Lecture 12

CIS 341: COMPILERS
Announcements

• **Homework 3:** Compiling LLVMlite

• **Goal:**
  – Familiarize yourself with (a subset of) the LLVM IR
  – Implement a translation down to (inefficient) X86lite

• **Due:** Wednesday, March 4th

---

**MIDTERM EXAM**

– This Thursday, February 27th in class
– Coverage: interpreters / program transformers / x86 / calling conventions / IRs / LLVM / Lexing / Parsing (as of Friday, Feb. 21st)
– See examples on the web pages

---

*it is officially too late to START EARLY!!*
Searching for derivations.
A Context-free Grammar (CFG) consists of
- A set of **terminals** (e.g., a token or ε)
- A set of **nonterminals** (e.g., S and other syntactic variables)
- A designated nonterminal called the **start symbol**
- A set of productions: LHS ⟷ RHS
  - LHS is a nonterminal
  - RHS is a string of terminals and nonterminals

**Example:** The balanced parentheses language:

```
S ⟷ (S)S
S ⟷ ε
```

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

- 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>( \downarrow E + S )</td>
<td>(</td>
<td>((1 + 2 + (3 + 4)) + 5 )</td>
</tr>
<tr>
<td>( \downarrow (S) + S )</td>
<td>1</td>
<td>((1 + 2 + (3 + 4)) + 5 )</td>
</tr>
<tr>
<td>( \downarrow (E + S) + S )</td>
<td>1</td>
<td>((1 + 2 + (3 + 4)) + 5 )</td>
</tr>
<tr>
<td>( \downarrow (1 + S) + S )</td>
<td>2</td>
<td>((1 + 2 + (3 + 4)) + 5 )</td>
</tr>
<tr>
<td>( \downarrow (1 + E + S) + S )</td>
<td>2</td>
<td>((1 + 2 + (3 + 4)) + 5 )</td>
</tr>
<tr>
<td>( \downarrow (1 + 2 + S) + S )</td>
<td>(</td>
<td>((1 + 2 + (3 + 4)) + 5 )</td>
</tr>
<tr>
<td>( \downarrow (1 + 2 + E) + S )</td>
<td>(</td>
<td>((1 + 2 + (3 + 4)) + 5 )</td>
</tr>
<tr>
<td>( \downarrow (1 + 2 + (S)) + S )</td>
<td>3</td>
<td>((1 + 2 + (3 + 4)) + 5 )</td>
</tr>
<tr>
<td>( \downarrow (1 + 2 + (E + S)) + S )</td>
<td>3</td>
<td>((1 + 2 + (3 + 4)) + 5 )</td>
</tr>
<tr>
<td>( \downarrow \ldots )</td>
<td></td>
<td>\ldots</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:

\[
\begin{align*}
(1) & \quad S \rightarrow E \rightarrow (S) \rightarrow (E) \rightarrow (1) \\
\text{vs.} & \quad (1) + 2 \quad S \rightarrow E + S \rightarrow (S) + S \rightarrow (E) + S \rightarrow (1) + S \rightarrow (1) + E \\
& \quad \rightarrow (1) + 2
\end{align*}
\]

- 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?

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

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

- 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>I → E S'</td>
<td>(</td>
<td>(1 + 2 + (3 + 4)) + 5</td>
</tr>
<tr>
<td>I → (S) S'</td>
<td>1</td>
<td>(1 + 2 + (3 + 4)) + 5</td>
</tr>
<tr>
<td>I → (E S') S'</td>
<td>1</td>
<td>(1 + 2 + (3 + 4)) + 5</td>
</tr>
<tr>
<td>I → (1 S') S'</td>
<td>+</td>
<td>(1 + 2 + (3 + 4)) + 5</td>
</tr>
<tr>
<td>I → (1 + S) S'</td>
<td>2</td>
<td>(1 + 2 + (3 + 4)) + 5</td>
</tr>
<tr>
<td>I → (1 + E S') S'</td>
<td>2</td>
<td>(1 + 2 + (3 + 4)) + 5</td>
</tr>
<tr>
<td>I → (1 + 2 S') S'</td>
<td>+</td>
<td>(1 + 2 + (3 + 4)) + 5</td>
</tr>
<tr>
<td>I → (1 + 2 + S) S'</td>
<td>(</td>
<td>(1 + 2 + (3 + 4)) + 5</td>
</tr>
<tr>
<td>I → (1 + 2 + E S') S'</td>
<td>(</td>
<td>(1 + 2 + (3 + 4)) + 5</td>
</tr>
<tr>
<td>I → (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 → production

<table>
<thead>
<tr>
<th></th>
<th>number</th>
<th>+</th>
<th>(</th>
<th>)</th>
<th>$ (EOF)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>T</strong></td>
<td>↔ S$</td>
<td></td>
<td></td>
<td>↔ S$</td>
<td></td>
</tr>
<tr>
<td><strong>S</strong></td>
<td>↦ E S’</td>
<td>↔ E S’</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>S’</strong></td>
<td></td>
<td>↔ + S</td>
<td>↔ ε</td>
<td>↔ ε</td>
<td></td>
</tr>
<tr>
<td><strong>E</strong></td>
<td>↔ num.</td>
<td>↔ ( 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 \textit{first} in strings that can be derived from $\gamma$
  - Add the production $\rightarrow \gamma$ to the entry $(A, \text{token})$ for each such token.
- If $\gamma$ can derive $\varepsilon$ (the empty string), then we construct the set of all input tokens that may \textit{follow} the nonterminal $A$ in the grammar.
  - Add the production $\rightarrow \gamma$ to the entry $(A, \text{token})$ for each such token.

- Note: if there are two different productions for a given entry, the grammar is not LL(1)
Example

• First(T) = First(S)
• First(S) = First(E)
• First(S’) = { + }
• First(E) = { number, ‘(‘ } 

• Follow(S’) = Follow(S)
• Follow(S) = { $, ‘)’ } ∪ 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>$S$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>E S’</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S’</td>
<td>+ S</td>
<td></td>
<td></td>
<td></td>
<td>ε</td>
</tr>
<tr>
<td>E</td>
<td>num.</td>
<td></td>
<td></td>
<td></td>
<td>( S )</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.
<table>
<thead>
<tr>
<th></th>
<th><strong>number</strong></th>
<th><strong>+</strong></th>
<th></th>
<th></th>
<th><strong>$ (EOF)$</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>T</strong></td>
<td>$\rightarrow S$</td>
<td></td>
<td></td>
<td></td>
<td>$\rightarrow S$</td>
</tr>
<tr>
<td><strong>S</strong></td>
<td>$\rightarrow E S'$</td>
<td></td>
<td></td>
<td></td>
<td>$\rightarrow E S'$</td>
</tr>
<tr>
<td><strong>S'</strong></td>
<td></td>
<td>$\rightarrow + S$</td>
<td></td>
<td>$\rightarrow \epsilon$</td>
<td>$\rightarrow \epsilon$</td>
</tr>
<tr>
<td><strong>E</strong></td>
<td>$\rightarrow \text{num.}$</td>
<td></td>
<td></td>
<td></td>
<td>$\rightarrow ( S )$</td>
</tr>
</tbody>
</table>

Hand-generated LL(1) code for the table above.

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

• Top-down parsing that finds the leftmost derivation.
  Language Grammar \Rightarrow LL(1) grammar \Rightarrow prediction table \Rightarrow 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
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…

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) + 5
(S + (3 + 4)) + 5 | (1 + 2) | + (3 + 4) + 5
(S + (E + 4)) + 5 | (1 + 2 + (3 + 4)) | + 4) + 5
(S + (S + 4)) + 5 | (1 + 2 + (3 + 4)) | + 4) + 5
(S + (S + E)) + 5 | (1 + 2 + (3 + 4)) | + 4) + 5
(S + (S)) + 5 | (1 + 2 + (3 + 4)) | + 4) + 5
(S + E) + 5 | (1 + 2 + (3 + 4)) | + 4) + 5
(S) + 5 | (1 + 2 + (3 + 4)) | + 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 ⟷ 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 \( \text{stack + input} \)

• 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>S</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 ) 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 ) number</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 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)))

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: e.g.

    | Stack   | Input          | Action |
    |---------|----------------|--------|
    | (x, (y, z), w) | shift (        |        |
    | x, (y, z), w  | shift x        |        |

  - **Reduce:** Replace symbols γ at top of stack with nonterminal X such that X ⟷ γ is a production. (pop γ, push X): e.g.

    | Stack   | Input          | Action         |
    |---------|----------------|----------------|
    | (x      | , (y, z), w)   | reduce S ⟷ id  |
    | S       | , (y, z), w    | reduce L ⟷ 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>(x, (y, z), w)</td>
<td>shift (x)</td>
</tr>
<tr>
<td>(x, (y, z), w)</td>
<td>(x, (y, z), w)</td>
<td>shift x</td>
</tr>
<tr>
<td>(S, (y, z), w)</td>
<td>(S, (y, z), w)</td>
<td>reduce S → id</td>
</tr>
<tr>
<td>(L, (y, z), w)</td>
<td>(L, (y, z), w)</td>
<td>reduce L → S</td>
</tr>
<tr>
<td>(L, (z), w)</td>
<td>(L, (z), w)</td>
<td>shift ,</td>
</tr>
<tr>
<td>(L, (L))</td>
<td>(L, (L))</td>
<td>shift (L)</td>
</tr>
<tr>
<td>(L, (L, z), w)</td>
<td>(L, (L, z), w)</td>
<td>reduce S → id</td>
</tr>
<tr>
<td>(L, (S), w)</td>
<td>(L, (S), w)</td>
<td>reduce L → S</td>
</tr>
<tr>
<td>(L, (L))</td>
<td>(L, (L))</td>
<td>shift )</td>
</tr>
<tr>
<td>(L, (L), w)</td>
<td>(L, (L), w)</td>
<td>reduce S → (L)</td>
</tr>
<tr>
<td>(L, S, w)</td>
<td>(L, S, w)</td>
<td>reduce L → L, S</td>
</tr>
<tr>
<td>(L, w)</td>
<td>(L, 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, \( X \rightarrow \varepsilon \) 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
  
  \[
  \begin{align*}
  S & \rightarrow ( L ) \mid id \\
  L & \rightarrow S \mid L, S
  \end{align*}
  \]

- 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 \( \alpha \) 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: $\text{CLOSURE}([S' \rightarrow .S]) = \{S' \rightarrow .S, S \rightarrow .(L), S \rightarrow \text{id}\}$
- 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 $S' \rightarrow .S$

\[
\begin{align*}
S' & \rightarrow S$
S & \rightarrow (L) \mid \text{id}
L & \rightarrow S \mid L, S
\end{align*}
\]
Example: Constructing the DFA

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

• 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)
Example: Constructing the DFA

• Finally, for each new state, we take the closure.
• Note that we have to perform two iterations to compute $CLOSURE(\{S \rightarrow ( . L )\})$
  – 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

1. \( S' \rightarrow .S\) 
2. \( S \rightarrow \text{id}. \)
3. \( S \rightarrow (\text{L}) \)
4. \( S \rightarrow \text{id} \)
5. \( L \rightarrow L, . \)
6. \( S \rightarrow (\text{L}). \)
7. \( L \rightarrow S. \)
8. \( L \rightarrow L, .S \)
9. \( S \rightarrow L, S. \)

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

Done!

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.
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. $1(3(3L_5)_6$
  – On a reduction $X \xrightarrow{\gamma}$, pop stack to reveal the state too:
    e.g. From stack $1(3(3L_5)_6$ 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:
state \ast (\text{terminals} + \text{nonterminals})

- Entries for the “action table” specify two kinds of actions:
  - Shift and goto state $n$
  - Reduce using reduction $X \dashrightarrow \gamma$
    - 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></th>
<th></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>
<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>s2</td>
<td></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></td>
<td>DONE</td>
<td></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>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 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))$</td>
<td>(s3)</td>
</tr>
<tr>
<td>(\varepsilon_1(3)</td>
<td>((y, z), w))$</td>
<td>(s2)</td>
</tr>
<tr>
<td>(\varepsilon_1(3x_2)</td>
<td>(x, (y, z), w))$</td>
<td>(\text{Reduce: } S \rightarrow \text{id})</td>
</tr>
<tr>
<td>(\varepsilon_1(3S)</td>
<td>((y, z), w))$</td>
<td>(g7) (from state 3 follow S)</td>
</tr>
<tr>
<td>(\varepsilon_1(3S_7)</td>
<td>((y, z), w))$</td>
<td>(\text{Reduce: } L \rightarrow S)</td>
</tr>
<tr>
<td>(\varepsilon_1(3L)</td>
<td>((y, z), w))$</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  
  
  shift/reduce  
  
  reduce/reduce  
  
  S \rightarrow ( L ).  
  
  S \rightarrow ( L ).  
  
  L \rightarrow .L, S  
  
  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:

  \[
  \begin{align*}
  \text{left} & : S & \rightarrow & S + E \mid E \\
  & E & \rightarrow & \text{number} \mid ( S )
  \\
  \text{right} & : S & \rightarrow & E + S \mid E \\
  & E & \rightarrow & \text{number} \mid ( S )
  \end{align*}
  \]

- 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:
    \[ A \rightarrow \alpha \beta \text{, } \mathcal{L} \]

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

- Start item: \( S' \mapsto .S$, {}, \)
- Since \( S \) is to the right of a ‘.’, add:
  \( S \mapsto .E + S, \{\$\} \) \hspace{1cm} \text{Note: \{\$\} is FIRST(\$)}
  \( S \mapsto .E, \{\$\} \)
- Need to keep closing, since \( E \) appears to the right of a ‘.’ in ‘.E + S’:
  \( E \mapsto .\text{number}, \{+\} \) \hspace{1cm} \text{Note: + added for reason 1}
  \( E \mapsto .( S ), \{+\} \)
- Because \( E \) also appears to the right of ‘.’ in ‘.E’ we get:
  \( E \mapsto .\text{number}, \{\$\} \) \hspace{1cm} \text{Note: \$ added for reason 2}
  \( E \mapsto .( S ), \{\$\} \)
- All items are distinct, so we’re done
Using the DFA

- 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 ‘.’

S’ ⟷ .S$  {}
S ⟷ .E + S  {$}
S ⟷ .E  {$}
E ⟷ .num  {+}
E ⟷ .( S )  {+}
E ⟷ .num  {$}
E ⟷ .( S )  {$}

E +

S ⟷ E .+ S  {$}
S ⟷ E.  {$}

<table>
<thead>
<tr>
<th></th>
<th>+</th>
<th>$</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>g2</td>
</tr>
<tr>
<td>2</td>
<td>s3</td>
<td>S ⟷ E</td>
<td></td>
</tr>
</tbody>
</table>

Fragment of the Action & Goto tables
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:
  - 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)