Announcements

• Midterm Exam: THURSDAY March 2\textsuperscript{nd} in class!
  – Coverage: x86 / calling conventions / IRs / LLVM / Lexing / Parsing
  – Note: example exams covered more topics

  * Dr. Zdancewic will be out of town on the exam day

• HW4: Parsing & basic code generation
  – Available soon (i.e. before Spring break)
  – Due: Tuesday, March 21\textsuperscript{st}
Simple LR parsing with no look ahead.

LR(0) GRAMMARS
Example LR(0) Grammar: Tuples

- Example grammar for non-empty tuples and identifiers:

  \[
  S \rightarrow (L) \mid id
  \]

  \[
  L \rightarrow S \mid L, S
  \]

- Example strings:
  - x
  - (x, y)
  - (((x)))
  - (x, (y, z), w)
  - (x, (y, (z, w)))
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 the shift-reduce parser works.
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: e.g.

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x, (y, z), w)</td>
<td></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 such that \( X \vdash \gamma \) is a production. (pop γ, push X): e.g.

<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 \vdash id</td>
</tr>
<tr>
<td>(S</td>
<td>, (y, z), w)</td>
<td>reduce L \vdash S</td>
</tr>
</tbody>
</table>
### Example Run

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

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

• Sometimes the parser can reduce but shouldn’t
  – For example, \( X \rightarrow \epsilon \) can always be reduced

• Sometimes the stack can be reduced in different ways

• Main idea: decide what to do based on a prefix \( \alpha \) of the stack plus the look-ahead symbol.
  – The prefix \( \alpha \) is different for different possible reductions since in productions \( X \rightarrow \gamma \) and \( Y \rightarrow \beta \) the \( \gamma \) 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

- 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 $\alpha$ are represented by the state itself
Example: Constructing the DFA

- First, we construct a state with the initial item $S' \mapsto .S$

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

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 label.
- 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)
Example: Constructing the DFA

Finally, for each new state, we take the closure.

Note that we have to perform two iterations to compute
\text{CLOSURE}\left(\{S \rightarrow ( . L )\}\right)

- 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

1. S’ ⟷ .S$
2. S ⟷ id.
3. S ⟷ (. L )
4. L ⟷ .S
5. S ⟷ .(L)
6. S ⟷ .id
7. S ⟷ (L).
8. L ⟷ L, . S
9. S ⟷ L, S.

- 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, $X \rightarrow \gamma$ 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 $X \rightarrow \gamma$, pop stack to reveal the state too:
    e.g. From stack $1(3(3L5)_6$ reduce $S \rightarrow ( L )$ 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 goto state \( n \)
  - Reduce using reduction \( X \rightarrow \gamma \)
    - First pop \( \gamma \) off the stack to reveal the state \( s \)
    - Look up \( X \) in the “goto table” for the state \( s \) and goto the target state
- Missing table entries = parse errors

<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

<table>
<thead>
<tr>
<th></th>
<th>(</th>
<th>)</th>
<th>id</th>
<th>,</th>
<th>$</th>
<th>S</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>s3</td>
<td>s2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>g4</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>s2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>g7</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DONE</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>s6</td>
<td>s8</td>
<td></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>s2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>g9</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 goto state x  
gx = goto state x
### Example

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

<table>
<thead>
<tr>
<th>Stack</th>
<th>Stream</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 \text{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 $\rightarrow$ ( L ).</td>
<td>S $\rightarrow$ ( L ).</td>
<td></td>
</tr>
<tr>
<td>L $\rightarrow$.L , S</td>
<td>L $\rightarrow$.L , S</td>
<td></td>
</tr>
<tr>
<td>S $\rightarrow$ L ,S.</td>
<td>S $\rightarrow$ L ,S.</td>
<td></td>
</tr>
<tr>
<td>S $\rightarrow$ ,S.</td>
<td>S $\rightarrow$ ,S.</td>
<td></td>
</tr>
</tbody>
</table>

• Such conflicts can often be resolved by using a 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 isn’t… which is which and why?

• What kind of conflict do you get? Shift/reduce or Reduce/reduce?

• Ambiguities in associativity/precedence usually 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:
    \[ A \rightarrow \alpha \beta , \ L \]

• LR(1) closure is a little more complex:
• Form the set of items just as for LR(0) algorithm.
• Whenever a new item \( C \rightarrow \gamma \) is added because \( A \rightarrow \beta . C \delta , \ L \) 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 \( \delta \) can derive \( \epsilon \) (it is nullable), then the look-ahead \( \mathcal{M} \) also contains \( \mathcal{L} \)
Example Closure

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

- Start item: \( S' \longrightarrow .S$ \ , \ {} \)
- Since \( S \) is to the right of a \\( \cdot \), add:
  \[
  \begin{align*}
  S & \longrightarrow .E + S \ , \ \{ \}$ \\
  S & \longrightarrow .E \ , \ \{ \}$
  \end{align*}
  \]
  Note: \( \{ \} \) is FIRST($$)
- Need to keep closing, since \( E \) appears to the right of a \\( \cdot \) in \( .E + S' \):
  \[
  \begin{align*}
  E & \longrightarrow .\text{number} \ , \ \{ + \} \\
  E & \longrightarrow .( S ) \ , \ \{ + \}
  \end{align*}
  \]
- Because \( E \) also appears to the right of \\( \cdot \) in \( .E \) we get:
  \[
  \begin{align*}
  E & \longrightarrow .\text{number} \ , \ \{ \}$ \\
  E & \longrightarrow .( S ) \ , \ \{ \}$
  \end{align*}
  \]
- 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 variants

- LR(1) gives maximal power out of a 1 look-ahead symbol parsing table
  - DFA + stack is a push-down automaton (recall 262)
- In practice, LR(1) tables are big.
  - Modern implementations (e.g. menhir) directly generate code

- LALR(1) = “Look-ahead LR”
  - Merge any two LR(1) states whose items are identical except for the look-ahead sets:

    \[
    
    S' \rightarrow .S$ \quad \{\}
    
    S \rightarrow .E + S \quad \{$
    
    S \rightarrow .E \quad \{$
    
    E \rightarrow .num \quad \{+$
    
    E \rightarrow .( S ) \quad \{+$
    
    E \rightarrow .num \quad \{$
    
    E \rightarrow .( S ) \quad \{$
    
    \]
  - Such merging can lead to nondeterminism (e.g. reduce/reduce conflicts), but
  - Results in a much smaller parse table and works well in practice
  - This is the usual technology for automatic parser generators: yacc, ocamlyacc

- 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.
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:
    
    ```bash
    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:
  $S \rightarrow \text{if } (E) \ S$
  $S \rightarrow \text{if } (E) \ S \ \text{else} \ S$
  $S \rightarrow X = E$
  $E \rightarrow \ldots$

• 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:

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

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

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

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

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

```plaintext
if (E₁) { if (E₂) { S₁ } } else S₂ // unambiguous
if (E₁) { if (E₂) { S₁ } else S₂ } // unambiguous
```

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

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

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

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

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