Parsers

Agenda

- Terminology
- LL(1) Parsers
- Overview of LR Parsing

Terminology

- Grammar $G = (V_t, V_n, S, P)$
  - $V_t$ is the set of terminals
  - $V_n$ is the set of non-terminals
  - $S$ is the start symbol
  - $P$ is the set of productions
  - Each production takes the form: $V_n \rightarrow \lambda | (V_n | V_t)^+$
  - Grammar is context-free (why?)
- A simple grammar:
  $G = \{(a, b), (S, A, B), (S \rightarrow A B \$, A \rightarrow A a, A \rightarrow a, B \rightarrow B b, B \rightarrow b), S\}$

Generating strings

- Given a start rule, productions tell us how to rewrite a non-terminal into a different set of symbols
- By convention, first production applied has the start symbol on the left, and there is only one such production

To derive the string "a a b b b" we can do the following rewrites:

$$
S \Rightarrow A B \$ \Rightarrow a a B b \$ \Rightarrow a a a B b b b \$
$$

Productions (rewrite rules) tell us how to derive strings in the language

- Apply productions to rewrite strings into other strings
- We will use the standard BNF form

$P = \{
S \rightarrow A B \$
A \rightarrow A a
A \rightarrow a
B \rightarrow B b
B \rightarrow b
\}$

Vocabulary

- $V$ is the vocabulary of a grammar, consisting of terminal ($V_t$) and non-terminal ($V_n$) symbols
- For our sample grammar
  - $V_t = (S, A, B)$
    - Non-terminals are symbols on the LHS of a production
    - Non-terminals are constructs in the language that are recognized during parsing
  - $V_n = (a, b)$
    - Terminals are the tokens recognized by the scanner
    - They correspond to symbols in the text of the program
**Terminology**

- Strings are composed of symbols
  - A a a B b A a is a string
- We will use Greek letters to represent strings composed of both terminals and non-terminals
- \( L(G) \) is the language produced by the grammar \( G \)
  - All strings consisting of only terminals that can be produced by \( G \)
  - In our example, \( L(G) = a+b+\$ \)
  - All regular expressions can be expressed as grammars for context-free languages, but not vice-versa
  - Consider: \( ab\$ \) (what is the grammar for this?)

**Parse trees**

- Tree which shows how a string was produced by a language
  - Interior nodes of tree: non-terminals
  - Children: the terminals and non-terminals generated by applying a production rule
  - Leaf nodes: terminals

**Leftmost derivation**

- Rewriting of a given string starts with the leftmost symbol
- Exercise: do a leftmost derivation of the input program \( F(V + V) \) using the following grammar:

<table>
<thead>
<tr>
<th>Rule</th>
<th>Production</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>Prefix (E)</td>
</tr>
<tr>
<td>E</td>
<td>V Tail</td>
</tr>
<tr>
<td>Prefix</td>
<td>F</td>
</tr>
<tr>
<td>Prefix</td>
<td>( \lambda )</td>
</tr>
<tr>
<td>Tail</td>
<td>+ E</td>
</tr>
<tr>
<td>Tail</td>
<td>( \lambda )</td>
</tr>
</tbody>
</table>

- What does the parse tree look like?

**Rightmost derivation**

- Rewrite using the rightmost non-terminal, instead of the left
- What is the rightmost derivation of this string? \( F(V + V) \)

**Simple conversions**

<table>
<thead>
<tr>
<th>Rule</th>
<th>Production</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>A</td>
<td>C</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rule</th>
<th>Production</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>E (F)</td>
</tr>
<tr>
<td>D</td>
<td>E Ftail</td>
</tr>
<tr>
<td>Ftail</td>
<td>F Ftail</td>
</tr>
<tr>
<td>Ftail</td>
<td>( \lambda )</td>
</tr>
</tbody>
</table>

**Top-down vs. Bottom-up parsers**

- Top-down parsers use left-most derivation
- Bottom-up parsers use right-looking parse
- Notation:
  - LL(1): Leftmost derivation with 1 symbol lookahead
  - LL(k): Leftmost derivation with k symbols lookahead
  - LR(1): Right-looking derivation with 1 symbol lookahead
What is parsing

- Parsing is recognizing members in a language specified/defined/generated by a grammar
- When a construct (corresponding to a production in a grammar) is recognized, a typical parser will take some action
  - In a compiler, this action generates an intermediate representation of the program construct
  - In an interpreter, this action might be to perform the action specified by the construct. Thus, if \(a+b\) is recognized, the value of \(a\) and \(b\) would be added and placed in a temporary variable

Another simple grammar

PROGRAM \(\rightarrow\) begin STMTLIST $
STMTLIST \rightarrow\) STMT ; STMTLIST
STMTLIST \rightarrow end
STMT \rightarrow id
STMT \rightarrow if ( id ) STMTLIST

- A sentence in the grammar:
  begin if (id) id ; end; end; end; $

Another example

S \(\rightarrow\) A B $
A \rightarrow x A$
A \rightarrow y A$
A \rightarrow \lambda$
B \rightarrow b

- Consider S \(\rightarrow\) A B $\rightarrow$ x A $A B \rightarrow x A B $\rightarrow$ x A b $
- When parsing x A b $ we know from the goal production we need to match an A. The next token is x, so we apply A \(\rightarrow\) x A
- The parser matches x, matches s and now needs to parse A again
- How do we know which A to use? We need to use A \(\rightarrow\) \lambda
  - When matching the right hand side of A \(\rightarrow\) \lambda, the next token comes from a non-terminal that follows A (i.e., it must be b)
  - Tokens that can follow A are called the follow set of A

First and follow sets

- First(\(\alpha\)) the set of terminals that begin all strings that can be derived from \(\alpha\)
  - First(A) = \{x, y\}
  - First(xA) = \{x\}
  - First (AB) = \{x, y, b\}
- Follow(A): the set of terminals that can appear immediately after A in some partial derivation
  - Follow(A) = \{b\}

First and follow sets

- First(\(\alpha\)) = \{a \in V_T \mid \alpha \Rightarrow^* aB\} \cup \{\lambda \mid \alpha \Rightarrow^* \lambda\}
- Follow(\(\alpha\)) = \{a \in V_T \mid S \Rightarrow^* \ldots Aa \ldots \} \cup \{$ \mid S \Rightarrow^* \ldots AS \}$
Computing first sets

- **Terminal:** First(a) = {a}
- **Non-terminal:** First(A)
  - Look at all productions for A
    \[ A \rightarrow X_1 X_2 \ldots X_k \]
  - First(A) \supset (First(X_i) - \lambda)
  - If \lambda \in First(X_i), First(A) \supset (First(X_i) - \lambda)
  - If \lambda is in First(X_i) for all i, then \lambda \in First(A)
  - Computing First(a): similar procedure to computing First(A)

### Exercise

- What are the first sets for all the non-terminals in following grammar:
  - \[ S \rightarrow A \ B \ \$ \]
  - \[ A \rightarrow x \ a \ A \]
  - \[ A \rightarrow y \ a \ A \]
  - \[ A \rightarrow \lambda \]
  - \[ B \rightarrow b \]
  - \[ B \rightarrow A \]

Computing follow sets

- **Follow(S) = \{\$\}
- To compute Follow(A):
  - Find productions which have A on rhs. Three rules:
    1. \[ X \rightarrow \alpha \ A \ \beta: \text{Follow(A) } \supset (\text{First(\beta) - \lambda}) \]
    2. \[ X \rightarrow \alpha \ A \ \beta: \text{If } \lambda \in \text{First(\beta)}, \text{Follow(A) } \supset \text{Follow(X)} \]
    3. \[ X \rightarrow \alpha \ A: \text{Follow(A) } \supset \text{Follow(X)} \]
  - Note: Follow(X) never has \lambda in it.

### Exercise

- What are the follow sets for
  - \[ S \rightarrow A \ B \ \$ \]
  - \[ A \rightarrow x \ a \ A \]
  - \[ A \rightarrow y \ a \ A \]
  - \[ A \rightarrow \lambda \]
  - \[ B \rightarrow b \]
  - \[ B \rightarrow A \]

Towards parser generators

- Key problem: as we read the source program, we need to decide what productions to use
  - Step 1: find the tokens that can tell which production P (of the form A \rightarrow X_1 X_2 \ldots X_n) applies
    \[ \text{Predict(P)} = \]
    \[
    \begin{cases}
    (\text{First}(X_1 \ldots X_m) ) & \text{if } \lambda \notin \text{First}(X_1 \ldots X_m) \\
    (\text{First}(X_1 \ldots X_m) - \lambda) \cup \text{Follow(A)} & \text{otherwise}
    \end{cases}
    \]
    - If next token is in Predict(P), then we should choose this production

Parse tables

- Step 2: build a parse table
  - Given some non-terminal V_n (the non-terminal we are currently processing) and a terminal V_t (the lookahead symbol), the parse table tells us which production P to use (or that we have an error)
  - More formally:
    \[ T: V_n \times V_t \rightarrow P \cup \{\text{Error}\} \]
Building the parse table

- Start: \( T[A][t] = //\text{initialize all fields to "error"} \)
  - foreach A:
    - foreach P with A on its lhs:
      - foreach \( t \) in Predict(P):
        - \( T[A][t] = P \)

- Exercise: build parse table for our toy grammar

<table>
<thead>
<tr>
<th>Rule</th>
<th>Parse stack</th>
<th>Remaining input</th>
<th>Parser action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. ( S \rightarrow A \ B \ $ )</td>
<td>( S )</td>
<td>( x \ a \ y \ a \ b \ $ )</td>
<td>predict 1</td>
</tr>
<tr>
<td>2. ( A \rightarrow x \ a \ A )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3. ( A \rightarrow y \ a \ A )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4. ( A \rightarrow \lambda )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5. ( B \rightarrow b )</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Recursive-descent parsers

- Given the parse table, we can create a program which generates recursive descent parsers
- Remember the recursive descent parser we saw for MICRO
- If the choice of production is not unique, the parse table tells us which one to take
- However, there is an easier method!

Stack-based parser for LL(1)

- Given the parse table, a stack-based algorithm is much simpler to generate than a recursive descent parser
- Basic algorithm:
  1. Push the RHS of a production onto the stack
  2. Pop a symbol, if it is a terminal, match it
  3. If it is a non-terminal, take its production according to the parse table and go to 1

- Algorithm on page 121
- Note: always start with start state

<table>
<thead>
<tr>
<th>Parse stack</th>
<th>Remaining input</th>
<th>Parser action</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S )</td>
<td>( x \ a \ y \ a \ b \ $ )</td>
<td>predict 1</td>
</tr>
<tr>
<td>( A \ B \ $ )</td>
<td>( x \ a \ y \ a \ b \ $ )</td>
<td>predict 2</td>
</tr>
</tbody>
</table>

An example

- How would a stack-based parser parse: \( x \ a \ y \ a \ b \)

<table>
<thead>
<tr>
<th>Parse stack</th>
<th>Remaining input</th>
<th>Parser action</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S )</td>
<td>( x \ a \ y \ a \ b \ $ )</td>
<td>predict 1</td>
</tr>
<tr>
<td>( A \ B \ $ )</td>
<td>( x \ a \ y \ a \ b \ $ )</td>
<td>predict 2</td>
</tr>
</tbody>
</table>

An example

- How would a stack-based parser parse: \( x \ a \ y \ a \ b \)

<table>
<thead>
<tr>
<th>Parse stack</th>
<th>Remaining input</th>
<th>Parser action</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S )</td>
<td>( x \ a \ y \ a \ b \ $ )</td>
<td>predict 1</td>
</tr>
<tr>
<td>( A \ B \ $ )</td>
<td>( x \ a \ y \ a \ b \ $ )</td>
<td>predict 2</td>
</tr>
<tr>
<td>( x \ a \ A \ B \ $ )</td>
<td>( x \ a \ y \ a \ b \ $ )</td>
<td>match(a)</td>
</tr>
</tbody>
</table>
An example

How would a stack-based parser parse:

\[ x \ a \ y \ a \ b \]

<table>
<thead>
<tr>
<th>Parse stack</th>
<th>Remaining input</th>
<th>Parser action</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>x y a b $</td>
<td>predict 1</td>
</tr>
<tr>
<td>A B $</td>
<td>x y a b $</td>
<td>predict 2</td>
</tr>
<tr>
<td>x a A B $</td>
<td>x y a b $</td>
<td>match(x)</td>
</tr>
<tr>
<td>A B $</td>
<td>y a b $</td>
<td>predict 3</td>
</tr>
<tr>
<td>y a A B $</td>
<td>y a b $</td>
<td>match(y)</td>
</tr>
<tr>
<td>A B $</td>
<td>b $</td>
<td>predict 4</td>
</tr>
</tbody>
</table>

An example

How would a stack-based parser parse:

\[ x \ a \ y \ a \ b \]

<table>
<thead>
<tr>
<th>Parse stack</th>
<th>Remaining input</th>
<th>Parser action</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>x y a b $</td>
<td>predict 1</td>
</tr>
<tr>
<td>A B $</td>
<td>x y a b $</td>
<td>predict 2</td>
</tr>
<tr>
<td>x a A B $</td>
<td>x y a b $</td>
<td>match(x)</td>
</tr>
<tr>
<td>A B $</td>
<td>y a b $</td>
<td>predict 3</td>
</tr>
<tr>
<td>y a A B $</td>
<td>y a b $</td>
<td>match(y)</td>
</tr>
<tr>
<td>A B $</td>
<td>b $</td>
<td>predict 4</td>
</tr>
<tr>
<td>B $</td>
<td>b $</td>
<td>predict 5</td>
</tr>
</tbody>
</table>
An example

- How would a stack-based parser parse:

  \( x \ y \ a \ y \ a \ b \)

<table>
<thead>
<tr>
<th>Parse stack</th>
<th>Remaining input</th>
<th>Parser action</th>
</tr>
</thead>
<tbody>
<tr>
<td>S \x a y a b $</td>
<td>x a y a b $</td>
<td>predict 1</td>
</tr>
<tr>
<td>A B $</td>
<td>x a y a b $</td>
<td>predict 2</td>
</tr>
<tr>
<td>x a A B $</td>
<td>x a y a b $</td>
<td>match(x)</td>
</tr>
<tr>
<td>a A B $</td>
<td>a y a b $</td>
<td>match(a)</td>
</tr>
<tr>
<td>A B $</td>
<td>y a b $</td>
<td>predict 3</td>
</tr>
<tr>
<td>y a A B $</td>
<td>y a b $</td>
<td>match(y)</td>
</tr>
<tr>
<td>a A B $</td>
<td>a b $</td>
<td>match(a)</td>
</tr>
<tr>
<td>A B $</td>
<td>b $</td>
<td>predict 4</td>
</tr>
<tr>
<td>B $</td>
<td>b $</td>
<td>predict 5</td>
</tr>
<tr>
<td>b $</td>
<td>b $</td>
<td>match(b)</td>
</tr>
</tbody>
</table>

LL(k) parsers

- Can use similar techniques for LL(k) parsers
- Use more than one symbol of look-ahead to distinguish productions
- Why might this be bad?

Dealing with semantic actions

- Recall: we can annotate a grammar with action symbols
- Tell the parser to invoke a semantic action routine
- Can simply push action symbols onto stack as well
- When popped, the semantic action routine is called

Non-LL(1) grammars

- Not all grammars are LL(1)!
- Consider:

  \[
  \text{<stmt>} \rightarrow \text{<expr> then <stmt list> endif} \\
  \text{<stmt>} \rightarrow \text{<expr> then <stmt list> else <stmt list> endif}
  \]
- This is not LL(1) (why?)
- We can turn this in to

  \[
  \text{<stmt>} \rightarrow \text{<expr> then <stmt list> <if suffix> where <if suffix> \rightarrow \text{<if suffix> else <stmt list> endif}}
  \]

Left recursion

- Left recursion is a problem for LL(1) parsers
- LHS is also the first symbol of the RHS
- Consider:

  \[
  E \rightarrow E + T
  \]
- What would happen with the stack-based algorithm?
Removing left recursion

\[ E \rightarrow E + T \\
E \rightarrow T \\
E \rightarrow + T E \]

Etail \rightarrow \lambda

Algorithm on page 125

Are all grammars LL(1)?

- No! Consider the if-then-else problem
- if x then y else z
- Problem: else is optional
- if a then if b then c else d
- Which if does the else belong to?
- This is analogous to a "bracket language": \([ i \] (i \geq j)

\[ S \rightarrow [ S C \\
S \rightarrow \lambda \\
C \rightarrow ] \\
C \rightarrow \lambda \]

([ can be parsed: SS\lambda C or S\lambda C (it's ambiguous!)

Solving the if-then-else problem

- The ambiguity exists at the language level. To fix, we need to define the semantics properly
- "]" matches nearest unmatched [
- This is the rule C uses for if-then-else
- What if we try this?

\[ S \rightarrow [ S \\
S \rightarrow S I \\
S I \rightarrow [ S I ] \\
S I \rightarrow \lambda \]

This grammar is still not LL(1) (or LL(k) for any k!)

Two possible fixes

- If there is an ambiguity, prioritize one production over another
- e.g., if C is on the stack, always match "]" before matching "\]

\[ S \rightarrow [ S C \\
S \rightarrow \lambda \\
C \rightarrow ] \\
C \rightarrow \lambda \]

- Another option: change the language!
- e.g., all if-statements need to be closed with an endif

\[ S \rightarrow \text{if S E} \\
S \rightarrow \text{other} \\
E \rightarrow \text{else S endif} \\
E \rightarrow \text{endif} \]

Parsing if-then-else

- What if we don’t want to change the language?
- C does not require {} to delimit single-statement blocks
- To parse if-then-else, we need to be able to look ahead at the entire rhs of a production before deciding which production to use
- In other words, we need to determine how many ""]" to match before we start matching "["s

- LR parsers can do this!

LR Parsers

- Parser which does a Left-to-right, Right-most derivation
- Rather than parse top-down, like LL parsers do, parse bottom-up, starting from leaves
- Basic idea: put tokens on a stack until an entire production is found
- Issues:
  - Recognizing the endpoint of a production
  - Finding the length of a production (RHS)
  - Finding the corresponding nonterminal (the LHS of the production)
Data structures

- At each state, given the next token,
- A *goto table* defines the successor state
- An *action table* defines whether to
  - *shift* – put the next state and token on the stack
  - *reduce* – an RHS is found; process the production
  - *terminate* – parsing is complete

Example

- Consider the simple grammar:
  
  `<program> → begin <stmts> end $`
  
  `<stmts> → SimpleStmt <stmts>`
  
  `<stmts> → begin <stmts> end <stmts>`
  
  `<stmts> → λ`

  - Shift-reduce driver algorithm on page 142

Action and *goto* tables

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>begin</td>
<td>end</td>
<td>$</td>
<td>SimpleStmt</td>
<td>$&lt;program&gt;</td>
</tr>
<tr>
<td>0</td>
<td>S / 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>S / 4</td>
<td>R4</td>
<td>S / 5</td>
<td>S / 2</td>
</tr>
<tr>
<td>2</td>
<td>S / 3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>S / 4</td>
<td>R4</td>
<td>S / 5</td>
<td>S / 7</td>
</tr>
<tr>
<td>5</td>
<td>S / 6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>S / 4</td>
<td>R4</td>
<td>S / 5</td>
<td>S / 10</td>
</tr>
<tr>
<td>7</td>
<td>S / 8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>S / 9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>S / 4</td>
<td>R4</td>
<td>S / 6</td>
<td>S / 11</td>
</tr>
<tr>
<td>10</td>
<td>R2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>R3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

LR Parsers

- Basic idea:
  - *shift* tokens onto the stack. At any step, keep the set of productions that could generate the read-in tokens
  - *reduce* the RHS of recognized productions to the corresponding non-terminal on the LHS of the production. Replace the RHS tokens on the stack with the LHS non-terminal.

LR(k) parsers

- LR(0) parsers
  - No lookahead
  - Predict which action to take by looking only at the symbols currently on the stack
- LR(k) parsers
  - Can look ahead k symbols
  - Most powerful class of deterministic bottom-up parsers
  - LR(1) and variants are the most common parsers
Terminology for LR parsers

- Configuration: a production augmented with a "•"
  \[ A \rightarrow X_1 \ldots X_i \bullet X_{i+1} \ldots X_j \]
- The "•" marks the point to which the production has been recognized. In this case, we have recognized \[ X_1 \ldots X_i \]
- Configuration set: all the configurations that can apply at a given point during the parse:
  \[ A \rightarrow B \bullet CD \]
  \[ A \rightarrow B \bullet GH \]
  \[ T \rightarrow B \bullet Z \]
- Idea: every configuration in a configuration set is a production that can possibly be matched

Configuration closure set

- Include all the configurations necessary to recognize the next symbol after the "•"
- closure0(configuration_set) defined on page 146
- Example:
  \[ \text{closure0}(\{S \rightarrow \bullet E \}) = \{ S \rightarrow \bullet E \}
     \]

Successor configuration set

- Starting with the initial configuration set
  \[ s_0 = \text{closure0}(\{S \rightarrow \bullet \# \}) \]
  an LR(0) parser will find the successor given the next symbol \( X \)
- \( X \) can be either a terminal (the next token from the scanner) or a non-terminal (the result of applying a reduction)
- Determining the successor \( s' = \text{go_to0}(s, X) \):
  - For each configuration in \( s \) of the form \( A \rightarrow \beta \bullet X \gamma \) add \( A \rightarrow \beta \bullet X \cdot \gamma \) to \( t \)
  - \( s' = \text{closure0}(t) \)

CFSM

- CFSM = Characteristic Finite State Machine
- Nodes are configuration sets (starting from \( s_0 \))
- Arcs are \( \text{go_to} \) relationships

Building the goto table

- We can just read this off from the CFSM

<table>
<thead>
<tr>
<th>Symbol</th>
<th>ID</th>
<th>$</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>State</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Building the action table

- Given the configuration set \( s \):
  - We \textbf{shift} if the next token matches a terminal after the "•" in some configuration
    \[ A \rightarrow \alpha \bullet \beta \in s \text{ and } \alpha \in V_t, \text{ else error} \]
  - We \textbf{reduce} production \( P \) if the "•" is at the end of a production
    \[ B \rightarrow \alpha \bullet \in s \text{ where production } P \text{ is } B \rightarrow \alpha \]
  - Extra actions:
    - \textbf{shift} if goto table transitions between states on a non-terminal
    - \textbf{accept} if we are about to shift $
### Action table

<table>
<thead>
<tr>
<th>State</th>
<th>Symbol</th>
<th>$</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
<td>$</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>R2</td>
<td>R2</td>
<td>R2</td>
</tr>
<tr>
<td>2</td>
<td>A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Conflicts in action table

- For LR(0) grammars, the action table entries are unique: from each state, can only shift or reduce.
- But other grammars may have conflicts:
  - Reduce/reduce conflicts: multiple reductions possible from the given configuration.
  - Shift/reduce conflicts: we can either shift or reduce from the given configuration.

### Shift/reduce example

- Consider the following grammar:
  
  \[
  S \rightarrow A \ y \\
  A \rightarrow \lambda \ | \ x
  \]

- This leads to the following initial configuration set:
  
  \[
  S \rightarrow \ast A \ y \\
  A \rightarrow \ast x \\
  A \rightarrow \lambda \ast
  \]

- Can shift or reduce here.

### Lookahead

- Can resolve reduce/reduce conflicts and shift/reduce conflicts by employing lookahead.
  
  - Looking ahead one (or more) tokens allows us to determine whether to shift or reduce.
  
  - (cf how we resolved ambiguity in LL(1) parsers by looking ahead one token)

### Parsers with lookahead

- Adding lookahead creates an LR(1) parser.
  
  - Built using similar techniques as LR(0) parsers, but uses lookahead to distinguish states.
  
  - LR(1) machines can be much larger than LR(0) machines, but resolve many shift/reduce and reduce/reduce conflicts.
  
  - Other types of LR parsers are SLR(1) and LALR(1).
  
  - Differ in how they resolve ambiguities.
  
  - yacc and bison produce LALR(1) parsers.

### Semantic actions

- Recall: in LL parsers, we could integrate the semantic actions with the parser.
  
  - Why? Because the parser was **predictive**.
  
  - Why doesn’t that work for LR parsers?
    
    - Don’t know which production is matched until parser reduces.
    
    - For LR parsers, we put semantic actions at the end of productions.
    
    - May have to rewrite grammar to support all necessary semantic actions.
Backup slides on LR(1)

Parsing

- Configurations in LR(1) look similar to LR(0), but they are extended to include a lookahead symbol
  \[ A \rightarrow X_1 \ldots X_i \cdot X_{i+1} \ldots X_j \cdot l \] (where \( l \in V_t \cup \lambda \))

- If two configurations differ only in their lookahead component, we combine them
  \[ A \rightarrow X_1 \ldots X_i \cdot X_{i+1} \ldots X_j \cdot (l_1 \ldots l_m) \]

Building configuration sets

- To close a configuration
  \[ B \rightarrow \alpha \cdot A \beta \cdot l \]

- Add all configurations of the form \( A \rightarrow \cdot \gamma \cdot u \) where \( u \in \text{First}(\beta) \)

- Intuition: the parse could apply the production for \( A \), and the lookahead after we apply the production should match the next token that would be produced by \( B \)

Example

\[
\text{closure}_1(S \rightarrow E \cdot S \cdot (\lambda)) = \\
S \rightarrow E \cdot S
\]

Example

\[
\text{closure}_1((S \rightarrow E \cdot S \cdot (\lambda)) = \\
S \rightarrow E \cdot S
\]

Example

\[
\text{closure}_1((S \rightarrow E \cdot S \cdot (\lambda)) = \\
S \rightarrow E \cdot S
\]

Example

\[
\text{closure}_1((S \rightarrow E \cdot S \cdot (\lambda)) = \\
S \rightarrow E \cdot S
\]
Example

Building goto and action tables

- The function $\text{goto}_1$ (configuration-set, symbol) is analogous to $\text{goto}_0$ (configuration-set, symbol) for LR(0).
- Build goto table in the same way as for LR(0).
- Key difference: the action table.
  
  \[ \text{action}[s][x] = \]
  
  - $\text{reduce}$ when • is at end of configuration and $x \in \text{lookahead set of configuration}$
    
    \[ A \to \alpha \cdot (\ldots x \ldots) \in s \]
  
  - $\text{shift}$ when • is before $x$
    
    \[ A \to \beta \cdot x \psi \in s \]

Problems with LR(1) parsers

- LR(1) parsers are very powerful ...
  
  - But the table size is much larger than LR(0) — as much as a factor of $|V| \cdot (why?)$
  
  - Example: Algol 60 (a simple language) includes several thousand states!
  
  - Storage efficient representations of tables are an important issue

Solutions to the size problem

- Different parser schemes
  
  - SLR (simple LR): build an CFSM for a language, then add lookahead wherever necessary (i.e., add lookahead to resolve shift/reduce conflicts)
  
  - What should the lookahead symbol be?
  
  - To decide whether to reduce using production $A \to \alpha$, use Follow($A$)
  
  - LALR: merge LR states in certain cases (we won’t discuss this)