Parsers

Agenda

- How to define a language
- Context-free grammars
- How to recognize a language
- LL(1) Parsers
- 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 $V_n \rightarrow \lambda$ | $(V_n \mid V_t)^+$
    - Grammar is context-free (why?)
  - A simple grammar:
    $G = (\{a, b\}, \{S, A, B\}, \{S \rightarrow A B, A \rightarrow A a, A \rightarrow a, B \rightarrow B b, B \rightarrow b\}, S)$

Terminology

- $V$ is the vocabulary of a grammar, consisting of terminal ($V_t$) and non-terminal ($V_n$) symbols
- For our sample grammar
  - $V_t = \{A, B\}$
    - 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

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:

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

- Strings are composed of symbols
- \( A a B b A a \) 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: \( a b^i \) (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:

  \[
  \begin{align*}
  E & \rightarrow \text{Prefix} (E) \\
  E & \rightarrow V \text{Tail} \\
  \text{Prefix} & \rightarrow F \\
  \text{Prefix} & \rightarrow \lambda \\
  \text{Tail} & \rightarrow + E \\
  \text{Tail} & \rightarrow \lambda
  \end{align*}
  \]

- 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? \( F(V + V) \)

  Using the following grammar:

  \[
  \begin{align*}
  E & \rightarrow \text{Prefix} (E) \\
  E & \rightarrow V \text{Tail} \\
  \text{Prefix} & \rightarrow F \\
  \text{Prefix} & \rightarrow \lambda \\
  \text{Tail} & \rightarrow + E \\
  \text{Tail} & \rightarrow \lambda
  \end{align*}
  \]

Simple conversions

- \( A \rightarrow B | C \) \( \rightarrow \) \( A \rightarrow B \)
  \( A \rightarrow C \)

- \( D \rightarrow E \{F\} \) \( \rightarrow \) \( D \rightarrow E \text{ Ftail} \)
  \( \text{Ftail} \rightarrow F \text{ Ftail} \)
  \( \text{Ftail} \rightarrow \lambda \)

Top-down vs. Bottom-up parsers

- Top-down parsers expand the parse tree in pre-order
  - Identify parent nodes before the children
- Bottom-up parsers expand the parse tree in post-order
  - Identify children before the parents
- Notation:
  - \( LL(1) \): Top-down derivation with 1 symbol lookahead
  - \( LL(k) \): Top-down derivation with \( k \) symbols lookahead
  - \( LR(1) \): Bottom-up 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.

Top-down parsing

• Idea: we know sentence has to start with initial symbol
• Build up partial derivations by predicting what rules are used to expand non-terminals
• Often called predictive parsers
• If partial derivation has terminal characters, match them from the input stream

A simple example

\[
\begin{align*}
S & \rightarrow A \ B \ c \ \$ \\
A & \rightarrow x \ a \ A \\
A & \rightarrow y \ a \ A \\
A & \rightarrow c \\
B & \rightarrow b \\
B & \rightarrow \lambda
\end{align*}
\]

• A sentence in the grammar:
\(x \ a \ c \ c \ \$\)

Current derivation: \(S\)
A simple example

\[ \begin{align*}
S & \rightarrow A B c \\
A & \rightarrow x a A \\
A & \rightarrow y a A \\
A & \rightarrow c \\
B & \rightarrow b \\
B & \rightarrow \lambda
\end{align*} \]

Current derivation: \( A B c \)$

**Predict rule**

A sentence in the grammar:

\[ \begin{align*}
x & \rightarrow a c c \\
x & \rightarrow a c c \\
x & \rightarrow a c c \\
x & \rightarrow a c c \\
x & \rightarrow a c c
\end{align*} \]

Wednesday, September 4, 13
A simple example

\[ S \rightarrow A \ B \ c \ \$ \]
\[ A \rightarrow x \ a \ A \]
\[ A \rightarrow y \ a \ A \]
\[ A \rightarrow c \]
\[ B \rightarrow b \]
\[ B \rightarrow \lambda \]

Choose based on follow set

A sentence in the grammar:
\[ x \ a \ c \ c \ \$ \]

Current derivation:
\[ x \ a \ c \ \lambda \ c \ \$ \]

Predict rule based on next token

A sentence in the grammar:
\[ x \ a \ c \ c \ \$ \]

Current derivation:
\[ x \ a \ c \ c \ \$ \]

Match token

First and follow sets

- First(\(\alpha\)): the set of terminals (and/or \(\lambda\)) that begin all strings that can be derived from \(\alpha\)
- First(A) = \{x, y, \lambda\}
- First(xA) = \{x\}
- First(AB) = \{x, y, b\}
- Follow(A): the set of terminals (and/or $, but no \(\lambda\)s) that can appear immediately after A in some partial derivation
- Follow(A) = \{b\}

First and follow sets

- Terminal: First(a) = \{a\}
- Non-terminal: First(A)
- Look at all productions for A
  \[ A \rightarrow X_1 X_2 ... X_n \]
  \[ \text{First}(A) \supset (\text{First}(X_1) - \lambda) \]
- If \(\lambda \in \text{First}(X_i), \text{First}(A) \supset (\text{First}(X_i) - \lambda)\)
- If \(\lambda\) is in First(\(X_i\)) for all i, then \(\lambda \in \text{First}(A)\)

Computing first sets

- Computing First(\(\alpha\)): 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 \(\text{Follow}(A)\):
  
  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 \(A \rightarrow X_1 X_2 \ldots X_m\)) 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 \(V_i\) (the lookahead symbol), the parse table tells us which production \(P\) to use (or that we have an error).

  • More formally:

    \[T: V_n \times V_i \rightarrow P \cup \{\text{Error}\} \]

Building the parse table

• Start: \(T[A][c] = \{\text{initialize all fields to “error”}\} \)

  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.

  1. \(S \rightarrow A \ B \ \$\)
  2. \(A \rightarrow x \ a \ A\)
  3. \(A \rightarrow y \ a \ A\)
  4. \(A \rightarrow \lambda\)
  5. \(B \rightarrow \ b\)
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

Note: always start with start state

Stack-based parser for LL(1)

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>
<tr>
<td>$a A B$</td>
<td>$x y a b$</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>$x a y a b$</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>$x A B$</td>
<td>$x a y a b$</td>
<td>match(a)</td>
</tr>
<tr>
<td>$a B$</td>
<td>$y a b$</td>
<td>predict 3</td>
</tr>
</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>$x A B$</td>
<td>$x a y a b$</td>
<td>match(a)</td>
</tr>
<tr>
<td>$A B$</td>
<td>$y a b$</td>
<td>predict 3</td>
</tr>
</tbody>
</table>
An example

- How would a stack-based parser parse:
  
  \texttt{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>$x A B$</td>
<td>$x$ $a$ $y$ $a$ $b$ $$$</td>
<td>match(x)</td>
</tr>
<tr>
<td>$a A B$</td>
<td>$y$ $a$ $b$ $$$</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>$A B$</td>
<td>$a$ $b$ $$$</td>
<td>match(a)</td>
</tr>
<tr>
<td>$B$</td>
<td>$b$ $$$</td>
<td>predict 4</td>
</tr>
<tr>
<td>$b$</td>
<td>$b$ $$$</td>
<td>match(b)</td>
</tr>
</tbody>
</table>

1. $S \rightarrow A B$ $\\$
2. $A \rightarrow x a A$
3. $A \rightarrow y a A$
4. $A \rightarrow \lambda$
5. $B \rightarrow b$

An example

- How would a stack-based parser parse:
  
  \texttt{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>$x A B$</td>
<td>$x$ $a$ $y$ $a$ $b$ $$$</td>
<td>match(x)</td>
</tr>
<tr>
<td>$a A B$</td>
<td>$a$ $b$ $$$</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>$A B$</td>
<td>$a$ $b$ $$$</td>
<td>match(a)</td>
</tr>
<tr>
<td>$B$</td>
<td>$b$ $$$</td>
<td>predict 4</td>
</tr>
<tr>
<td>$b$</td>
<td>$b$ $$$</td>
<td>match(b)</td>
</tr>
</tbody>
</table>

1. $S \rightarrow A B$ $\\$
2. $A \rightarrow x a A$
3. $A \rightarrow y a A$
4. $A \rightarrow \lambda$
5. $B \rightarrow b$
Dealing with semantic actions

- When a construct (corresponding to a production in a grammar) is recognized, a typical parser will invoke a semantic 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.

Non-LL(1) grammars

- Not all grammars are LL(1)!
- Consider:
  
  \[
  \text{<stmt>} \rightarrow \text{if <expr> then <stmt list> endif} \\
  \text{<stmt>} \rightarrow \text{if <expr> then <stmt list> else <stmt list> endif}
  \]
- This is not LL(1) (why?)
- We can turn this into:
  
  \[
  \text{<stmt>} \rightarrow \text{if <expr> then <stmt list> <if suffix>} \\
  \text{<if suffix>} \rightarrow \text{endif} \\
  \text{<if suffix>} \rightarrow \text{else <stmt list> endif}
  \]

Removing left recursion

\[
\begin{align*}
E &\rightarrow E + T \\
E &\rightarrow T \\
E &\rightarrow E1 Etail \\
E1 &\rightarrow T \\
Etail &\rightarrow + T Etail \\
Etail &\rightarrow \lambda
\end{align*}
\]

LL(k) parsers

- Can look ahead more than one symbol at a time.
- \( k \)-symbol lookahead requires extending first and follow sets.
- 2-symbol lookahead can distinguish between more rules:
  \[
  A \rightarrow ax | ay
  \]
- More lookahead leads to more powerful parsers.
- What are the downsides?
Are all grammars LL(k)?

• No! Consider the following grammar:

\[
\begin{align*}
S & \rightarrow E \\
E & \rightarrow (E + E) \\
E & \rightarrow (E - E) \\
E & \rightarrow x
\end{align*}
\]

• When parsing E, how do we know whether to use rule 2 or 3?
  • Potentially unbounded number of characters before the distinguishing `+` or `–` is found
  • No amount of lookahead will help!

In real languages?

• Consider the if-then-else problem

```plaintext
if x then y else z
```

• Problem: else is optional

```plaintext
if a then if b then c else d
```

• Which if does the else belong to?

• This is analogous to a “bracket language”:

```
[ i ] j [ i ≥ j ]
```

```
S → [ S C \\
S → λ \\
C → ] \\
C → λ
```

This is 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 → [ S \\
S → S ] \\
S ] → [ S ] \\
S ] → λ
```

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 “λ”

```
S → [ S C \\
S → λ \\
C → ] \\
C → λ
```

• Another option: change the language!

  • e.g., all if-statements need to be closed with an endif

```
S → if S E \\
S → other \\
E → else S endif \\
E → 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)
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.

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

Simple example

1. P → S
2. S → x ; S
3. S → e

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>Shift</td>
</tr>
<tr>
<td>;</td>
<td>Shift</td>
</tr>
<tr>
<td>e</td>
<td>Shift</td>
</tr>
<tr>
<td>P</td>
<td>Reduce 3</td>
</tr>
<tr>
<td>S</td>
<td>Reduce 2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>State</th>
<th>Symbol</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Parsing using an LR(0) parser

- Basic idea: parser keeps track, simultaneously, of all possible productions that could be matched given what it's seen so far. When it sees a full production, match it.
- Maintain a **parse stack** that tells you what state you're in
  - Start in state 0
- In each state, look up in action table whether to:
  - **shift**: consume a token off the input; look for next state in goto table; push next state onto stack
  - **reduce**: match a production; pop off as many symbols from state stack as seen in production; look up where to go according to non-terminal we just matched; push next state onto stack
  - **accept**: terminate parse

Example

- Parse “x : x ; e”

<table>
<thead>
<tr>
<th>Step</th>
<th>Parse Stack</th>
<th>Remaining Input</th>
<th>Parser Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>x ; x ; e</td>
<td>Shift 1</td>
</tr>
<tr>
<td>2</td>
<td>0 i</td>
<td>: x ; e</td>
<td>Shift 2</td>
</tr>
<tr>
<td>3</td>
<td>0 1 2</td>
<td>x ; e</td>
<td>Shift 1</td>
</tr>
<tr>
<td>4</td>
<td>0 1 2 1</td>
<td>: e</td>
<td>Shift 2</td>
</tr>
<tr>
<td>5</td>
<td>0 1 2 1 2</td>
<td>e</td>
<td>Shift 3</td>
</tr>
<tr>
<td>6</td>
<td>0 1 2 1 2 3</td>
<td></td>
<td>Reduce 3 (goto 4)</td>
</tr>
<tr>
<td>7</td>
<td>0 1 2 1 2 4</td>
<td></td>
<td>Reduce 2 (goto 4)</td>
</tr>
<tr>
<td>8</td>
<td>0 1 2 4</td>
<td></td>
<td>Reduce 2 (goto 5)</td>
</tr>
<tr>
<td>9</td>
<td>0 5</td>
<td></td>
<td>Accept</td>
</tr>
</tbody>
</table>

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 “•”
  \[
  A \rightarrow X_1 \ldots X_i \cdot X_{i+1} \ldots X_j
  \]
- 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:
  \[
  A \rightarrow B \cdot CD \\
  A \rightarrow B \cdot GH \\
  T \rightarrow B \cdot Z
  \]
- Idea: every configuration in a configuration set is a production that we could be in the process of matching.

Configuration closure set

- Include all the configurations necessary to recognize the next symbol after the “•”
- For each configuration in set:
  - If next symbol is terminal, no new configuration added
  - If next symbol is non-terminal \(X\), for each production of the form \(X \rightarrow \alpha\), add configuration \(X \rightarrow \cdot \alpha\)

\[
\text{closure}_0(\{S \rightarrow \cdot E \}\}) = \\
\{ \\
S \rightarrow \cdot E \\
E \rightarrow \cdot E + T | T \\
T \rightarrow \cdot ID | \cdot (E) \\
\}
\]

Successor configuration set

- Starting with the initial configuration set
  \[
  s_0 = \text{closure}_0(\{S \rightarrow \cdot \alpha \})
  \]
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 \(s' = \text{go}_\text{to}_0(s, X)\):
  - For each configuration in \(s\) of the form \(A \rightarrow \beta \cdot X \gamma\) add \(A \rightarrow \beta X \cdot \gamma\) to \(t\)
  - \(s' = \text{closure}_0(t)\)

CFSM

- CFSM = Characteristic Finite State Machine
- Nodes are configuration sets (starting from \(s_0\))
- Arrows are \(\text{go}_\text{to}\) relationships

Building the goto table

- We can just read this off from the CFSM

| State | Symbol | \(\cdot\) | \$ | \(|\) |
|-------|--------|--------|-----|-----|
| 0     | 1      | 2      |     |     |
| 1     |        |        |     |     |
| 2     | 3      |        |     |     |
| 3     |        |        |     |     |

Building the action table

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

<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Shift</td>
</tr>
<tr>
<td>1</td>
<td>Reduce 2</td>
</tr>
<tr>
<td>2</td>
<td>Shift</td>
</tr>
<tr>
<td>3</td>
<td>Accept</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 conflict

- Consider the following grammar:
  
  \[
  S \rightarrow A \ y \\
  A \rightarrow x \mid xx
  \]

  This leads to the following configuration set (after shifting one “x”):
  
  \[
  A \rightarrow x \ x \\
  A \rightarrow x
  \]

  Can shift or reduce here

### Shift/reduce example (2)

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

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

  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)*

### 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
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

LR(1) parsing

• Configurations in LR(1) look similar to LR(0), but they are extended to include a lookahead symbol
  A → X₁ ... Xᵢ • Xᵢ₊₁ ... Xⱼ . l (where l ∈ Vᵣ ∪ λ)
• If two configurations differ only in their lookahead component, we combine them
  A → X₁ ... Xᵢ • Xᵢ₊₁ ... Xⱼ . {l₁ ... lₘ}

Building configuration sets

• To close a configuration
  B → α • A β /
• Add all configurations of the form A → • γ; u where u ∈ First(β)
• Intuition: the lookahead symbol for any configuration is the terminal we expect to see after the configuration has been matched
• 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

\[
\text{closure}_1(\{S \to \cdot E \$. (\lambda)\}) = \]

\[
\begin{align*}
S & \to E $. \\
E & \to E + T | T \\
T & \to ID | ($)
\end{align*}
\]

Example

\[
\text{closure}_1(\{S \to \cdot E \$. (\lambda)\}) = \]

\[
\begin{align*}
S & \to E $. \\
E & \to E + T | T \\
T & \to ID | ($)
\end{align*}
\]
Example

closure1({S → • E $, {λ}}) =

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>S → E $</td>
<td></td>
</tr>
<tr>
<td>E → • E + T, ($)</td>
<td></td>
</tr>
<tr>
<td>T → • ID, ($)</td>
<td></td>
</tr>
</tbody>
</table>

Example

closure1({S → • E $, {λ}}) =

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>S → E $</td>
<td></td>
</tr>
<tr>
<td>E → • E + T, ($)</td>
<td></td>
</tr>
<tr>
<td>T → • ID, ($)</td>
<td></td>
</tr>
</tbody>
</table>

Wednesday, September 4, 13
Example

### Building goto and action tables

- The function $\text{goto1}(\text{configuration-set, symbol})$ is analogous to $\text{goto0}(\text{configuration-set, symbol})$ for LR(0).
- Build goto table in the same way as for LR(0).
- Key difference: the action table.

$$\text{action}[s][x] =$$

- **reduce** when $x$ is at end of configuration and $x \in$ lookahead set of configuration
  $$A \rightarrow \alpha \cdot \{x\} \in s$$
- **shift** when $x$ is before $x$
  $$A \rightarrow \beta \cdot x \gamma \in s$$

---

### Example

- Consider the simple grammar:
  $<\text{program}> \rightarrow \text{begin} <\text{stmts}> \text{end} \ $  
  $<\text{stmts}> \rightarrow \text{SimpleStmt} ; <\text{stmts}>$  
  $<\text{stmts}> \rightarrow \text{begin} <\text{stmts}> \text{end} ; <\text{stmts}>$  
  $<\text{stmts}> \rightarrow \lambda$

---

### Action and goto tables

<table>
<thead>
<tr>
<th>Step</th>
<th>Parse Stack</th>
<th>Remaining Input</th>
<th>Parser Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>S ; S ; end S</td>
<td>Shift 1</td>
</tr>
<tr>
<td>1</td>
<td>0 1</td>
<td>S ; S ; end S</td>
<td>Shift 1</td>
</tr>
<tr>
<td>2</td>
<td>0 1 S</td>
<td>S ; S ; end S</td>
<td>Shift 5</td>
</tr>
<tr>
<td>3</td>
<td>0 1 5</td>
<td>; S ; end S</td>
<td>Shift 6</td>
</tr>
<tr>
<td>4</td>
<td>0 1 5 6</td>
<td>S ; end S</td>
<td>Shift 5</td>
</tr>
<tr>
<td>5</td>
<td>0 1 5 6 5</td>
<td>; S ; end S</td>
<td>Shift 6</td>
</tr>
<tr>
<td>6</td>
<td>0 1 5 6 5 6</td>
<td>end S</td>
<td>Reduce 4 (goto 10)</td>
</tr>
<tr>
<td>7</td>
<td>0 1 5 6 5 6 10</td>
<td>end S</td>
<td>Reduce 2 (goto 10)</td>
</tr>
<tr>
<td>8</td>
<td>0 1 5 6 10</td>
<td>end S</td>
<td>Reduce 2 (goto 2)</td>
</tr>
<tr>
<td>9</td>
<td>0 1 2</td>
<td>end S</td>
<td>Shift 3</td>
</tr>
<tr>
<td>10</td>
<td>0 1 2 3</td>
<td>$\ $</td>
<td>Accept</td>
</tr>
</tbody>
</table>

---

### 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 $|V_t|$ (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 $A \rightarrow \alpha$, use $\text{Follow}(A)$
  - LALR: merge LR states when they only differ by lookahead symbols