Announcements

• Reminder: HW3 LLVM backend
  – Due: TONIGHT!

• Midterm Exam: March 2nd 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
  – Due: After break
Searching for derivations.
A Context-free Grammar (CFG) consists of:

- A set of *terminals* (e.g., a token or $\varepsilon$)
- A set of *nonterminals* (e.g., $S$ and other syntactic variables)
- A designated nonterminal called the *start symbol*.
- A set of productions: $\text{LHS} \rightarrow \text{RHS}$
  - LHS is a nonterminal
  - RHS is a *string* of terminals and nonterminals

Example: The balanced parentheses language:

- $S \rightarrow (S)S$
- $S \rightarrow \varepsilon$

How many terminals? How many nonterminals? Productions?
Consider finding left-most derivations

- Look at only one input symbol at a time.

S \Rightarrow E + S \mid E
E \Rightarrow \text{number} \mid ( S )

<table>
<thead>
<tr>
<th>Partly-derived String</th>
<th>Look-ahead</th>
<th>Parsed/Unparsed Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>(</td>
<td>(1 + 2 + (3 + 4)) + 5</td>
</tr>
<tr>
<td>\Rightarrow E + S</td>
<td>(</td>
<td>(1 + 2 + (3 + 4)) + 5</td>
</tr>
<tr>
<td>\Rightarrow (S) + S</td>
<td>1</td>
<td>(1 + 2 + (3 + 4)) + 5</td>
</tr>
<tr>
<td>\Rightarrow (E + S) + S</td>
<td>1</td>
<td>(1 + 2 + (3 + 4)) + 5</td>
</tr>
<tr>
<td>\Rightarrow (1 + S) + S</td>
<td>2</td>
<td>(1 + 2 + (3 + 4)) + 5</td>
</tr>
<tr>
<td>\Rightarrow (1 + E + S) + S</td>
<td>2</td>
<td>(1 + 2 + (3 + 4)) + 5</td>
</tr>
<tr>
<td>\Rightarrow (1 + 2 + S) + S</td>
<td>(</td>
<td>(1 + 2 + (3 + 4)) + 5</td>
</tr>
<tr>
<td>\Rightarrow (1 + 2 + E) + S</td>
<td>(</td>
<td>(1 + 2 + (3 + 4)) + 5</td>
</tr>
<tr>
<td>\Rightarrow (1 + 2 + (S)) + S</td>
<td>3</td>
<td>(1 + 2 + (3 + 4)) + 5</td>
</tr>
<tr>
<td>\Rightarrow (1 + 2 + (E + S)) + S</td>
<td>3</td>
<td>(1 + 2 + (3 + 4)) + 5</td>
</tr>
<tr>
<td>\Rightarrow ...</td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>
There is a problem

- We want to decide which production to apply based on the look-ahead symbol.
- But, there is a choice:

  $S \rightarrow E \rightarrow (S) \rightarrow (E) \rightarrow (1)$

  vs.

  $S \rightarrow E + S \rightarrow (S) + S \rightarrow (E) + S \rightarrow (1) + S \rightarrow (1) + E \rightarrow (1) + 2$

- Given the look-ahead symbol: ‘(‘ it isn’t clear whether to pick $S \rightarrow E$ or $S \rightarrow E + S$ first.
LL(1) GRAMMARS
Grammar is the problem

- Not all grammars can be parsed “top-down” with only a single lookahead symbol.
- **Top-down**: starting from the start symbol (root of the parse tree) and going down

- LL(1) means
  - **Left-to-right scanning**
  - **Left-most derivation**,  
  - 1 lookahead symbol

- This language isn’t “LL(1)”
- Is it LL(k) for some k?

- What can we do?
Making a grammar LL(1)

- **Problem:** We can’t decide which S production to apply until we see the symbol after the first expression.
- **Solution:** “Left-factor” the grammar. There is a common S prefix for each choice, so add a new non-terminal S’ at the decision point:

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

- Also need to eliminate left-recursion somehow. Why?
- Consider:

\[
\begin{align*}
S & \rightarrow S + E \mid E \\
E & \rightarrow \text{number} \mid (S)
\end{align*}
\]
LL(1) Parse of the input string

- Look at only one input symbol at a time.

<table>
<thead>
<tr>
<th>Partly-derived String</th>
<th>Look-ahead</th>
<th>Parsed/Unparsed Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
<td>(</td>
<td>$(1 + 2 + (3 + 4)) + 5$</td>
</tr>
<tr>
<td>$\Rightarrow E S'$</td>
<td>(</td>
<td>$(1 + 2 + (3 + 4)) + 5$</td>
</tr>
<tr>
<td>$\Rightarrow (S) S'$</td>
<td>1</td>
<td>$(1 + 2 + (3 + 4)) + 5$</td>
</tr>
<tr>
<td>$\Rightarrow (E S') S'$</td>
<td>1</td>
<td>$(1 + 2 + (3 + 4)) + 5$</td>
</tr>
<tr>
<td>$\Rightarrow (1 S') S'$</td>
<td>+</td>
<td>$(1 + 2 + (3 + 4)) + 5$</td>
</tr>
<tr>
<td>$\Rightarrow (1 + S) S'$</td>
<td>2</td>
<td>$(1 + 2 + (3 + 4)) + 5$</td>
</tr>
<tr>
<td>$\Rightarrow (1 + E S') S'$</td>
<td>2</td>
<td>$(1 + 2 + (3 + 4)) + 5$</td>
</tr>
<tr>
<td>$\Rightarrow (1 + 2 S') S'$</td>
<td>+</td>
<td>$(1 + 2 + (3 + 4)) + 5$</td>
</tr>
<tr>
<td>$\Rightarrow (1 + 2 + S) S'$</td>
<td>(</td>
<td>$(1 + 2 + (3 + 4)) + 5$</td>
</tr>
<tr>
<td>$\Rightarrow (1 + 2 + E S') S'$</td>
<td>(</td>
<td>$(1 + 2 + (3 + 4)) + 5$</td>
</tr>
<tr>
<td>$\Rightarrow (1 + 2 + (S)S') S'$</td>
<td>3</td>
<td>$(1 + 2 + (3 + 4)) + 5$</td>
</tr>
</tbody>
</table>
Predictive Parsing

- Given an LL(1) grammar:
  - For a given nonterminal, the lookahead symbol uniquely determines the production to apply.
  - Top-down parsing = predictive parsing
  - Driven by a predictive parsing table:
    nonterminal * input token $\rightarrow$ production

<table>
<thead>
<tr>
<th></th>
<th>number</th>
<th>+</th>
<th>(</th>
<th>)</th>
<th>$ \ (EOF)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>$\rightarrow$ S$</td>
<td></td>
<td></td>
<td>$\rightarrow$ S$</td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>$\rightarrow$ E S’</td>
<td>$\rightarrow$ E S’</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S’</td>
<td>$\rightarrow$ + S</td>
<td>$\rightarrow$ $\varepsilon$</td>
<td>$\rightarrow$ $\varepsilon$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>$\rightarrow$ num.</td>
<td>$\rightarrow$ ( S )</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Note: it is convenient to add a special end-of-file token $\$ and a start symbol T (top-level) that requires $\$. 
How do we construct the parse table?

• Consider a given production: \( A \rightarrow \gamma \)
• Construct the set of all input tokens that may appear *first* in strings that can be derived from \( \gamma \)
  – Add the production \( \rightarrow \gamma \) to the entry (A, token) for each such token.
• If \( \gamma \) can derive \( \varepsilon \) (the empty string), then we construct the set of all input tokens that may *follow* the nonterminal A in the grammar.
  – Add the production \( \rightarrow \gamma \) to the entry (A, token) for each such token.

• Note: if there are two different productions for a given entry, the grammar is not LL(1)
• First(T) = First(S)
• First(S) = First(E)
• First(S’) = { + }
• First(E) = { number, ‘(‘ }

• Follow(S’) = Follow(S)
• Follow(S) = { $, ‘)’ } \cup Follow(S’)

Note: we want the least solution to this system of set equations… a fixpoint computation. More on these later in the course.

<table>
<thead>
<tr>
<th></th>
<th>number</th>
<th>+</th>
<th>(</th>
<th>)</th>
<th>$ (EOF)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>\rightarrow S$</td>
<td></td>
<td>\rightarrow S$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>\rightarrow E S’</td>
<td></td>
<td>\rightarrow E S’</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S’</td>
<td></td>
<td>\rightarrow + S</td>
<td></td>
<td>\rightarrow \epsilon</td>
<td>\rightarrow \epsilon</td>
</tr>
<tr>
<td>E</td>
<td>\rightarrow num.</td>
<td></td>
<td>\rightarrow ( S )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Converting the table to code

• Define \( n \) mutually recursive functions
  – one for each nonterminal \( A \): \( \text{parse}_A \)
  – The type of \( \text{parse}_A \) is \( \text{unit} \rightarrow \text{ast} \) if \( A \) is not an auxiliary nonterminal
  – Parse functions for auxiliary nonterminals (e.g. \( S' \)) take extra ast’s as inputs, one for each nonterminal in the “factored” prefix.

• Each function “peeks” at the lookahead token and then follows the production rule in the corresponding entry.
  – Consume terminal tokens from the input stream
  – Call \( \text{parse}_X \) to create sub-tree for nonterminal \( X \)
  – If the rule ends in an auxiliary nonterminal, call it with appropriate ast’s. (The auxiliary rule is responsible for creating the ast after looking at more input.)
  – Otherwise, this function builds the ast tree itself and returns it.
Hand-generated LL(1) code for the table above.

<table>
<thead>
<tr>
<th></th>
<th>number</th>
<th>+</th>
<th>(</th>
<th>)</th>
<th>$ (EOF)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>$T \rightarrow S S$</td>
<td></td>
<td>$T \rightarrow S S$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>$T \rightarrow E S'$</td>
<td></td>
<td>$T \rightarrow E S'$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S'</td>
<td>$T \rightarrow E S'$</td>
<td>$T \rightarrow E S'$</td>
<td></td>
<td>$T \rightarrow \epsilon$</td>
<td>$T \rightarrow \epsilon$</td>
</tr>
<tr>
<td>E</td>
<td>$T \rightarrow \text{num.}$</td>
<td></td>
<td></td>
<td>$T \rightarrow (S)$</td>
<td></td>
</tr>
</tbody>
</table>

**DEMO: PARSER.ML**
LL(1) Summary

• Top-down parsing that finds the leftmost derivation.
• Language Grammar ⇒ LL(1) grammar ⇒ prediction table ⇒ recursive-descent parser

• Problems:
  – Grammar must be LL(1)
  – Can extend to LL(k) (it just makes the table bigger)
  – Grammar cannot be left recursive (parser functions will loop!)

• Is there a better way?
LR GRAMMARS
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, merlin, etc.)
  – Better error detection/recovery
• 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…

Top-down vs. Bottom up

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

<table>
<thead>
<tr>
<th>Reductions</th>
<th>Scanned</th>
<th>Input Remaining</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1 + 2 + (3 + 4)) + 5 ←</td>
<td>(</td>
<td>(1 + 2 + (3 + 4)) + 5</td>
</tr>
<tr>
<td>(E + 2 + (3 + 4)) + 5 ←</td>
<td>(1</td>
<td>+ 2 + (3 + 4)) + 5</td>
</tr>
<tr>
<td>(S + 2 + (3 + 4)) + 5 ←</td>
<td>(1 + 2</td>
<td>+ 2 + (3 + 4)) + 5</td>
</tr>
<tr>
<td>(S + E + (3 + 4)) + 5 ←</td>
<td>(1 + 2</td>
<td>+ (3 + 4)) + 5</td>
</tr>
<tr>
<td>(S + (3 + 4)) + 5 ←</td>
<td>(1 + 2</td>
<td>+ (3 + 4)) + 5</td>
</tr>
<tr>
<td>(S + (E + 4)) + 5 ←</td>
<td>(1 + 2 + (3</td>
<td>+ 4)) + 5</td>
</tr>
<tr>
<td>(S + (S + 4)) + 5 ←</td>
<td>(1 + 2 + (3</td>
<td>+ 4)) + 5</td>
</tr>
<tr>
<td>(S + (S + E)) + 5 ←</td>
<td>(1 + 2 + (3 + 4)</td>
<td>) + 5</td>
</tr>
<tr>
<td>(S + (S)) + 5 ←</td>
<td>(1 + 2 + (3 + 4)</td>
<td>) + 5</td>
</tr>
<tr>
<td>(S + E) + 5 ←</td>
<td>(1 + 2 + (3 + 4)</td>
<td>) + 5</td>
</tr>
<tr>
<td>(S) + 5 ←</td>
<td>(1 + 2 + (3 + 4)</td>
<td>) + 5</td>
</tr>
<tr>
<td>E + 5 ←</td>
<td>(1 + 2 + (3 + 4)</td>
<td>+ 5</td>
</tr>
<tr>
<td>S + 5 ←</td>
<td>(1 + 2 + (3 + 4)</td>
<td>+ 5</td>
</tr>
<tr>
<td>S + E ←</td>
<td>(1 + 2 + (3 + 4)</td>
<td>+ 5</td>
</tr>
<tr>
<td>S</td>
<td>(1 + 2 + (3 + 4)</td>
<td>+ 5</td>
</tr>
</tbody>
</table>

S → S + E | E
E → number | ( 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 *shift* and *reduce* operations:
- **Shift:** move look-ahead token to the stack
- **Reduce:** Replace symbols \( \gamma \) at top of stack with nonterminal \( X \) such that \( X \rightarrow \gamma \) is a production. (pop \( \gamma \), push \( X \))

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

S \( \rightarrow S + E \mid E \)
E \( \rightarrow \text{number} \mid ( \text{S} ) \)
Simple LR parsing with no look ahead.
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.
Example LR(0) Grammar: Tuples

• Example grammar for non-empty tuples and identifiers:

\[
S \rightarrow (L) \mid \text{id}
\]
\[
L \rightarrow S \mid L, S
\]

• Example strings:
  
  – x
  – (x, y)
  – (((x)))
  – (x, (y, z), w)
  – (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: e.g.
    | Stack     | Input        | Action |
    |------------|--------------|--------|
    | (x, (y, z), w) | shift (     |
    | (x, (y, z), w) | shift x     |

  - **Reduce:** Replace symbols $\gamma$ at top of stack with nonterminal $X$ such that $X \rightarrow \gamma$ is a production. (pop $\gamma$, push $X$): e.g.
    | Stack     | Input        | Action              |
    |------------|--------------|---------------------|
    | (x, (y, z), w) | reduce S $\rightarrow$ id |
    | (S, (y, z), w) | reduce L $\rightarrow$ S    |
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>shift (</td>
<td>(x, (y, z), w)</td>
</tr>
<tr>
<td>(x, (y, z), w)</td>
<td>shift x</td>
<td></td>
</tr>
<tr>
<td>(S, (y, z), w)</td>
<td>reduce S</td>
<td>id</td>
</tr>
<tr>
<td>(L, (y, z), w)</td>
<td>shift ,</td>
<td></td>
</tr>
<tr>
<td>(L, (y, z), w)</td>
<td>shift (</td>
<td>(L, (y, z), w)</td>
</tr>
<tr>
<td>(L, (y, z), w)</td>
<td>shift y</td>
<td></td>
</tr>
<tr>
<td>(L, (y, z), w)</td>
<td>reduce S</td>
<td>id</td>
</tr>
<tr>
<td>(L, (y, z), w)</td>
<td>reduce S</td>
<td>L, S</td>
</tr>
<tr>
<td>(L, (y, z), w)</td>
<td>shift ,</td>
<td></td>
</tr>
<tr>
<td>(L, (y, z), w)</td>
<td>shift z</td>
<td></td>
</tr>
<tr>
<td>(L, (y, z), w)</td>
<td>reduce S</td>
<td>id</td>
</tr>
<tr>
<td>(L, (y, z), w)</td>
<td>reduce L</td>
<td>L, S</td>
</tr>
<tr>
<td>(L, (y, z), w)</td>
<td>shift ,</td>
<td></td>
</tr>
<tr>
<td>(L, (y, z), w)</td>
<td>reduce S</td>
<td>(L)</td>
</tr>
<tr>
<td>(L, (y, z), w)</td>
<td>reduce L</td>
<td>L, S</td>
</tr>
<tr>
<td>(L, (y, z), w)</td>
<td>shift ,</td>
<td></td>
</tr>
<tr>
<td>(L, (y, z), w)</td>
<td>reduce S</td>
<td>id</td>
</tr>
<tr>
<td>(L, (y, z), w)</td>
<td>reduce L</td>
<td>L, S</td>
</tr>
<tr>
<td>(L, (y, z), w)</td>
<td>shift ,</td>
<td></td>
</tr>
<tr>
<td>(L, (y, z), w)</td>
<td>reduce S</td>
<td>id</td>
</tr>
<tr>
<td>(L, (y, z), w)</td>
<td>reduce L</td>
<td>L, S</td>
</tr>
<tr>
<td>(L, (y, z), w)</td>
<td>shift ,</td>
<td>(L, (y, z), w)</td>
</tr>
<tr>
<td>(L, (y, z), w)</td>
<td>reduce S</td>
<td>id</td>
</tr>
<tr>
<td>(L, (y, z), w)</td>
<td>reduce L</td>
<td>L, S</td>
</tr>
<tr>
<td>(L, (y, z), w)</td>
<td>shift ,</td>
<td></td>
</tr>
<tr>
<td>(L, (y, z), w)</td>
<td>reduce S</td>
<td>id</td>
</tr>
<tr>
<td>(L, (y, z), w)</td>
<td>reduce L</td>
<td>L, S</td>
</tr>
<tr>
<td>(L, (y, z), w)</td>
<td>shift ,</td>
<td></td>
</tr>
<tr>
<td>(L, (y, z), w)</td>
<td>reduce S</td>
<td>id</td>
</tr>
<tr>
<td>(L, (y, z), w)</td>
<td>reduce L</td>
<td>L, S</td>
</tr>
<tr>
<td>(L, (y, z), w)</td>
<td>shift ,</td>
<td></td>
</tr>
<tr>
<td>(L, (y, z), w)</td>
<td>reduce S</td>
<td>id</td>
</tr>
<tr>
<td>(L, (y, z), w)</td>
<td>reduce L</td>
<td>L, S</td>
</tr>
<tr>
<td>(L, (y, z), w)</td>
<td>shift ,</td>
<td>(L, (y, z), w)</td>
</tr>
<tr>
<td>(L, (y, z), w)</td>
<td>reduce S</td>
<td>id</td>
</tr>
<tr>
<td>(L, (y, z), w)</td>
<td>reduce L</td>
<td>L, S</td>
</tr>
<tr>
<td>(L, (y, z), w)</td>
<td>shift ,</td>
<td></td>
</tr>
<tr>
<td>(L, (y, z), w)</td>
<td>reduce S</td>
<td>id</td>
</tr>
<tr>
<td>(L, (y, z), w)</td>
<td>reduce L</td>
<td>L, S</td>
</tr>
<tr>
<td>(L, (y, z), w)</td>
<td>shift ,</td>
<td></td>
</tr>
<tr>
<td>(L, (y, z), w)</td>
<td>reduce S</td>
<td>id</td>
</tr>
<tr>
<td>(L, (y, z), w)</td>
<td>reduce L</td>
<td>L, S</td>
</tr>
<tr>
<td>(L, (y, z), w)</td>
<td>shift ,</td>
<td>(L, (y, z), w)</td>
</tr>
<tr>
<td>(L, (y, z), w)</td>
<td>reduce S</td>
<td>id</td>
</tr>
<tr>
<td>(L, (y, z), w)</td>
<td>reduce L</td>
<td>L, S</td>
</tr>
<tr>
<td>(L, (y, z), w)</td>
<td>shift ,</td>
<td>(L, (y, z), w)</td>
</tr>
<tr>
<td>(L, (y, z), w)</td>
<td>reduce S</td>
<td>id</td>
</tr>
<tr>
<td>(L, (y, z), w)</td>
<td>reduce L</td>
<td>L, S</td>
</tr>
<tr>
<td>(L, (y, z), w)</td>
<td>shift ,</td>
<td>(L, (y, z), w)</td>
</tr>
<tr>
<td>(L, (y, z), w)</td>
<td>reduce S</td>
<td>id</td>
</tr>
<tr>
<td>(L, (y, z), w)</td>
<td>reduce L</td>
<td>L, S</td>
</tr>
<tr>
<td>(L, (y, z), w)</td>
<td>shift ,</td>
<td>(L, (y, z), w)</td>
</tr>
<tr>
<td>(L, (y, z), w)</td>
<td>reduce S</td>
<td>id</td>
</tr>
<tr>
<td>(L, (y, z), w)</td>
<td>reduce L</td>
<td>L, S</td>
</tr>
<tr>
<td>(L, (y, z), w)</td>
<td>shift ,</td>
<td>(L, (y, z), w)</td>
</tr>
<tr>
<td>(L, (y, z), w)</td>
<td>reduce S</td>
<td>id</td>
</tr>
<tr>
<td>(L, (y, z), w)</td>
<td>reduce L</td>
<td>L, S</td>
</tr>
<tr>
<td>(L, (y, z), w)</td>
<td>shift ,</td>
<td>(L, (y, z), w)</td>
</tr>
<tr>
<td>(L, (y, z), w)</td>
<td>reduce S</td>
<td>id</td>
</tr>
<tr>
<td>(L, (y, z), w)</td>
<td>reduce L</td>
<td>L, S</td>
</tr>
<tr>
<td>(L, (y, z), w)</td>
<td>shift ,</td>
<td>(L, (y, z), w)</td>
</tr>
<tr>
<td>(L, (y, z), w)</td>
<td>reduce S</td>
<td>id</td>
</tr>
<tr>
<td>(L, (y, z), w)</td>
<td>reduce L</td>
<td>L, S</td>
</tr>
<tr>
<td>(L, (y, z), w)</td>
<td>shift ,</td>
<td>(L, (y, z), w)</td>
</tr>
<tr>
<td>(L, (y, z), w)</td>
<td>reduce S</td>
<td>id</td>
</tr>
<tr>
<td>(L, (y, z), w)</td>
<td>reduce L</td>
<td>L, S</td>
</tr>
<tr>
<td>(L, (y, z), w)</td>
<td>shift ,</td>
<td>(L, (y, z), w)</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, $X \rightarrow \varepsilon$ 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 $Y \rightarrow \beta$, $\gamma$ 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: \( S \mapsto .(L) \) or \( S \mapsto (.L) \) or \( L \mapsto S \).

• Intuition:
  – Stuff before the ‘.’ is already on the stack (beginnings of possible \( \gamma \)’s to be reduced)
  – Stuff after the ‘.’ is what might be seen next
  – The prefixes \( \alpha \) are represented by the state itself

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

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

In the set of items, the nonterminal $S$ appears after the `.'

So we add items for each $S$ production in the grammar

\[
\begin{align*}
S' &\mapsto S \\
S &\mapsto (L) \mid \text{id} \\
L &\mapsto S \mid L, S
\end{align*}
\]
• 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 CLOSURE(\{S \mapsto ( . L )\})
  - First iteration adds L \mapsto .S and L \mapsto .L, S
  - Second iteration adds S \mapsto .(L) and S \mapsto .id
Full DFA for the Example

S' → .S$
S → .( L )
S → .id

S → id.

L → L, . S
S → .( L )
S → .id

L → L, S.

S → ( . L )
L → . S
L → .L, S
S → .( L )
S → .id

S → ( L ).
L → ( L .)
L → 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

Done!
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, \( X \mapsto \gamma \) 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. \( 1(3(3L)5)_6 \)
  – On a reduction \( X \mapsto \gamma \), pop stack to reveal the state too:
    e.g. From stack \( 1(3(3L)5)_6 \) reduce \( S \mapsto ( L ) \) 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
    - Look up \( X \) in the “goto table” and goto that state
### Example Parse Table

<table>
<thead>
<tr>
<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>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>
</tr>
<tr>
<td>3</td>
<td>s3</td>
<td>s2</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>DONE</td>
</tr>
<tr>
<td>5</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>
</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>
</tr>
<tr>
<td>8</td>
<td>s3</td>
<td>s2</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>
</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(3)x_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:

  **OK**
  
  \[
  S \rightarrow ( L ). \]

  **shift/reduce**
  
  \[
  S \rightarrow ( L ). \\
  L \rightarrow .L , S 
  \]

  **reduce/reduce**
  
  \[
  S \rightarrow L ,S. \\
  S \rightarrow ,S. 
  \]

- 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

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

right

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

• 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 $M$:
  1. The look-ahead set $M$ includes FIRST($\delta$) (the set of terminals that may start strings derived from $\delta$)
  2. If $\delta$ can derive $\varepsilon$ (it is nullable), then the look-ahead $M$ also contains $L$
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 \mid \{\}$ \\
  S & \rightarrow .E \mid \{\} \\
  \text{Note: } \{\} \text{ is FIRST($$)}
  \end{align*}
\]
- **Need to keep closing, since E appears to the right of a ‘.’ in ‘E + S’:**
  \[
  \begin{align*}
  E & \rightarrow \text{number} \mid (+) \\
  E & \rightarrow .(S) \mid (+) \\
  \text{Note: } + \text{ added for reason 1}
  \end{align*}
\]
- **Because E also appears to the right of ‘.’ in ‘E’ we get:**
  \[
  \begin{align*}
  E & \rightarrow \text{number} \mid \{\}$ \\
  E & \rightarrow .(S) \mid \{\} \\
  \text{Note: $ added for reason 2}
  \end{align*}
\]
- **All items are distinct, so we’re done**
Using the DFA

1. 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 ‘.’

The fragment of the Action & Goto tables shows the choices between shift and reduce are resolved.
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:

```plaintext
S' → .S$  {}
S → .E + S  {$}
S → .E  {$}
E → .num  {+}
E → .( S )  {+}
E → .num  {$}
E → .( S )  {$}
```

  - 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)