Announcements

• **HW 3**: Compiling LLVMlite
  • **Due**: Tuesday, October 29\(^{th}\) at 23:59

• **HW 4**: Building a Frontend
  • **Goal**
    – Work with lexer and parser (generators)
    – Compile a C-like source language to LLVM
  • **Available soon on Moodle (i.e., by next Tuesday)**
  • **Due**: Tuesday, November 12\(^{th}\) at 23:59
How to Remove Left Recursion?

• In general

\[ S \rightarrow S \alpha_1 | \ldots | S \alpha_n | \beta_1 | \ldots | \beta_m \]

• Rewrite as

\[ S \rightarrow \beta_1 S' | \ldots | \beta_m S' \]

\[ S' \rightarrow \alpha_1 S' | \ldots | \alpha_n S' | \varepsilon \]
Bottom-up Parsing (LR Parsers)

• LR(k) parser:
  – Left-to-right scanning
  – Rightmost derivation
  – k lookahead symbols

• LR grammars are more expressive than LL
  – Can handle left-recursive (and right recursive) grammars; virtually all programming languages
  – Easier to express programming language syntax (no left factoring)

• Technique: “Shift-Reduce” parsers
  – Work bottom up instead of top down
  – Construct right-most derivation of a program in the grammar
  – Used by many parser generators (e.g. yacc, CUP, ocamlyacc, menhir, etc.)
  – Better error detection/recovery
### Top-down vs. Bottom-up

- Consider the left-recursive grammar:

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

- \((1 + 2 + (3 + 4)) + 5\)

- What part of the tree must we know after scanning just \((1 + 2\)

- In top-down, must be able to guess which productions to use…

---

<table>
<thead>
<tr>
<th>S</th>
<th>S + E</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>S + E</td>
</tr>
<tr>
<td>((S))</td>
<td>S + E</td>
</tr>
<tr>
<td>E</td>
<td>S + E</td>
</tr>
<tr>
<td>1</td>
<td>E + 4</td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

**Top-down**

<table>
<thead>
<tr>
<th>S</th>
<th>S + E</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>S + E</td>
</tr>
<tr>
<td>((S))</td>
<td>S + E</td>
</tr>
<tr>
<td>E</td>
<td>S + E</td>
</tr>
<tr>
<td>((S))</td>
<td>S + E</td>
</tr>
<tr>
<td>E</td>
<td>S + E</td>
</tr>
<tr>
<td>1</td>
<td>E + 4</td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

**Bottom-up**

Note: ‘(’ has been scanned but not consumed. Processing it is still pending.
Progress of Bottom-up Parsing

Reductions | Scanned | Input Remaining
--- | --- | ---
(1 + 2 + (3 + 4)) + 5 | (1 + 2) | (1 + 2 + (3 + 4)) + 5
(E + 2 + (3 + 4)) + 5 | (1) | 1 + 2 + (3 + 4)) + 5
(S + 2 + (3 + 4)) + 5 | (1 + 2) | + 2 + (3 + 4)) + 5
(S + E + (3 + 4)) + 5 | (1 + 2 + (3 + 4)) | + (3 + 4)) + 5
(S + (3 + 4)) + 5 | (1 + 2 + (3 + 4)) | + (3 + 4)) + 5
(S + (E + 4)) + 5 | (1 + 2 + (3 + 4)) | + (3 + 4)) + 5
(S + (S + 4)) + 5 | (1 + 2 + (3 + 4)) | + (3 + 4)) + 5
(S + (S + E)) + 5 | (1 + 2 + (3 + 4)) | + (3 + 4)) + 5
(S + (S)) + 5 | (1 + 2 + (3 + 4)) | + (3 + 4)) + 5
(S + E) + 5 | (1 + 2 + (3 + 4)) | + (3 + 4)) + 5
(S) + 5 | (1 + 2 + (3 + 4)) | + (3 + 4)) + 5
E + 5 | (1 + 2 + (3 + 4)) | + 5
S + 5 | (1 + 2 + (3 + 4)) | + 5
S + E | (1 + 2 + (3 + 4)) + 5
S

S \rightarrow S + E \mid E
E \rightarrow \text{number} \mid ( \ S \ )
Shift/Reduce Parsing

• Parser state
  – Stack of terminals and nonterminals
  – Unconsumed input is a string of terminals
  – Current derivation step is stack + input

• Parsing is a sequence of \textit{shift} and \textit{reduce} operations
  – \textbf{Shift}: Move look-ahead token to the stack
  – \textbf{Reduce}: Replace symbols $\gamma$ at top of stack with nonterminal $X$ s.t. $X \rightarrow \gamma$ is a production, i.e., $\text{pop } \gamma$, $\text{push } X$

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1 + 2 + (3 + 4)) + 5</td>
<td>shift (</td>
<td></td>
</tr>
<tr>
<td>(1 + 2 + (3 + 4)) + 5</td>
<td>shift 1</td>
<td></td>
</tr>
<tr>
<td>(S + 2 + (3 + 4)) + 5</td>
<td>reduce: $E \rightarrow \text{number}$</td>
<td></td>
</tr>
<tr>
<td>(2 + (3 + 4)) + 5</td>
<td>reduce: $S \rightarrow E$</td>
<td></td>
</tr>
<tr>
<td>(S + 2)</td>
<td>shift +</td>
<td></td>
</tr>
<tr>
<td>(S + 2 + (3 + 4)) + 5</td>
<td>shift 2</td>
<td></td>
</tr>
<tr>
<td>(1 + 2 + (3 + 4)) + 5</td>
<td>reduce: $E \rightarrow \text{number}$</td>
<td></td>
</tr>
</tbody>
</table>

$S \rightarrow S + E \mid E$

$E \rightarrow \text{number} \mid (S)$
Simple LR parsing with no look ahead

LR(0) GRAMMARS
LR Parser States

• Goal: Know what set of reductions are legal at any given point
• Idea: Summarize all possible stack prefixes $\alpha$ as a finite parser state
  – Parser state is computed by a DFA that reads the stack $\sigma$
  – Accept states of the DFA correspond to unique reductions that apply

• Example: LR(0) parsing
  – **Left-to-right scanning**, **Right-most derivation**, **zero** look-ahead tokens
  – Too weak to handle many language grammars (e.g. the “sum” grammar)
  – But, helpful for understanding how shift-reduce parsers work
Example LR(0) Grammar: Tuples

- Example grammar for non-empty tuples and identifiers

\[
\begin{align*}
S & \rightarrow ( L ) \mid \text{id} \\
L & \rightarrow S \mid L , S
\end{align*}
\]

- Example strings
  - x
  - (x, y)
  - (((x))))
  - (x, (y, z), w)
  - (x, (y, (z, w)))

Parse tree for: (x, (y, z), w)
Shift/Reduce Parsing

• Parser state
  – Stack of terminals and nonterminals
  – Unconsumed input is a string of terminals
  – Current derivation step is stack + input

• Parsing is a sequence of shift and reduce operations

• **Shift**: Move look-ahead token to the stack

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x, (y, z), w)</td>
<td>(x, (y, z), w)</td>
<td>shift (</td>
</tr>
<tr>
<td>(</td>
<td>x, (y, z), w)</td>
<td>shift x</td>
</tr>
</tbody>
</table>

• **Reduce**: Replace symbols γ at top of stack with nonterminal X s.t.
  \( X \rightarrow \gamma \) is a production, i.e., pop γ, push X

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x</td>
<td>, (y, z), w)</td>
<td>reduce S \rightarrow id</td>
</tr>
<tr>
<td>(S</td>
<td>, (y, z), w)</td>
<td>reduce L \rightarrow S</td>
</tr>
</tbody>
</table>
Example Run

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(x, (y, z), w)</td>
<td>shift (</td>
</tr>
<tr>
<td>(</td>
<td>x, (y, z), w)</td>
<td>shift x</td>
</tr>
<tr>
<td>(x</td>
<td>, (y, z), w)</td>
<td>reduce S ⟷ id</td>
</tr>
<tr>
<td>(S</td>
<td>, (y, z), w)</td>
<td>reduce L ⟷ S</td>
</tr>
<tr>
<td>(L</td>
<td>, (y, z), w)</td>
<td>shift ,</td>
</tr>
<tr>
<td>(L,</td>
<td>(y, z), w)</td>
<td>shift (</td>
</tr>
<tr>
<td>(L,</td>
<td>y, z), w)</td>
<td>shift y</td>
</tr>
<tr>
<td>(L, y</td>
<td>, z), w)</td>
<td>reduce S ⟷ id</td>
</tr>
<tr>
<td>(L, S</td>
<td>, z), w)</td>
<td>reduce L ⟷ S</td>
</tr>
<tr>
<td>(L,</td>
<td>(L, z), w)</td>
<td>shift ,</td>
</tr>
<tr>
<td>(L,</td>
<td>L, z), w)</td>
<td>shift z</td>
</tr>
<tr>
<td>(L,</td>
<td>L, z), w)</td>
<td>reduce S ⟷ id</td>
</tr>
<tr>
<td>(L,</td>
<td>S), w)</td>
<td>reduce L ⟷ L, S</td>
</tr>
<tr>
<td>(L,</td>
<td>L), w)</td>
<td>shift )</td>
</tr>
<tr>
<td>(L,</td>
<td>L), w)</td>
<td>reduce S ⟷ ( L )</td>
</tr>
<tr>
<td>(L,</td>
<td>S), w)</td>
<td>reduce L ⟷ L, S</td>
</tr>
<tr>
<td>(L</td>
<td>, w)</td>
<td>shift ,</td>
</tr>
<tr>
<td>(L,</td>
<td>w)</td>
<td>shift w</td>
</tr>
<tr>
<td>(L, w</td>
<td>)</td>
<td>reduce S ⟷ id</td>
</tr>
<tr>
<td>(L, S</td>
<td>)</td>
<td>reduce L ⟷ L, S</td>
</tr>
<tr>
<td>(L</td>
<td>)</td>
<td>shift )</td>
</tr>
<tr>
<td>(L)</td>
<td>)</td>
<td>reduce S ⟷ ( L )</td>
</tr>
</tbody>
</table>

S ⟷ ( L ) | id
L ⟷ S | L , S
Action Selection Problem

• Given a stack $\sigma$ and a look-ahead symbol $b$, should the parser
  – Shift $b$ onto the stack (new stack is $\sigma b$), or
  – Reduce a production $X \rightarrow \gamma$, assuming that $\sigma = \alpha \gamma$ (new stack is $\alpha X$)?

• Sometimes the parser can reduce but should not
  – For example, $X \rightarrow \varepsilon$ can always be reduced

• Sometimes the stack can be reduced in different ways

• Main idea: Decide based on a prefix $\alpha$ of the stack plus look-ahead
  – The prefix $\alpha$ is different for different possible reductions since in
    productions $X \rightarrow \gamma$ and $Y \rightarrow \beta$, $\gamma$ and $\beta$ might have different lengths

• Main goal: Know what set of reductions are legal at any point
  – How do we keep track?
LR(0) States

• An LR(0) state is a set of items keeping track of progress on possible upcoming reductions
• An LR(0) item is a production from the language with an extra separator “.” somewhere in the right-hand-side

\[
\begin{align*}
S & \rightarrow ( L ) \mid id \\
L & \rightarrow S \mid L , S
\end{align*}
\]

• Example items:  \( S \rightarrow .( L ) \) or  \( S \rightarrow (. L) \) or  \( L \rightarrow S. \)
• Intuition
  – Stuff before the ‘.’ is already on the stack (beginnings of possible γ’s to be reduced)
  – Stuff after the ‘.’ is what might be seen next
  – The prefixes α are represented by the state itself
Constructing the DFA: Start state & Closure

- First step: Add a new production $S' \rightarrow S\$ to the grammar
- Start state of the DFA = empty stack, so it contains the item: $S' \rightarrow .S\$
- Closure of a state
  - Adds items for all productions whose LHS nonterminal occurs in an item in the state just after the ‘.’
  - The added items have the ‘.’ located at the beginning (no symbols for those items have been added to the stack yet)
  - Note that newly added items may cause yet more items to be added to the state… keep iterating until a fixed point is reached
- Example: $\text{CLOSURE}\{ S' \rightarrow .S\$ \} = \{ S' \rightarrow .S\$, $S \rightarrow .(L), S \rightarrow .id \}$
- Resulting “closed state” contains the set of all possible productions that might be reduced next
Example: Constructing the DFA

- First, we construct a state with the initial item $S' \rightarrow .S$
Example: Constructing the DFA

- Next, we take the closure of that state
  \[ \text{CLOSURE}\{ S' \mapsto .S$ \} = \{ S' \mapsto .S$, S \mapsto .( L ), S \mapsto .id \} \]

- In the set of items, the nonterminal S appears after the ‘.’
- So we add items for each S production in the grammar
Example: Constructing the DFA

Next we add the transitions:

- First, we see what terminals and nonterminals can appear after the ‘.’ in the source state.
  - Outgoing edges have those labels.
- The target state (initially) includes all items from the source state that have the edge-label symbol after the ‘.’, but we advance the ‘.’ (to simulate shifting the item onto the stack).

S' \rightarrow S$
S \rightarrow (L) | id
L \rightarrow S | L, S
Example: Constructing the DFA

\[ S' \rightarrow .S$ \]
\[ S \rightarrow .( L ) \]
\[ S \rightarrow .id \]

\[ S' \rightarrow S.$ \]
\[ S \rightarrow .( L ) \]
\[ L \rightarrow .S \]
\[ L \rightarrow .L, S \]
\[ S \rightarrow .(L) \]
\[ S \rightarrow .id \]

- Finally, for each new state, we take the closure
- Note that we have to perform two iterations to compute CLOSURE({S \rightarrow ( . L ))
  - First iteration adds L \rightarrow .S and L \rightarrow .L, S
  - Second iteration adds S \rightarrow .(L) and S \rightarrow .id
Full DFA for the Example

- Current state: run the DFA on the stack
- If a reduce state is reached, reduce
- Otherwise, if the next token matches an outgoing edge, shift
- If no such transition, it is a parse error

Reduce state: ‘.’ at the end of the production
Using the DFA

• Run the parser stack through the DFA
• The resulting state tells us which productions might be reduced next
  – If not in a reduce state, then shift the next symbol and transition according to DFA
  – If in a reduce state, $X \xrightarrow{\gamma}$ with stack $\alpha\gamma$, pop $\gamma$ and push $X$

• Optimization: No need to re-run the DFA from beginning every step
  – Store the state with each symbol on the stack: e.g. $1(3(3L5)_6$
  – On a reduction $X \xrightarrow{\gamma}$, pop stack to reveal the state too e.g., from stack $1(3(3L5)_6$ reduce $S \xrightarrow{(L)}$ to reach stack $1(3$
  – Next, push the reduction symbol: e.g. to reach stack $1(3S$
  – Then take just one step in the DFA to find next state: $1(3S_7$
Implementing the Parsing Table

Represent the DFA as a table of shape
\[ \text{state} \times (\text{terminals} + \text{nonterminals}) \]

- Entries for the “action table” specify two kinds of actions
  - Shift and go to state \( n \)
  - Reduce using reduction \( X \rightarrow \gamma \)
    - First pop \( \gamma \) off the stack to reveal the state
    - Look up \( X \) in the “goto table” and go to that state

<table>
<thead>
<tr>
<th>State</th>
<th>Terminal Symbols</th>
<th>Nonterminal Symbols</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Action table</td>
<td>Goto table</td>
</tr>
</tbody>
</table>
### Example Parse Table

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>s3</td>
<td></td>
<td>s2</td>
<td></td>
<td></td>
<td>g4</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>S→id</td>
<td>S→id</td>
<td>S→id</td>
<td>S→id</td>
<td>S→id</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>s3</td>
<td></td>
<td>s2</td>
<td></td>
<td></td>
<td>g7</td>
<td>g5</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DONE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>s6</td>
<td></td>
<td>s8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>S → (L)</td>
<td>S → (L)</td>
<td>S → (L)</td>
<td>S → (L)</td>
<td>S → (L)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>L → S</td>
<td>L → S</td>
<td>L → S</td>
<td>L → S</td>
<td>L → S</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>s3</td>
<td></td>
<td>s2</td>
<td></td>
<td></td>
<td>g9</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>L → L,S</td>
<td>L → L,S</td>
<td>L → L,S</td>
<td>L → L,S</td>
<td>L → L,S</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

sx = shift and go to state x
gx = go to state x
### Example

- Parse the token stream: $(x, (y, z), w)$

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action (according to table)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\varepsilon_1$</td>
<td>$(x, (y, z), w)$</td>
<td>s3</td>
</tr>
<tr>
<td>$\varepsilon_1(3$</td>
<td>$x, (y, z), w)$</td>
<td>s2</td>
</tr>
<tr>
<td>$\varepsilon_1(3x_2$</td>
<td>$, (y, z), w)$</td>
<td>Reduce: $S \rightarrow id$</td>
</tr>
<tr>
<td>$\varepsilon_1(3S$</td>
<td>$, (y, z), w)$</td>
<td>g7 (from state 3 follow $S$)</td>
</tr>
<tr>
<td>$\varepsilon_1(3S_7$</td>
<td>$, (y, z), w)$</td>
<td>Reduce: $L \rightarrow S$</td>
</tr>
<tr>
<td>$\varepsilon_1(3L$</td>
<td>$, (y, z), w)$</td>
<td>g5 (from state 3 follow $L$)</td>
</tr>
<tr>
<td>$\varepsilon_1(3L_5$</td>
<td>$, (y, z), w)$</td>
<td>s8</td>
</tr>
<tr>
<td>$\varepsilon_1(3L_5,8$</td>
<td>$(y, z), w)$</td>
<td>s3</td>
</tr>
<tr>
<td>$\varepsilon_1(3L_5,8(3$</td>
<td>$y, z), w)$</td>
<td>s2</td>
</tr>
</tbody>
</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

<table>
<thead>
<tr>
<th>OK</th>
<th>shift/reduce</th>
<th>reduce/reduce</th>
</tr>
</thead>
<tbody>
<tr>
<td>S → (L).</td>
<td>S → (L).</td>
<td>S → L ,S.</td>
</tr>
<tr>
<td>L → .L , S</td>
<td>S → ,S.</td>
<td></td>
</tr>
</tbody>
</table>

• Such conflicts can often be resolved by using a single look-ahead symbol: LR(1)
Examples

- Consider the left associative and right associative “sum” grammars

  left
  \[
  S \rightarrow S + E \mid E \\
  E \rightarrow \text{number} \mid ( S )
  \]

  right
  \[
  S \rightarrow E + S \mid E \\
  E \rightarrow \text{number} \mid ( S )
  \]

- One is LR(0) the other is not. Which is which, and why?
- What kind of conflict do we get?
  - shift/reduce, or
  - reduce/reduce?

- Ambiguities in associativity/precedence often lead to shift/reduce conflicts
LR(1) Parsing

- Algorithm is similar to LR(0) DFA construction
  - LR(1) state = set of LR(1) items
  - An LR(1) item is an LR(0) item + a set of look-ahead symbols
    \[ A \rightarrow \alpha \beta \cdot \mathcal{L} \]

- LR(1) closure is a little more complex
- Form the set of items just as for LR(0) algorithm
- Whenever a new item \( C \rightarrow \cdot \gamma \) is added because \( A \rightarrow \beta \cdot C \delta \cdot \mathcal{L} \) is already in the set, we need to compute its look-ahead set \( \mathcal{M} \)
  1. The look-ahead set \( \mathcal{M} \) includes FIRST(\( \delta \))
     (the set of terminals that may start strings derived from \( \delta \))
  2. If \( \delta \) is or can derive \( \epsilon \), then the look-ahead \( \mathcal{M} \) also contains \( \mathcal{L} \)
Example Closure

\[
\begin{align*}
S' & \rightarrow S$ \\
S & \rightarrow E + S \mid E \\
E & \rightarrow \text{number} \mid ( S )
\end{align*}
\]

- Start item: \( S' \rightarrow .S$ \ , {} \)
- Since \( S \) is to the right of a '..', add
  \[
  \begin{align*}
  S & \rightarrow .E + S \ , \ {$} \quad \text{Note: \{${}\} \text{ is FIRST($)} \\
  S & \rightarrow .E \ , \ {$}
  \end{align*}
  \]
- Need to keep closing, since \( E \) appears to the right of a '..' in '.E + S':
  \[
  \begin{align*}
  E & \rightarrow .\text{number} \ , \ {+} \quad \text{Note: + added for reason 1} \\
  E & \rightarrow .( S ) \ , \ {+} \quad \text{FIRST(+ S) = \{+\}}
  \end{align*}
  \]
- Because \( E \) also appears to the right of '..' in '.E' we get:
  \[
  \begin{align*}
  E & \rightarrow .\text{number} \ , \ {$} \quad \text{Note: $ added for reason 2} \\
  E & \rightarrow .( S ) \ , \ {$}
  \end{align*}
  \]
- All items are distinct, so we're done
The behavior is determined if:
- There is no overlap among the look-ahead sets for each reduce item, and
- None of the look-ahead symbols appear to the right of a ‘.’
LR(1) issues

• LR(1) gives maximal power out of a 1 look-ahead symbol parsing table
  – DFA + stack is a push-down automaton

• In practice, LR(1) tables are big
  – Modern implementations (e.g. menhir) directly generate code
LR Variants: LALR(1) & GLR

• Consider for example the LR(1) states
  \{[X \rightarrow \alpha \bullet, a], [Y \rightarrow \beta \bullet, c]\}
  \{[X \rightarrow \alpha \bullet, b], [Y \rightarrow \beta \bullet, d]\}
• They have the same core and can be merged
• And the merged state contains
  \{[X \rightarrow \alpha \bullet, a/b], [Y \rightarrow \beta \bullet, c/d]\}
• These are called LALR(1) states
  – Stands for LookAhead LR
  – Typically 10 times fewer LALR(1) states than LR(1)
• Compared to LR(1), LALR(1) may introduce new reduce/reduce conflicts, but not new shift/reduce conflicts. Why?

• GLR = “Generalized LR” parsing
  – Efficiently compute the set of all parses for a given input
  – Later passes should disambiguate based on other context
Classification of Grammars

LR(1)
LALR(1)
LL(1)
SLR
LR(0)
DEBUGGING PARSER CONFLICTS.
DISAMBIGUATING GRAMMARS.

MENHIR IN PRACTICE
Practical Issues

• Dealing with source file location information
  – In the lexer and parser
  – In the abstract syntax
    – See range.ml, ast.ml

• Lexing comments / strings
Menhir output

• 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
Precedence and Associativity Declarations

- Parser generators, like menhir often support precedence and associativity declarations
  - Hints to the parser about how to resolve conflicts
  - See: good-parser.mly

- Pros
  - Avoids having to manually resolve those ambiguities by manually introducing extra nonterminals (as seen in parser.mly)
  - Easier to maintain the grammar

- Cons
  - Can’t as easily re-use the same terminal (if associativity differs)
  - Introduces another level of debugging

- Limits
  - Not always easy to disambiguate the grammar based on just precedence and associativity
Example Ambiguity in Real Languages

• Consider this grammar
  \[ S \rightarrow \text{if } (E) \text{ S} \]
  \[ S \rightarrow \text{if } (E) \text{ S else } S \]
  \[ S \rightarrow \text{X } = \text{ E} \]
  \[ E \rightarrow \ldots \]

• Is this grammar OK?

• Consider how to parse
  \[ \text{if } (E_1) \text{ if } (E_2) S_1 \]
  \[ \text{else } S_2 \]

• This is known as the “dangling else” problem.
• What should the “right” answer be?

• How do we change the grammar?
How to Disambiguate if-then-else

• Want to rule out

\[
\text{if } (E_1) \begin{cases} \text{if } (E_2) \ S_1 \end{cases} \ \text{else} \ S_2
\]

• Observation: An un-matched ‘if’ should not appear as the ‘then’ clause of a containing ‘if’

\[
S \rightarrow M \mid U \quad \text{// } M = \text{“matched”}, \ U = \text{“unmatched”}
\]
\[
U \rightarrow \text{if } (E) \ S \quad \text{// Unmatched ‘if’}
\]
\[
U \rightarrow \text{if } (E) \ M \text{ else } U \quad \text{// Nested if is matched}
\]
\[
M \rightarrow \text{if } (E) \ M \text{ else } M \quad \text{// Matched ‘if’}
\]
\[
M \rightarrow X = E \quad \text{// Other statements}
\]

• See: else-resolved-parser.mly
Alternative: Use { }

• Ambiguity arises because the ‘then’ branch is not well bracketed

\[
\begin{align*}
&\text{if } (E_1) \{ \text{if } (E_2) \{ S_1 \} \} \text{ else } S_2 \quad \text{ // unambiguous} \\
&\text{if } (E_1) \{ \text{if } (E_2) \{ S_1 \} \text{ else } S_2 \} \quad \text{ // unambiguous}
\end{align*}
\]

• So, one could just require brackets
  – But requiring them for the else clause too leads to ugly code for chained if-statements

```c
if (c1) {
  ...
} else {
  if (c2) {
    }
  } else {
    if (c3) {
    } else {
    }
}
```

So, compromise? Allow unbracketed else block only if the body is ‘if’

```c
if (c1) {
  ...
} else if (c2) {
    } else if (c3) {
  } else {
  }
```

Benefits
  • Less ambiguous
  • Easy to parse
  • Enforces good style