text{int} \rightarrow ) + (\text{int})$: red. $E \rightarrow \text{int}$
- $E + (E \rightarrow ) + (\text{int})$: shift
- $E + (E) \rightarrow + (\text{int})$: red. $E \rightarrow E + (E)$
- $E \rightarrow + (\text{int})$: shift 3 times
- $E + (\text{int} \rightarrow )$: red. $E \rightarrow \text{int}$
- $E + (E \rightarrow )$: shift
- $E + (E) \rightarrow$: red. $E \rightarrow E + (E)$
Shift-Reduce Example

$\uparrow \text{int }+ \text{(int }+ \text{(int))}$ shift
\text{int }\uparrow + \text{(int }+ \text{(int))}$ red. E $\rightarrow$ int
E $\uparrow + \text{(int }+ \text{(int))}$ shift 3 times
E + (int $\uparrow$ ) + (int)$ red. E $\rightarrow$ int
E + (E $\uparrow$ ) + (int)$ shift
E + (E $\uparrow$ ) + (int)$ red. E $\rightarrow$ E + (E)
E $\uparrow + \text{(int))}$ shift 3 times
E + (int $\uparrow$ )$ red. E $\rightarrow$ int
E + (E $\uparrow$ )$ shift
E + (E $\uparrow$ )$ red. E $\rightarrow$ E + (E)
E $\uparrow$ $\rightarrow$ accept
A Hierarchy of Grammar Classes

From Andrew Appel, “Modern Compiler Implementation in Java”
Shift/Reduce Conflicts

• If a DFA state contains both
  \[X \rightarrow \alpha \cdot a\beta, b\] and \[Y \rightarrow \gamma \cdot, a\]

• Then on input “a” we could either
  - Shift into state \[X \rightarrow \alpha a \cdot \beta, b\], or
  - Reduce with \[Y \rightarrow \gamma\]

• This is called a shift-reduce conflict
Shift/Reduce Conflicts

• Typically due to ambiguities in the grammar
• Classic example: the dangling else
  \[ S \rightarrow \text{if } E \text{ then } S \mid \text{if } E \text{ then } S \text{ else } S \mid \text{OTHER} \]
• Will have DFA state containing
  \[ [S \rightarrow \text{if } E \text{ then } S, \quad \text{else}] \]
  \[ [S \rightarrow \text{if } E \text{ then } S \text{ else } S, \quad x] \]
• If \texttt{else} follows then we can shift or reduce
• Default (bison, CUP, etc.) is to shift
  - Default behavior is as needed in this case
The Stack

- Left string can be implemented as a stack
  - Top of the stack is the ➤

- Shift pushes a terminal on the stack

- Reduce
  - Pops 0 or more symbols off the stack: production rhs
  - Pushes a non-terminal on the stack: production lhs
Key Issue: When to Shift or Reduce?

- Decide based on the left string (the stack)
- Idea: use a finite automaton (DFA) to decide when to shift or reduce
  - The DFA input is the stack
  - The language consists of terminals and non-terminals

- We run the DFA on the stack and examine the resulting state X and the token tok after
  - If X has a transition labeled tok then shift
  - If X is labeled with “A → β on tok” then reduce
LR(1) Parsing: An Example

[Diagram of LR(1) parsing with rules and actions]

- **0**: int
- **1**: int + (int) + (int)$ shift
- **2**: E → int accept on $
- **3**: E + (int) + (int)$ shift
- **4**: E + (int)$ shift
- **5**: E + (E)$ shift
- **6**: E + (E) + (int)$ shift
- **7**: E → E + (E) on $ shift
- **8**: E + (int)$ shift
- **9**: E + (E)$ shift
- **10**: E → E + (E) on $ accept
- **11**: E → E + (E) on $ accept
Representing the DFA

- Parsers represent the DFA as a 2D table
  - Recall table-driven lexical analysis
- Lines correspond to DFA states
- Columns correspond to terminals and non-terminals
- Typically columns are split into:
  - Those for **terminals**: action table
  - Those for **non-terminals**: goto table
Representing the DFA. Example

The table for a fragment of our DFA

<table>
<thead>
<tr>
<th></th>
<th>int</th>
<th>+</th>
<th>( )</th>
<th>$</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>g6</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>s4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td>s7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

E → E + (E)
on $, +
The LR Parsing Algorithm

• After a shift or reduce action we rerun the DFA on the entire stack
  - This is wasteful, since most of the work is repeated
• Remember for each stack element to which state it brings the DFA
• LR parser maintains a stack

\[
\langle \text{sym}_1, \text{state}_1 \rangle \ldots \langle \text{sym}_n, \text{state}_n \rangle
\]

\text{state}_k \text{ is the final state of the DFA on } \text{sym}_1 \ldots \text{sym}_k
The LR Parsing Algorithm

Let $I = w\$ be initial input
Let $j = 0$
Let DFA state 0 be the start state
Let stack = $\langle \text{dummy}, 0 \rangle$

repeat
    case action[top_state(stack), $I[j]$] of
        shift $k$: push $\langle I[j++], k \rangle$
        reduce $X \rightarrow \alpha$:
            - pop $|\alpha|$ pairs off the stack
            - push $\langle X, \text{Goto[top_state(stack), X]} \rangle$
        accept: halt normally
        error: halt and report error
LR Parsing Notes

• *Can be used to parse more grammars than LL*

• *Most programming languages grammars are LR*

• *Can be described as a simple table*

• *There are tools for building the table*

• *How is the table constructed?*
Recap ...

- A bottom-up parser rewrites the input string to the start symbol
- The state of the parser is described as
  \[ \alpha \triangleright \gamma \]
  - \( \alpha \) is a stack of terminals and non-terminals
  - \( \gamma \) is the string of terminals not yet examined

- Initially: \( \triangleright x_1x_2 \ldots x_n \)
The Shift and Reduce Actions

• Recall the CFG: $E \rightarrow \text{int} \mid E + (E)$

• A bottom-up parser uses two kinds of actions
  - **Shift** pushes a terminal from input on the stack
    $$E + (\text{int}) \Rightarrow E + (\text{int})$$

  - **Reduce** pops 0 or more symbols off of the stack (production rhs) and pushes a non-terminal on the stack (production lhs)
    $$E + (E + (E) \text{ int}) \Rightarrow E + (E)$$
Key Issue: When to Shift or Reduce?

- Idea: use a finite automaton (DFA) to decide when to shift or reduce
  - The input is the stack
  - The language consists of terminals and non-terminals

- We run the DFA on the stack and we examine the resulting state $X$ and the token $tok$ after
  - If $X$ has a transition labeled $tok$ then shift
  - If $X$ is labeled with “$A \rightarrow \beta$ on $tok$” then reduce
Key Issue: How is the DFA Constructed?

• The stack describes the context of the parse
  - What non-terminal we are looking for
  - What production rhs we are looking for
  - What we have seen so far from the rhs

• Each DFA state describes several such contexts
  - E.g., when we are looking for non-terminal $E$, we might be looking either for an int or an $E + (E)$ rhs
LR(1) Items

• An LR(1) item is a pair
  \[ X \rightarrow \alpha \cdot \beta, \ a \]
  - \( X \rightarrow \alpha \beta \) is a production
  - \( a \) is a terminal (the lookahead terminal)
  - LR(1) means 1 lookahead terminal

• \([X \rightarrow \alpha \cdot \beta, \ a]\) describes a context of the parser
  - We are trying to find an \( X \) followed by an \( a \), and
  - We have \( \alpha \) already on top of the stack
  - Thus we need to see next a prefix derived from \( \beta a \)
Note

- The symbol $\uparrow$ was used before to separate the stack from the rest of input
  - $\alpha \uparrow \gamma$, where $\alpha$ is the stack and $\gamma$ is the remaining string of terminals
- In LR(1) items $\bullet$ is used to mark a prefix of a production rhs:
  $$X \rightarrow \alpha \bullet \beta, \alpha$$
  - Here $\beta$ might contain non-terminals as well
- In both case the stack is on the left
Convention

• We add to our grammar a fresh new start symbol $S$ and a production $S \rightarrow E$
  - Where $E$ is the old start symbol

• The initial parsing context contains:
  \[ S \rightarrow \cdot E, $ \]
  - Trying to find an $S$ as a string derived from $E$\$
  - The stack is empty
LR(1) Items (Cont.)

- In context containing
  \[ E \rightarrow E + \bullet (E), + \]
  - If ( follows then we can perform a shift to context containing
    \[ E \rightarrow E + (\bullet E), + \]
- In a context containing
  \[ E \rightarrow E + (E) \bullet, + \]
  - We can perform a reduction with \[ E \rightarrow E + (E) \]
  - But only if a + follows
LR(1) Items (Cont.)

- Consider a context with the item
  \[ E \rightarrow E + (\cdot E ), + \]
- We expect next a string derived from \( E ) + \)
- There are two productions for \( E \)
  \[ E \rightarrow \text{int} \quad \text{and} \quad E \rightarrow E + (E) \]
- We describe this by extending the context
  with two more items:
    \[ E \rightarrow \cdot \text{int}, ) \]
    \[ E \rightarrow \cdot E + (E ), ) \]
The Closure Operation

- The operation of extending the context with items is called the **closure operation**

\[
\text{Closure}(\text{Items}) = \\
\text{repeat} \\
\text{for each } [X \rightarrow \alpha \cdot Y\beta, a] \text{ in Items} \\
\text{for each production } Y \rightarrow \gamma \\
\text{for each } b \in \text{First}(\beta a) \\
\text{add } [Y \rightarrow \bullet \gamma, b] \text{ to Items} \\
\text{until Items is unchanged}
\]
Constructing the Parsing DFA (1)

• Construct the start context: Closure({S → E, $})

  \[
  \begin{align*}
  S & \rightarrow E, \\
  E & \rightarrow E+(E), E, +, \text{int, } \\
  \end{align*}
  \]

• We abbreviate as

  \[
  \begin{align*}
  S & \rightarrow E, \\
  E & \rightarrow E+(E), $/+, \text{int, } \\
  \end{align*}
  \]
Constructing the Parsing DFA (2)

• A DFA state is a **closed** set of LR(1) items
  - This means that we performed Closure

• The start state contains \([S \rightarrow \bullet E, \$]\)

• A state that contains \([X \rightarrow \alpha \bullet, b]\) is labeled with “reduce with \(X \rightarrow \alpha\) on \(b\)”

• And now the transitions …
The DFA Transitions

• A state “State” that contains \([X \rightarrow \alpha\cdot y\beta, b]\) has a transition labeled \(y\) to a state that contains the items “Transition(State, y)”
  - \(y\) can be a terminal or a non-terminal

Transition(State, y)

\[
\text{Items} \leftarrow \emptyset \\
\text{for each } [X \rightarrow \alpha\cdot y\beta, b] \in \text{State} \\
\quad \text{add } [X \rightarrow \alpha y\cdot \beta, b] \text{ to } \text{Items} \\
\text{return Closure(Items)}
\]
Constructing the Parsing DFA: An Example

S $ → \bullet E, $
E $ → \bullet E+(E), $/+,
E $ → \bullet \text{int}, $/+,

E $ → \text{int}, $/+,
E $ → E+\bullet (E), $/+,

E $ → E+(\bullet E), $/+,
E $ → \bullet E+(E), )/+,
E $ → \bullet \text{int}, )/+,

E $ → \text{int}, )/+,
E $ → \text{int}$ on $, +,

and so on...

ECS 142 Lectures 7-8 50
LR Parsing Tables. Notes

• Parsing tables (i.e. the DFA) can be constructed automatically for a CFG

• But we still need to understand the construction to work with parser generators
  - E.g., they report errors in terms of sets of items

• What kind of errors can we expect?
Shift/Reduce Conflicts

• If a DFA state contains both

\[ X \rightarrow \alpha \cdot a\beta, b \] and \[ Y \rightarrow \gamma \cdot, a \]

• Then on input “a” we could either
  - Shift into state \[ X \rightarrow \alpha a \cdot \beta, b \], or
  - Reduce with \[ Y \rightarrow \gamma \]

• This is called a shift-reduce conflict
Shift/Reduce Conflicts

- Typically due to ambiguities in the grammar
- Classic example: the dangling else
  \[ S \rightarrow \text{if E then S} \mid \text{if E then S else S} \mid \text{OTHER} \]
- Will have DFA state containing
  \[ [S \rightarrow \text{if E then S}, \quad \text{else}] \]
  \[ [S \rightarrow \text{if E then S else S}, \quad \text{x}] \]
- If else follows then we can shift or reduce
- Default (bison, CUP, etc.) is to shift
  - Default behavior is as needed in this case
More Shift/Reduce Conflicts

- Consider the ambiguous grammar
  \[ E \rightarrow E + E \mid E * E \mid \text{int} \]
- We will have the states containing
  \[ [E \rightarrow E * \cdot E, +] \quad [E \rightarrow E * E\cdot, +] \]
  \[ [E \rightarrow \cdot E + E, +] \quad \Rightarrow^E [E \rightarrow E \cdot + E, +] \]
  ... ...
- Again we have a shift/reduce on input +
  - We need to reduce (\* binds more tightly than +)
  - Recall solution: declare the precedence of \* and +
More Shift/Reduce Conflicts

- In bison declare precedence and associativity:
  \%left +
  \%left *

- Precedence of a rule = that of its last terminal
  - See bison manual for ways to override this default
    - Context-dependent precedence (Section 5.4, pp 70)

- Resolve shift/reduce conflict with a shift if:
  - no precedence declared for either rule or terminal
  - input terminal has higher precedence than the rule
  - the precedences are the same and right associative
Using Precedence to Solve S/R Conflicts

• Back to our example:

\[ [E \rightarrow E * \cdot E, +] \quad [E \rightarrow E * E \cdot, +] \]
\[ [E \rightarrow \cdot E + E, +] \Rightarrow^E [E \rightarrow E \cdot + E, +] \]

...                                ...

• Will choose reduce because precedence of rule \( E \rightarrow E * E \) is higher than of terminal +
Using Precedence to Solve S/R Conflicts

• Same grammar as before
  \[ E \rightarrow E + E \mid E \ast E \mid \text{int} \]

• We will also have the states
  \[
  [E \rightarrow E + \bullet E, +] \quad [E \rightarrow E + E\bullet, +] \\
  [E \rightarrow \bullet E + E, +] \quad \Rightarrow^E \quad [E \rightarrow E \bullet + E, +] \\
  \ldots \quad \ldots
  \]

• Now we also have a shift/reduce on input +
  - We choose reduce because \( E \rightarrow E + E \) and + have
    the same precedence and + is left-associative
Using Precedence to Solve S/R Conflicts

• Back to our dangling else example
  \[ S \to \text{if } E \text{ then } S \bullet, \quad \text{else} \]
  \[ S \to \text{if } E \text{ then } S \bullet \text{ else } S, \quad x \]

• Can eliminate conflict by declaring \text{else} with higher precedence than \text{then}
  - Or just rely on the default shift action

• But this starts to look like “hacking the parser”

• Best to avoid overuse of precedence declarations or you’ll end with unexpected parse trees
Reduce/Reduce Conflicts

• If a DFA state contains both
  \[ X \rightarrow \alpha \cdot, a \] and \[ Y \rightarrow \beta \cdot, a \]
  - Then on input “a” we don’t know which production to reduce

• This is called a reduce/reduce conflict
Reduce/Reduce Conflicts

- Usually due to gross 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 \text{id}
  \[ S \rightarrow \text{id} \]
  \[ S \rightarrow \text{id } S \rightarrow \text{id} \]

- How does this confuse the parser?
More on Reduce/Reduce Conflicts

- Consider the states

\[
[S \to \text{id} \bullet, \$] \\
[S' \to \bullet S, \$] \\
[S \to \bullet, \$] \Rightarrow^{\text{id}} [S \to \bullet, \$] \\
[S \to \bullet \text{id}, \$] \\
[S \to \bullet \text{id} S, \$]
\]

- Reduce/reduce conflict on input $\$

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

- Better rewrite the grammar:

\[
S \to \varepsilon \mid \text{id} S
\]
Using Parser Generators

• Parser generators construct the parsing DFA given a CFG
  - Use precedence declarations and default conventions to resolve conflicts
  - The parser algorithm is the same for all grammars (and is provided as a library function)

• But most parser generators do not construct the DFA as described before
  - Because the LR(1) parsing DFA has 1000s of states even for a simple language
LR(1) Parsing Tables are Big

• But many states are similar, e.g.

\[
\begin{align*}
E & \rightarrow \text{int} \bullet, \$, + \\
E & \rightarrow \text{int} \\
\end{align*}
\]

and

\[
\begin{align*}
E & \rightarrow \text{int} \bullet, \), + \\
E & \rightarrow \text{int} \\
\end{align*}
\]

• Idea: merge the DFA states whose items differ only in the lookahead tokens
  - We say that such states have the same core

• We obtain

\[
\begin{align*}
E & \rightarrow \text{int} \bullet, \$, +, )/ \\
E & \rightarrow \text{int} \\
\end{align*}
\]
The Core of a Set of LR Items

- Definition: The **core** of a set of LR items is the set of first components
  - Without the lookahead terminals

- Example: the core of
  \[
  \{ [X \rightarrow \alpha \cdot \beta, b], [Y \rightarrow \gamma \cdot \delta, d] \}
  \]
  is
  \[
  \{ X \rightarrow \alpha \cdot \beta, Y \rightarrow \gamma \cdot \delta \}
  \]
LALR States

- Consider for example the LR(1) states
  \{\{X \rightarrow \alpha \star, a\}, \{Y \rightarrow \beta \star, c\}\}
  \{\{X \rightarrow \alpha \star, b\}, \{Y \rightarrow \beta \star, d\}\}

- They have the same core and can be merged

- And the merged state contains:
  \{\{X \rightarrow \alpha \star, a/b\}, \{Y \rightarrow \beta \star, c/d\}\}

- These are called LALR(1) states
  - Stands for LookAhead LR
  - Typically 10 times fewer LALR(1) states than LR(1)
A LALR(1) DFA

- Repeat until all states have distinct core
  - Choose two distinct states with same core
  - Merge the states by creating a new one with the union of all the items
  - Point edges from predecessors to new state
  - New state points to all the previous successors
Conversion LR(1) to LALR(1). Example.
The LALR Parser Can Have Conflicts

- Consider for example the LR(1) states
  
  $$\{[X \rightarrow \alpha \cdot, a], [Y \rightarrow \beta \cdot, b]\}$$
  
  $$\{[X \rightarrow \alpha \cdot, b], [Y \rightarrow \beta \cdot, a]\}$$

- And the merged LALR(1) state
  
  $$\{[X \rightarrow \alpha \cdot, a/b], [Y \rightarrow \beta \cdot, a/b]\}$$

- Has a new reduce-reduce conflict
- In practice such cases are rare

- However, no new shift/reduce conflicts. Why?
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

From Andrew Appel, “Modern Compiler Implementation in Java”
Notes on Parsing

• Parsing
  - A solid foundation: context-free grammars
  - A simple parser: LL(1)
  - A more powerful parser: LR(1)
  - An efficiency hack: LALR(1)
  - LALR(1) parser generators
  - Didn’t discuss another variant: SLR(1)

• Now we move on to semantic analysis
General Idea

• Input tokens are shifted onto a parsing stack

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAME</td>
<td>x = 3 * 4 + 5</td>
</tr>
<tr>
<td>NAME =</td>
<td>= 3 * 4 + 5</td>
</tr>
<tr>
<td>NAME = NUM</td>
<td>3 * 4 + 5</td>
</tr>
</tbody>
</table>
* 4 + 5 |

• This continues until a complete grammar rule appears on the top of the stack
General Idea

- If rules are found, a "reduction" occurs

Stack

| NAME | NAME = |
| NAME = NUM |

Input

\[ x = 3 \times 4 + 5 \]

reduce

NAME = factor

- RHS of grammar rule replaced with LHS
Rule Functions

• During reduction, rule functions are invoked

```python
def p_factor(p):
    'factor : NUMBER'
```

• Parameter p contains grammar symbol values

```python
def p_factor(p):
    'factor : NUMBER'
    p[0]     p[1]
```

p[0]     p[1]
Using an LR Parser

- Rule functions generally process values on right hand side of grammar rule
- Result is then stored in left hand side
- Results propagate up through the grammar
- Bottom-up parsing
Example: Calculator

def p_assign(p):
    '''assign : NAME EQUALS expr'''
    vars[p[1]] = p[3]

def p_expr_plus(p):
    '''expr : expr PLUS term'''

def p_term_mul(p):
    '''term : term TIMES factor'''

def p_term_factor(p):
    '''term : factor'''
    p[0] = p[1]

def p_factor(p):
    '''factor : NUMBER'''
    p[0] = p[1]
Example: Parse Tree

def p_assign(p):
    '''assign : NAME EQUALS expr'''
    p[0] = ('ASSIGN',p[1],p[3])

def p_expr_plus(p):
    '''expr : expr PLUS term'''
    p[0] = ('+',p[1],p[3])

def p_term_mul(p):
    '''term : term TIMES factor'''
    p[0] = ('*',p[1],p[3])

def p_term_factor(p):
    '''term : factor'''
    p[0] = p[1]

def p_factor(p):
    '''factor : NUMBER'''
    p[0] = ('NUM',p[1])
Example: Parse Tree

```python
>>> t = yacc.parse("x = 3*4 + 5*6")

>>> t
('ASSIGN', 'x', ('+',
    ('*', ('NUM', 3), ('NUM', 4)),
    ('*', ('NUM', 5), ('NUM', 6))
)

>>> 
```
Why use PLY?

- There are many Python parsing tools
- Some use more powerful parsing algorithms
- Isn't parsing a "solved" problem anyways?
PLY is Informative

- Compiler writing is hard
- Tools should not make it even harder
- PLY provides extensive diagnostics
- Major emphasis on error reporting
- Provides the same information as yacc
PLY Diagnostics

• PLY produces the same diagnostics as yacc

• Yacc

  % yacc grammar.y
  4 shift/reduce conflicts
  2 reduce/reduce conflicts

• PLY

  % python mycompiler.py
  yacc: Generating LALR parsing table...
  4 shift/reduce conflicts
  2 reduce/reduce conflicts

• PLY also produces the same debugging output
Grammar

Rule 1     statement -> NAME = expression
Rule 2     statement -> expression
Rule 3     expression -> expression + expression
Rule 4     expression -> expression - expression
Rule 5     expression -> expression * expression
Rule 6     expression -> expression / expression
Rule 7     expression -> NUMBER

Terminals, with rules where they appear
*                    : 5
+                    : 3
-                    : 4
/                    : 6
=                    : 1
NAME                 : 1
NUMBER               : 7
error                :

Nonterminals, with rules where they appear
expression           : 1 2 3 4 5 5 6 6
statement            : 0

Parsing method: LALR

state 0
(0) $' -> . statement
(1) statement -> . NAME = expression
(2) statement -> . expression
(3) expression -> . expression + expression
(4) expression -> . expression - expression
(5) expression -> . expression * expression
(6) expression -> . expression / expression
(7) expression -> . NUMBER
NAME            shift and go to state 1
NUMBER         shift and go to state 2
expression     shift and go to state 4
statement      shift and go to state 3

state 1
(1) statement -> NAME . = expression
  =            shift and go to state 5

state 10
(1) statement -> NAME = expression .
(3) expression -> expression . + expression
(4) expression -> expression . - expression
(5) expression -> expression . * expression
(6) expression -> expression . / expression
Send                reduce using rule 1 (statement -> NAME = expression .)
+                shift and go to state 7
-                shift and go to state 6
*                shift and go to state 8
/                shift and go to state 9

state 11
(4) expression -> expression . - expression
(5) expression -> expression . * expression
(6) expression -> expression . / expression

! shift/reduce conflict for + resolved as shift.
! shift/reduce conflict for - resolved as shift.
! shift/reduce conflict for / resolved as shift.
Send                reduce using rule 4 (expression -> expression - expression .)
+                shift and go to state 7
-                shift and go to state 6
*                shift and go to state 8
/                shift and go to state 9
! +               [ reduce using rule 4 (expression -> expression - expression .) ]
! -               [ reduce using rule 4 (expression -> expression - expression .) ]
! *               [ reduce using rule 4 (expression -> expression - expression .) ]
! /               [ reduce using rule 4 (expression -> expression - expression .) ]
Debugging Output

... state 11

(4) expression \rightarrow\ expression - expression .
(3) expression \rightarrow\ expression . + expression
(4) expression \rightarrow\ expression . - expression
(5) expression \rightarrow\ expression . * expression
(6) expression \rightarrow\ expression . / expression

! shift/reduce conflict for + resolved as shift.
! shift/reduce conflict for - resolved as shift.
! shift/reduce conflict for * resolved as shift.
! shift/reduce conflict for / resolved as shift.
$end reduce using rule 4 (expression \rightarrow\ expression - expression .)
+
  shift and go to state 7
-
  shift and go to state 6
*
  shift and go to state 8
/
  shift and go to state 9

! +
  [ reduce using rule 4 (expression \rightarrow\ expression - expression .) ]
! -
  [ reduce using rule 4 (expression \rightarrow\ expression - expression .) ]
! *
  [ reduce using rule 4 (expression \rightarrow\ expression - expression .) ]
! /
  [ reduce using rule 4 (expression \rightarrow\ expression - expression .) ]
...

* shift and go to state 5
PLY Validation

- PLY validates all token/grammar specs
- Duplicate rules
- Malformed regexs and grammars
- Missing rules and tokens
- Unused tokens and rules
- Improper function declarations
- Infinite recursion
Error Example

import ply.lex as lex

tokens = [ 'NAME','NUMBER','PLUS','MINUS','TIMES','DIVIDE','EQUALS' ]

# Rule t_MINUS redefined.
t_ignore = ' \t'
t_PLUS   = r'\+'
t_MINUS  = r'-'
t_TIMES  = r'\*' 
t_DIVIDE = r'\/'
t_EQUALS = r'='
t_NAME   = r'\[a-zA-Z_\][a-zA-Z0-9_]*'
t_MINUS  = r'-'
t_POWER  = r'\^'

def t_NUMBER():
    r'\d+'
    t.value = int(t.value)
    return t

lex.lex() # Build the lexer
Error Example

import ply.lex as lex

tokens = [ 'NAME', 'NUMBER', 'PLUS', 'MINUS', 'TIMES', 'DIVIDE', 'EQUALS' ]

t_ignore = ' \t'
t_PLUS   = r'\+'
t_MINUS  = r'-'     
t_TIMES  = r'\*'
t_DIVIDE = r'/'
t_EQUALS = r'='     
t_NAME   = r'[^a-zA-Z_][a-zA-Z0-9_]*'
t_MINUS  = r'-'     

lex: Rule 't_POWER' defined for an unspecified token POWER

def t_NUMBER():
    r'\d+'
    t.value = int(t.value)
    return t

lex.lex()         # Build the lexer
import ply.lex as lex
tokens = [ 'NAME','NUMBER','PLUS','MINUS','TIMES','DIVIDE', 'EQUALS']
t_ignore = ' 	'
t_PLUS = r'+'
t_MINUS = r'-'
t_TIMES = r'\*'
t_DIVIDE = r'/'
t_EQUALS = r'='
t_NAME = r'[a-zA-Z_][a-zA-Z0-9_]*'
t_MINUS = r'-'
t_POWER = r'^'

def t_NUMBER():
    r'\d+'
    t.value = int(t.value)
    return t

lex.lex()  # Build the lexer
PLY is Yacc

- PLY supports all of the major features of Unix lex/yacc
- Syntax error handling and synchronization
- Precedence specifiers
- Character literals
- Start conditions
- Inherited attributes
Precedence Specifiers

- **Yacc**

  ```
  %left PLUS MINUS
  %left TIMES DIVIDE
  %nonassoc UMINUS
  ...
  expr : MINUS expr %prec UMINUS {
    $$ = -$1;
  }
  ```

- **PLY**

  ```python
  precedence = (
    ('left','PLUS','MINUS'),
    ('left','TIMES','DIVIDE'),
    ('nonassoc','UMINUS'),
  )
  def p_expr_uminus(p):
    'expr : MINUS expr %prec UMINUS'
    p[0] = -p[1]
  ```
Character Literals

• Yacc

```plaintext
expr : expr '+' expr { $$ = $1 + $3; }
| expr '-' expr { $$ = $1 - $3; }
| expr '*' expr { $$ = $1 * $3; }
| expr '/' expr { $$ = $1 / $3; }
;
```

• PLY

```python
def p_expr(p):
    '''expr : expr ' + ' expr
            | expr ' - ' expr
            | expr ' * ' expr
            | expr ' / ' expr''
...
```
Error Productions

• Yacc

    funcall_err : ID LPAREN error RPAREN {
        printf("Syntax error in arguments\n");
    }

• PLY

    def p_funcall_err(p):
        'ID LPAREN error RPAREN'
        print "Syntax error in arguments\n"
PLY is Simple

- Two pure-Python modules. That's it.
- Not part of a "parser framework"
- Use doesn't involve exotic design patterns
- Doesn't rely upon C extension modules
- Doesn't rely on third party tools
PLY is Fast

• For a parser written entirely in Python
• Underlying parser is table driven
• Parsing tables are saved and only regenerated if the grammar changes
• Considerable work went into optimization from the start (developed on 200Mhz PC)
PLY Performance

- Parse file with 1000 random expressions (805KB) and build an abstract syntax tree
  - PLY-2.3 : 2.95 sec, 10.2 MB (Python)
  - DParser : 0.71 sec, 72 MB (Python/C)
  - BisonGen : 0.25 sec, 13 MB (Python/C)
  - Bison : 0.063 sec, 7.9 MB (C)

- 12x slower than BisonGen (mostly C)
- 47x slower than pure C
- System: MacPro 2.66Ghz Xeon, Python-2.5
import ply.yacc as yacc

class MyParser:
    def p_assign(self,p):
        '''assign : NAME EQUALS expr'''
    def p_expr(self,p):
        '''expr : expr PLUS term
                | expr MINUS term
                | term'''
    def p_term(self,p):
        '''term : term TIMES factor
                | term DIVIDE factor
                | factor'''
    def p_factor(self,p):
        '''factor : NUMBER'''
    def build(self):
        self.parser = yacc.yacc(object=self)
Limitations

• LALR(1) parsing

• Not easy to work with very complex grammars (e.g., C++ parsing)

• Retains all of yacc's black magic

• Not as powerful as more general parsing algorithms (ANTLR, SPARK, etc.)

• Tradeoff: Speed vs. Generality