CS153: Compilers
Lecture 11: LR Parsing

Stephen Chong

https://www.seas.harvard.edu/courses/cs153
Contains content from lecture notes by Greg Morrisett and Steve Zdancewic
Announcements

• Reminder: CS Nights, Tuesdays 8pm
  • With pizza!

• HW3 LLVMlite out
  • Due Tuesday Oct 15 (1 week)

• HW4 Oat v1 will be released today
  • Due Tuesday Oct 29 (3 weeks)
  • Simple C-like Imperative Language
    • supports 64-bit integers, arrays, strings
    • top-level, mutually recursive procedures
    • scoped local, imperative variables
  • Compile to LLVMlite
Today

- Oat overview
- LR Parsing
  - Constructing a DFA and LR parsing table
  - Using Menhir
HW4: Oat v1

- Oat is a simple C-like imperative language
  - supports 64-bit integers, arrays, strings
  - top-level, mutually recursive procedures
  - scoped local, imperative variables
- See examples in hw04/at1programs directory
- You will:
  - Finish implementing lexer and parser
  - Compile from Oat v1 to LLVMlite
    - You can use your `backend.ml` from HW3 to compile from LLVMlite to X86!
- HW5 will extend Oat with more features...
LR($k$)

Left-to-right parse

Rightmost derivation
Derivation expands the rightmost non-terminal
(Constructs derivation in reverse order!)

$k$-symbol lookahead
LR($k$)

- Basic idea: LR parser has a stack and input
  - Given contents of stack and $k$ tokens look-ahead parser does one of following operations:
    - Shift: move first input token to top of stack
    - Reduce: top of stack matches rule, e.g., $X \rightarrow A B C$
      - Pop C, pop B, pop A, and push X
Example

Stack

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

Shift ( on to stack

Input

\((3 + 4) + (5 + 6)\)
Example

```
E → int
E → (E)
E → E + E
```

Stack

(  

Input

```
3 + 4 ) + ( 5 + 6 )
```

Shift ( on to stack
Shift 3 on to stack
Example

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

Stack

(3

Input

\((+4)+((5+6))\)

Shift ( on to stack
Shift 3 on to stack
Reduce using rule \( E \rightarrow \text{int} \)
Example

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

Stack

\[
(E)
\]

Input

\[
+4) + (5+6)
\]

Shift ( on to stack
Shift 3 on to stack
Reduce using rule \( E \to \text{int} \)
Shift + on to stack
Example

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

**Stack**

\[(E + \]

**Input**

\[4 + (5 + 6)\]

Shift ( on to stack
Shift 3 on to stack
Reduce using rule \(E \rightarrow \text{int}\)
Shift + on to stack
Shift 4 on to stack
Example

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

Stack

\(( E + 4 )\)

Input

\( ) + ( 5 + 6 )\)

Shift ( on to stack
Shift 3 on to stack
Reduce using rule \( E \rightarrow \text{int} \)
Shift + on to stack
Shift 4 on to stack
Reduce using rule \( E \rightarrow \text{int} \)
Example

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

Stack

\( (E + E) \)

Input

\( ) + (5 + 6) \)

Shift \( (\) on to stack
Shift 3 on to stack
Reduce using rule \( E \rightarrow \text{int} \)
Shift + on to stack
Shift 4 on to stack
Reduce using rule \( E \rightarrow \text{int} \)
Reduce using rule \( E \rightarrow E + E \)
Example

\begin{align*}
E & \rightarrow \text{int} \\
E & \rightarrow (E) \\
E & \rightarrow E + E
\end{align*}

Stack

( \ E \\

Input

) + ( 5 + 6 )

Reduce using rule \( E \rightarrow E + E \)

Shift \) on to stack
Example

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

**Stack**

\[(E)\]

**Input**

\[+ (5 + 6)\]

Reduce using rule \( E \rightarrow E + E \)

Shift \( ) \) on to stack

Reduce using rule \( E \rightarrow (E) \)
Example

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

Stack
\[ E \]

Input
\[ + (5 + 6) \]

Reduce using rule \[ E \rightarrow E + E \]
Shift \) on to stack
Reduce using rule \[ E \rightarrow (E) \]
Shift + on to stack
Example

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

**Stack**

\[ E + \]

**Input**

\( (5 + 6) \)

Reduce using rule \( E \rightarrow E + E \)
Shift \( ) \) on to stack
Reduce using rule \( E \rightarrow (E) \)
Shift \( + \) on to stack

... and so on ...
Example

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

Stack

\[ E + (E) \]

Reduce using rule \( E \rightarrow E + E \)
Shift ) on to stack
Reduce using rule \( E \rightarrow (E) \)
Shift + on to stack

... and so on ...
Example

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

Stack

\[ E + (E + E) \]

Reduce using rule \( E \rightarrow E + E \)
Shift \( ) \) on to stack
Reduce using rule \( E \rightarrow (E) \)
Shift + on to stack

... and so on ...

Input

( )
Example

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

Stack

\[ E + (E) \]

Reduce using rule \( E \rightarrow E + E \)
Shift \( ) \) on to stack
Reduce using rule \( E \rightarrow (E) \)
Shift + on to stack

\[ ... \text{and so on...} \]
Example

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

Stack

\[ E + E \]

Input

Reduce using rule \( E \rightarrow E + E \)
Shift \( ) \) on to stack
Reduce using rule \( E \rightarrow (E) \)
Shift + on to stack

... and so on ...

... and so on ...
Example

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

Stack

\[ E \]

Reduce using rule \( E \rightarrow E + E \)
Shift \( ) \) on to stack
Reduce using rule \( E \rightarrow (E) \)
Shift + on to stack

... and so on ...

Input
LR parsers produce a rightmost derivation

But do reductions in reverse order
What Action to Take?

- How does the LR(k) parser know when to shift and to reduce?
- Uses a DFA
  - At each step, parser runs DFA using symbols on stack as input
    - Input is sequence of terminals and non-terminals from bottom to top
  - Current state of DFA plus next $k$ tokens indicate whether to shift or reduce
Building the DFA for LR parsing

- Sketch only. For details, see Appel
- States of DFA are sets of **items**
  - An **item** is a production with an indication of current position of parser
  - E.g., Item $E \rightarrow E \cdot + E$ means that for production $E \rightarrow E + E$, we have parsed first expression $E$ have yet to parse $+$ token
  - In general, item $X \rightarrow \gamma \cdot \delta$ means $\gamma$ is at the top of the stack, and at the head of the input there is a string derivable from $\delta$
Example: LR(0)

Add new start symbol with production to indicate end-of-file

First item of first state: at the start of input
State 1: item is about to parse $S$: add productions for $S$
  From state 1, can take $x$, moving us to state 2
  From state 1, can take $\langle$, moving us to state 3
State 3: item is about to parse $L$: add productions for $L$
  State 3: item is about to parse $S$: add productions for $S$
Example: LR(0)

State 1: can take $S$, moving us to state 4

State 4 is an accepting state (if at end of input)
Example: LR(0)

Continue to add states based on next symbol in item
Example LR₀

- Build action table
- If state contains item $X \rightarrow Y \cdot \text{eof}$ then accept
- If state contains item $X \rightarrow Y$. then reduce $X \rightarrow Y$
- If state $i$ has edge to $j$ with terminal then shift

<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>shift</td>
</tr>
<tr>
<td>2</td>
<td>reduce $S \rightarrow x$</td>
</tr>
<tr>
<td>3</td>
<td>shift</td>
</tr>
<tr>
<td>4</td>
<td>accept</td>
</tr>
<tr>
<td>5</td>
<td>shift</td>
</tr>
<tr>
<td>6</td>
<td>reduce $S \rightarrow (L)$</td>
</tr>
<tr>
<td>7</td>
<td>reduce $L \rightarrow S$</td>
</tr>
<tr>
<td>8</td>
<td>shift</td>
</tr>
<tr>
<td>9</td>
<td>reduce $L \rightarrow L \cdot S$</td>
</tr>
</tbody>
</table>
Using the DFA & Action Table

• At each step, parser runs DFA using symbols on stack as input
  • Input is sequence of terminals and non-terminals from bottom to top
  • Current state of DFA and action table indicate whether to shift or reduce
Example Revisited

Stack

\[
\begin{align*}
S' &\rightarrow S \text{ eof} \\
S &\rightarrow ( L ) \\
S &\rightarrow x \\
L &\rightarrow S \\
L &\rightarrow L, S
\end{align*}
\]

Shift \((x, x)\) on to stack

State | Action
--- | ---
1 | shift
2 | reduce \(S \rightarrow x\)
3 | shift
4 | accept
5 | shift
6 | reduce \(S \rightarrow (L)\)
7 | reduce \(L \rightarrow S\)
8 | shift
9 | reduce \(L \rightarrow L, S\)
Example Revisited

Stack

\[
\begin{align*}
S' & \rightarrow S \text{ eof} \\
S & \rightarrow ( L ) \\
S & \rightarrow x \\
L & \rightarrow S \\
L & \rightarrow L, S
\end{align*}
\]

Input \( x, x \) 

Shift \( ( \) on to stack
Shift \( x \) on to stack

State | Action
--- | ---
1 | shift
2 | reduce \( S \rightarrow x \)
3 | shift
4 | accept
5 | shift
6 | reduce \( S \rightarrow ( L ) \)
7 | reduce \( L \rightarrow S \)
8 | shift
9 | reduce \( L \rightarrow L, S \)
Example Revisited

Stack

\[
\begin{align*}
S' &\rightarrow S \text{ EOF} \\
S &\rightarrow (L) \\
S &\rightarrow x \\
L &\rightarrow S \\
L &\rightarrow L, S
\end{align*}
\]

Shift ( on to stack
Shift x on to stack
Reduce \( S \rightarrow x \)

---

<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>shift</td>
</tr>
<tr>
<td>2</td>
<td>reduce ( S \rightarrow x )</td>
</tr>
<tr>
<td>3</td>
<td>shift</td>
</tr>
<tr>
<td>4</td>
<td>accept</td>
</tr>
<tr>
<td>5</td>
<td>shift</td>
</tr>
<tr>
<td>6</td>
<td>reduce ( S \rightarrow (L) )</td>
</tr>
<tr>
<td>7</td>
<td>reduce ( L \rightarrow S )</td>
</tr>
<tr>
<td>8</td>
<td>shift</td>
</tr>
<tr>
<td>9</td>
<td>reduce ( L \rightarrow L, S )</td>
</tr>
</tbody>
</table>
Example Revisited

Stack

\( S' \rightarrow S \, \text{eof} \)
\( S \rightarrow (L) \)
\( S \rightarrow x \)
\( L \rightarrow S \)
\( L \rightarrow L, S \)

Shift ( on to stack
Shift x on to stack
Reduce \( S \rightarrow x \)
Reduce \( L \rightarrow S \)

State | Action
--- | ---
1 | shift
2 | reduce \( S \rightarrow x \)
3 | shift
4 | accept
5 | shift
6 | reduce \( S \rightarrow (L) \)
7 | reduce \( L \rightarrow S \)
8 | shift
9 | reduce \( L \rightarrow L, S \)
Example Revisited

Stack

- \( S' \rightarrow S \text{ eof} \)
- \( S \rightarrow ( L ) \)
- \( S \rightarrow x \)
- \( L \rightarrow S \)
- \( L \rightarrow L, S \)

Shift \( ( \) on to stack
Shift \( x \) on to stack
Reduce \( S \rightarrow x \)
Reduce \( L \rightarrow S \)
Shift \( , \) on to stack

### State Transitions

<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>shift</td>
</tr>
<tr>
<td>2</td>
<td>reduce ( S \rightarrow x )</td>
</tr>
<tr>
<td>3</td>
<td>shift</td>
</tr>
<tr>
<td>4</td>
<td>accept</td>
</tr>
<tr>
<td>5</td>
<td>shift</td>
</tr>
<tr>
<td>6</td>
<td>reduce ( S \rightarrow ( L ) )</td>
</tr>
<tr>
<td>7</td>
<td>reduce ( L \rightarrow S )</td>
</tr>
<tr>
<td>8</td>
<td>shift</td>
</tr>
<tr>
<td>9</td>
<td>reduce ( L \rightarrow L, S )</td>
</tr>
</tbody>
</table>
Example Revisited

Stack

\( S' \rightarrow S \text{ eof} \)
\( S \rightarrow (L) \)
\( S \rightarrow x \)
\( L \rightarrow S \)
\( L \rightarrow L, S \)

Shift ( on to stack
Shift \( x \) on to stack
Reduce \( S \rightarrow x \)
Reduce \( L \rightarrow S \)
Shift , on to stack
Shift \( x \) on to stack

<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>shift</td>
</tr>
<tr>
<td>2</td>
<td>reduce ( S \rightarrow x )</td>
</tr>
<tr>
<td>3</td>
<td>shift</td>
</tr>
<tr>
<td>4</td>
<td>accept</td>
</tr>
<tr>
<td>5</td>
<td>shift</td>
</tr>
<tr>
<td>6</td>
<td>reduce ( S \rightarrow (L) )</td>
</tr>
<tr>
<td>7</td>
<td>reduce ( L \rightarrow S )</td>
</tr>
<tr>
<td>8</td>
<td>shift</td>
</tr>
<tr>
<td>9</td>
<td>reduce ( L \rightarrow L, S )</td>
</tr>
</tbody>
</table>
Example Revisited

Stack

\[ (L, x) \]

Shift ( on to stack
Shift \( x \) on to stack
Reduce \( S \to x \)
Reduce \( L \to S \)
Shift , on to stack
Shift \( x \) on to stack
Reduce \( S \to x \)

<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>shift</td>
</tr>
<tr>
<td>2</td>
<td>reduce ( S \to x )</td>
</tr>
<tr>
<td>3</td>
<td>shift</td>
</tr>
<tr>
<td>4</td>
<td>accept</td>
</tr>
<tr>
<td>5</td>
<td>shift</td>
</tr>
<tr>
<td>6</td>
<td>reduce ( S \to (L) )</td>
</tr>
<tr>
<td>7</td>
<td>reduce ( L \to S )</td>
</tr>
<tr>
<td>8</td>
<td>shift</td>
</tr>
<tr>
<td>9</td>
<td>reduce ( L \to L, S )</td>
</tr>
</tbody>
</table>
Example Revisited

Stack

\( S' \rightarrow S \; \text{eof} \)
\( S \rightarrow (L) \)
\( S \rightarrow x \)
\( L \rightarrow S \)
\( L \rightarrow L, S \)

Shift ( on to stack
Shift x on to stack
Reduce \( S \rightarrow x \)
Reduce \( L \rightarrow S \)
Shift , on to stack
Shift x on to stack
Reduce \( S \rightarrow x \)

State Action
1 shift
2 reduce \( S \rightarrow x \)
3 shift
4 accept
5 shift
6 reduce \( S \rightarrow (L) \)
7 reduce \( L \rightarrow S \)
8 shift
9 reduce \( L \rightarrow L, S \)
**Example Revisited**

**Stack**

\[
\begin{align*}
S' & \rightarrow \text{S} \text{ eof} \\
S & \rightarrow \text{ ( L )} \\
S & \rightarrow \text{ x} \\
L & \rightarrow \text{ S} \\
L & \rightarrow \text{ L, S}
\end{align*}
\]

**Reduce**

- \( S \rightarrow \text{ x} \)
- \( L \rightarrow \text{ L, S} \)

---

**State Transition Table**

<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>shift</td>
</tr>
<tr>
<td>2</td>
<td>reduce ( S \rightarrow \text{ x} )</td>
</tr>
<tr>
<td>3</td>
<td>shift</td>
</tr>
<tr>
<td>4</td>
<td>accept</td>
</tr>
<tr>
<td>5</td>
<td>shift</td>
</tr>
<tr>
<td>6</td>
<td>reduce ( S \rightarrow \text{ ( L )} )</td>
</tr>
<tr>
<td>7</td>
<td>reduce ( L \rightarrow \text{ S} )</td>
</tr>
<tr>
<td>8</td>
<td>shift</td>
</tr>
<tr>
<td>9</td>
<td>reduce ( L \rightarrow \text{ L, S} )</td>
</tr>
</tbody>
</table>
Example Revisited

Stack
(L)

Reduce S → x
Reduce L → L, S
Shift ) on to stack

<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>shift</td>
</tr>
<tr>
<td>2</td>
<td>reduce S → x</td>
</tr>
<tr>
<td>3</td>
<td>shift</td>
</tr>
<tr>
<td>4</td>
<td>accept</td>
</tr>
<tr>
<td>5</td>
<td>shift</td>
</tr>
<tr>
<td>6</td>
<td>reduce S → ( L )</td>
</tr>
<tr>
<td>7</td>
<td>reduce L → S</td>
</tr>
<tr>
<td>8</td>
<td>shift</td>
</tr>
<tr>
<td>9</td>
<td>reduce L → L , S</td>
</tr>
</tbody>
</table>
Example Revisited

Stack

$(L)$

Reduce $S \rightarrow x$
Reduce $L \rightarrow L, S$
Shift $)$ on to stack
Reduce $S \rightarrow (L)$

<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>shift</td>
</tr>
<tr>
<td>2</td>
<td>reduce $S \rightarrow x$</td>
</tr>
<tr>
<td>3</td>
<td>shift</td>
</tr>
<tr>
<td>4</td>
<td>accept</td>
</tr>
<tr>
<td>5</td>
<td>shift</td>
</tr>
<tr>
<td>6</td>
<td>reduce $S \rightarrow (L)$</td>
</tr>
<tr>
<td>7</td>
<td>reduce $L \rightarrow S$</td>
</tr>
<tr>
<td>8</td>
<td>shift</td>
</tr>
<tr>
<td>9</td>
<td>reduce $L \rightarrow L, S$</td>
</tr>
</tbody>
</table>
Example Revisited

Stack
S

Reduce S \rightarrow x
Reduce L \rightarrow L, S
Shift ) on to stack
Reduce S \rightarrow ( L )
Accept!

Stack
S

L \rightarrow S
L \rightarrow L, S

Reduce S \rightarrow x
Reduce L \rightarrow L, S
Shift ) on to stack
Reduce S \rightarrow ( L )
Accept!

State | Action
--- | ---
1 | shift
2 | reduce S \rightarrow x
3 | shift
4 | accept
5 | shift
6 | reduce S \rightarrow ( L )
7 | reduce L \rightarrow S
8 | shift
9 | reduce L \rightarrow L, S
Implementation Details

• Optimization: no need to run DFA from start state each time
  • Use stack to also record information about which DFA state corresponds to it
• Combine DFA and action table into single lookup table
LR(0) Limitations

- An LR(0) machine only works if states with reduce actions have a single reduce action.
  - In such states, the machine always reduces (ignoring lookahead).
- With more complex grammars, the DFA construction will yield states with shift/reduce and reduce/reduce conflicts:
  - Such conflicts can often be resolved by using a look-ahead symbol: LR(1)
LR(1)

• In practice, LR(1) is used for LR parsing
  • not LR(0) or LR(k) for k>1

• Item is now pair \((X \to \gamma . \delta, x)\)
  • Indicates that \(\gamma\) is at the top of the stack, and at the head of the input there is a string derivable from \(\delta x\) (where \(x\) is terminal)
  • Algorithm for constructing state transition table and action table adapted. See Appel for details.

    • Closure operation when constructing states uses FIRST(), incorporating lookahead token
    • Action table columns now terminals (i.e., 1-token lookahead)
    • Note: state transition relation and action table typically combined into single table, called parsing table
LR(0) Conflicts

• Consider the left associative and right associative “sum” grammars:
  
  left               right
  
  $S \rightarrow S + E$               $S \rightarrow E + S$
  $S \rightarrow E$                        $S \rightarrow E$
  $E \rightarrow \text{num}$                 $E \rightarrow \text{num}$
  $E \rightarrow ( S )$                      $E \rightarrow ( S )$

• One is LR(0) the other isn’t… which is which and why?
• What kind of conflict do you get? Shift/reduce or Reduce/reduce?
• Right associative gives a Shift/reduce conflict
  • Between items $S \rightarrow E$. + $S$ and $S \rightarrow E$.
• Ambiguities in associativity/precedence usually lead to shift/reduce conflicts
Dangling Else Problem

• Many language have productions such as
  \[ S \rightarrow \text{if } E \text{ then } S \text{ else } S \]
  \[ S \rightarrow \text{if } E \text{ then } S \]
  \[ S \rightarrow \ldots \]

• Program \text{if a then if b then s1 else s2} could be
  either \text{if a then \{ if b then s1 \} else s2}
  or \text{if a then \{if b then s1 else s2 \}}

• In LR parsing table there will be a shift-reduce conflict
  • \[ S \rightarrow \text{if } E \text{ then } S \cdot \text{ with lookahead else: reduce} \]
  • \[ S \rightarrow \text{if } E \text{ then } S \cdot \text{else } S \text{ with any lookahead: shift} \]
  • Which action corresponds to which interpretation of
    \text{if a then if b then s1 else s2} ?
Resolving Ambiguity

• Could rewrite grammar to avoid ambiguity
  • E.g.,

\[
\begin{align*}
S & \rightarrow O \\
O & \rightarrow V := E \\
O & \rightarrow \text{if } E \text{ then } O \\
O & \rightarrow \text{if } E \text{ then } C \text{ else } O \\
C & \rightarrow V := E \\
C & \rightarrow \text{if } E \text{ then } C \text{ else } C
\end{align*}
\]
Resolving Ambiguity

• Or tolerate conflicts, indicating how to resolve conflict
  • E.g., for dangling else, prefer shift to reduce.
    • i.e., for if a then if b then s1 else s2
      prefer if a then {if b then s1 else s2 }
      over if a then { if b then s1 } else s2
    • i.e., else binds to closest if
  • Expression grammars can express operator-precedence by resolution of conflicts
• Use sparingly! Only in well-understood cases
  • Most conflicts are indicative of ill-specified grammars
YACC and Menhir

• Yet Another Compiler-Compiler
  • Originally developed in early 1970s
  • Various versions/reimplimentations
    • Berkeley Yacc, Bison, Ocamlyacc, ...
  • From a suitable grammar, constructs an LALR(1) parser
    • A kind of LR parser, not as powerful as LR(1)
    • Most practical LR(1) grammars will be LALR(1) grammars

• Menhir
  • “90% compatible with ocamlyacc”
  • Adds some additional features including better explanations of conflicts
Menhir

• Usage: `menhir options grammar.mly`

• Produces output files
  • `grammar.ml`: OCaml code for a parser
  • `grammar.mli`: interface for parser
Structure of Menhir File

- Header and trailer are arbitrary OCaml code, copied to the output file
- Declarations of tokens, start symbols, OCaml types of symbols, associativity and precedence of operators
- Rules are productions for non-terminals, with semantic actions (OCaml expressions that are executed with production is reduced, to produce value for symbol)
Menhir example

- See `parser-eg.mll` and output files `parser-eg.ml` and `parser-eg.mli`

- Typically, the `.mly` declares the tokens, and the lexer opens the parser module

- You can get verbose ocamlyacc debugging information by doing:
  - `menhir --explain ...
  - or, if using ocamlbuild:
    `ocamlbuild --use-menhir -yaccflag --explain ...

- The result is a `<basename>.conflicts` file that contains a description of the error
- The parser items of each state use the ‘.’ just as described above
- The flag `--dump` generates a full description of the automaton
- Example: see `start-parser.mly`