Parsers
How do we combine tokens?

• Combine tokens ("words" in a language) to form programs ("sentences" in a language)

• Not all combinations of tokens are correct programs (not all sentences are grammatically correct)

• How do we define this?
Producing sentences

- Here are some possible rules for simplified English:
  - All sentences have a noun phrase, then a verb, then a noun phrase (a subject, a verb, an object)
  - Noun phrases are an article (“a” or “the”), an adjective (“black” or “big”) and a noun (“cat” or “dog”)
  - Verbs can be “eats” or “scratches”
- Sentences we can create:
  - “a black cat bites the big dog.” “the big dog eats the black cat.”
- Sentences we can’t:
  - “cat scratches black dog.” “dog the cat bites black.”
More formally

\[
\begin{align*}
\text{S[entence]} & \rightarrow \text{PV P} \\
\text{[noun ]P[hrase]} & \rightarrow \text{R A N} \\
\text{[a]R[ticle]} & \rightarrow \text{a | the} \\
\text{A[jective]} & \rightarrow \text{big | black} \\
\text{N[oun]} & \rightarrow \text{cat | dog} \\
\text{V[erb]} & \rightarrow \text{bites | scratches}
\end{align*}
\]
Generating strings

- Productions tell us how to rewrite a non-terminal into a different set of symbols
- Can rewrite non-terminals until we generate the string we want
- A parser’s job: do this in reverse!
- Figure out how a string was produced

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

\[ S \Rightarrow A \ B \ \$ \Rightarrow A \ a \ B \ \$ \Rightarrow a \ a \ B \ \$ \Rightarrow a \ a \ B \ b \ \$ \Rightarrow a \ a \ B \ b \ b \ \$ \Rightarrow a \ a \ B \ b \ b \ b \ \$ \]
Generalize

- 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:
  $$G = (\{a, b\}, \{S, A, B\}, \{S \rightarrow A \ B \$, \ A \rightarrow A \ a, \ A \rightarrow a, \ B \rightarrow B \ b, \ B \rightarrow b\}, \ S)$$
Terminology

• V is the *vocabulary* of a grammar, consisting of terminal ($V_t$) and non-terminal ($V_n$) symbols

• For our sample grammar
  • $V_n = \{S, A, B\}$
    • Non-terminals are symbols on the LHS of a production
    • Non-terminals are constructs in the language that are recognized during parsing
  • $V_t = \{a, b\}$
    • Terminals are the tokens recognized by the scanner
    • They correspond to symbols in the text of the program
Terminology

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

<table>
<thead>
<tr>
<th>Rule</th>
<th>Production</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>( \text{Prefix } (E) )</td>
</tr>
<tr>
<td>E</td>
<td>( V \text{ 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?

\[ F(V + V) \]

<table>
<thead>
<tr>
<th>E</th>
<th>→</th>
<th>Prefix (E)</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>→</td>
<td>V Tail</td>
</tr>
<tr>
<td>Prefix</td>
<td>→</td>
<td>F</td>
</tr>
<tr>
<td>Prefix</td>
<td>→</td>
<td>λ</td>
</tr>
<tr>
<td>Tail</td>
<td>→</td>
<td>+ E</td>
</tr>
<tr>
<td>Tail</td>
<td>→</td>
<td>λ</td>
</tr>
</tbody>
</table>
Simple conversions

A → B | C

D → E {F}

A → B
A → C

D → E F\text{tail}
F\text{tail} → F F\text{tail}
F\text{tail} → \lambda
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
Another simple grammar

PROGRAM → begin STMTLIST $
STMTLIST → STMT ; STMTLIST
STMTLIST → end
STMT → id
STMT → if ( id ) STMTLIST

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

• What are the terminals and non-terminals of this grammar?
Parsing this grammar

PROGRAM → begin STMTLIST $
STMTLIST → STMT ; STMTLIST
STMTLIST → end
STMT → id
STMT → if ( id ) STMTLIST

• Note
  • To parse STMT in STMTLIST → STMT; STMTLIST, it is necessary to choose between either STMT → id or STMT → if ...
  • Choose the production to parse by finding out if next token is if or id
    • i.e., which production the next input token matches
  • This is the first set of the production
Another example

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

- 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 \( A \rightarrow x \ a \ A \)

- The parser matches \( x \), matches \( a \) 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(\(\alpha\)) = \(\{a \in V_t | \alpha \Rightarrow^* a\beta\} \cup \{\lambda | \text{if } \alpha \Rightarrow^* \lambda\}\)

- Follow(A) = \(\{a \in V_t | S \Rightarrow^+ \ldots Aa \ldots\} \cup \{\$ | \text{if } S \Rightarrow^+ \ldots A \$\}\)

**S:** start symbol

**a:** a terminal symbol

**A:** a non-terminal symbol

**\(\alpha, \beta:** a string composed of terminals and non-terminals (typically, \(\alpha\) is the RHS of a production)

\(\Rightarrow:\) derived in 1 step

\(\Rightarrow^*:\) derived in 0 or more steps

\(\Rightarrow^+:\) derived in 1 or more steps
First and follow sets

- First(α): the set of terminals that begin all strings that can be derived from α
  - 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\}
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) $\supseteq$ (First(X_1) - $\lambda$)
  - If $\lambda \in$ First(X_1), First(A) $\supseteq$ (First(X_2) - $\lambda$)
  - If $\lambda$ is in First(X_i) for all i, then $\lambda \in$ First(A)
- **Computing First(\alpha)**: similar procedure to computing First(A)
Exercise

- What are the first sets for all the non-terminals in following grammar:

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

- Follow(S) = {}
- To compute Follow(A):
  - Find productions which have A on rhs. Three rules:
    1. $X \rightarrow \alpha A \beta$: Follow(A) $\supseteq$ (First(\beta) - \lambda)
    2. $X \rightarrow \alpha A \beta$: If $\lambda \in$ First(\beta), Follow(A) $\supseteq$ Follow(X)
    3. $X \rightarrow \alpha A$: Follow(A) $\supseteq$ 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 $A \rightarrow X_1X_2 \ldots X_m$) applies

\[
\text{Predict}(P) =
\begin{cases}
\text{First}(!_{1 \ldots m}) & \text{if } \lambda \not\in \text{First}(!_{1 \ldots m}) \\
(\text{First}(!_{1 \ldots m}) - \lambda) \cup \text{Follow}() & \text{otherwise}
\end{cases}
\]

• If next token is in \text{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: \( T[A][t] = \) "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. \( B \rightarrow b \)
Stack-based parser for LL(1)

- Given the parse table, we can use a simple algorithm to parse programs
- 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
An example

- How would a stack-based parser parse:

\[ \text{parse stack} \quad \text{remaining input} \quad \text{parser action} \]

<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(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 A B $</td>
<td>a b $</td>
<td>predict 4</td>
</tr>
<tr>
<td>A B $</td>
<td>b $</td>
<td>predict 5</td>
</tr>
<tr>
<td>B $</td>
<td>b $</td>
<td>predict 5</td>
</tr>
<tr>
<td>b $</td>
<td>b $</td>
<td>match(b)</td>
</tr>
<tr>
<td>$</td>
<td>$</td>
<td>Done!</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        

Wednesday, August 31, 2011
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

• 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
Dealing with semantic actions

• 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
  
• Routine manipulates *semantic records* on a stack

• Can generate new records (e.g., to store variable info)

• Can generate code using existing records

• Example: semantic actions for \( x = a + 3 \)

\[
\text{statement ::= ID \#id = expr \#assign} \\
\text{expr ::= term + term \#addop} \\
\text{term ::= ID \#id | LITERAL \#num}
\]
Non-LL(1) grammars

• Not all grammars are LL(1)!
• Consider

\[
\begin{align*}
<\text{stmt}> & \rightarrow \text{if } <\text{expr}> \text{ then } <\text{stmt list}> \text{ endif} \\
<\text{stmt}> & \rightarrow \text{if } <\text{expr}> \text{ then } <\text{stmt list}> \text{ else } <\text{stmt list}> \text{ endif}
\end{align*}
\]

• This is not LL(1) (why?)
• We can turn this in to

\[
\begin{align*}
<\text{stmt}> & \rightarrow \text{if } <\text{expr}> \text{ then } <\text{stmt list}> <\text{if suffix}> \\
<\text{if suffix}> & \rightarrow \text{endif} \\
<\text{if suffix}> & \rightarrow \text{else } <\text{stmt list}> \text{ endif}
\end{align*}
\]
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

\[
\begin{align*}
E & \rightarrow E + T \\
E & \rightarrow T \\
E & \rightarrow EI \ E\text{tail} \\
EI & \rightarrow T \\
E\text{tail} & \rightarrow + T \ E\text{tail} \\
E\text{tail} & \rightarrow \lambda
\end{align*}
\]
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
• 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”: \([i \ ji \ (i \geq j)\)

\[
\begin{align*}
S & \rightarrow [S C \\
S & \rightarrow \lambda \\
C & \rightarrow ] \\
C & \rightarrow \lambda
\end{align*}
\]

\([ ]\) can be parsed: SS\(\lambda\)C or SSS\(\lambda\)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?

\[
\begin{align*}
S & \rightarrow [ S \\
S & \rightarrow S1 \\
S1 & \rightarrow [ S1 ] \\
S1 & \rightarrow \lambda
\end{align*}
\]

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”

  \[
  \begin{align*}
  S & \rightarrow [ S C \\
  S & \rightarrow \lambda \\
  C & \rightarrow ] \\
  C & \rightarrow \lambda
  \end{align*}
  \]

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

  \[
  \begin{align*}
  S & \rightarrow \text{if } S E \\
  S & \rightarrow \text{other} \\
  E & \rightarrow \text{else } S \text{ endif} \\
  E & \rightarrow \text{endif}
  \end{align*}
  \]
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. $P \rightarrow S$
2. $S \rightarrow x ; S$
3. $S \rightarrow e$

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

Wednesday, August 31, 2011
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 A, for each production of the form $X \rightarrow \alpha$, add configuration $X \rightarrow \cdot \alpha$

\[
\text{closure}_0(\{S \rightarrow \cdot \ E \}) = \{
  S \rightarrow \cdot \ E \\
  E \rightarrow \cdot \ E + T \\
  E \rightarrow \cdot T \\
  T \rightarrow \cdot \ ID \\
  T \rightarrow \cdot \ (E)
\}
\]
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 \( s' = \text{go}_0(s, X) \):
  • 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 s0)
- Arcs are go_to relationships

\[
S' \rightarrow S \$ \\
S \rightarrow ID
\]

\[
S' \rightarrow S \$ \\
S \rightarrow ID
\]
Building the goto table

- We can just read this off from the CFSM

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

<table>
<thead>
<tr>
<th>State</th>
<th>ID</th>
<th>$</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td></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 **shift** if the next token matches a terminal after the • in some configuration

  $$A \rightarrow \alpha \cdot a \beta \in s \text{ and } a \in V_t, \text{ else error}$$

  • We **reduce** production $P$ if the • is at the end of a production

  $$B \rightarrow \alpha \cdot \in s \text{ where production } P \text{ is } B \rightarrow \alpha$$

• Extra actions:
  
  • **shift** if goto table transitions between states on a non-terminal
  
  • **accept** if we have matched the goal production
### Action table

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

<table>
<thead>
<tr>
<th>Symbol</th>
<th>ID</th>
<th>$</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ID</td>
<td>$</td>
<td>S</td>
</tr>
</tbody>
</table>

**Wednesday, August 31, 2011**
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:

\[ S \rightarrow A \, y \]
\[ A \rightarrow \lambda \mid x \]

• This leads to the following initial configuration set:

\[ S \rightarrow \cdot A \, y \]
\[ A \rightarrow \cdot x \]
\[ A \rightarrow \lambda \cdot \]

• 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

\[ A \rightarrow X_1 \ldots X_i \cdot X_{i+1} \ldots X_j, l \] (where \( l \in V_t \cup \lambda \))

- If two configurations differ only in their lookahead component, we combine them

\[ A \rightarrow X_1 \ldots X_i \cdot X_{i+1} \ldots X_j, \{l_1 \ldots l_m\} \]
Building configuration sets

- To close a configuration

\[ B \rightarrow \alpha \cdot A \beta, l \]

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

- 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

closure1 (\{S \rightarrow \cdot E \$, \{\lambda\}\}) =

<table>
<thead>
<tr>
<th>Production</th>
<th>Arc</th>
<th>Trailing Symbols</th>
</tr>
</thead>
<tbody>
<tr>
<td>S \rightarrow \cdot E $, {\lambda}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E \rightarrow \cdot E + T, ${}}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E \rightarrow \cdot T, {$}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T \rightarrow \cdot ID, {$}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T \rightarrow \cdot (E), {$}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E \rightarrow \cdot E + T, {+}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E \rightarrow \cdot T, {+}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T \rightarrow \cdot ID, {+}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T \rightarrow \cdot (E), {+}</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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 $\bullet$ is at end of configuration and $x \in$ lookahead set of configuration
  $$A \rightarrow \alpha \bullet, \{\ldots x \ldots\} \in s$$
- $\text{shift}$ when $\bullet$ is before $x$
  $$A \rightarrow \beta \bullet x \gamma \in s$$
Consider the simple grammar:

- \(<\text{program}>\) → begin \(<\text{stmts}>\) end $\
- \(<\text{stmts}>\) → SimpleStmt ; \(<\text{stmts}>\)
- \(<\text{stmts}>\) → begin \(<\text{stmts}>\) end ; \(<\text{stmts}>\)
- \(<\text{stmts}>\) → $\lambda$
**Action and goto tables**

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

- Parse: `begin SimpleStmt ; SimpleStmt ; end $`

<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><code>begin S ; S ; end $</code></td>
<td>Shift 1</td>
</tr>
<tr>
<td>2</td>
<td>0 1</td>
<td><code>S ; S ; end $</code></td>
<td>Shift 5</td>
</tr>
<tr>
<td>3</td>
<td>0 1 5</td>
<td><code>; S ; end $</code></td>
<td>Shift 6</td>
</tr>
<tr>
<td>4</td>
<td>0 1 5 6</td>
<td><code>S ; end $</code></td>
<td>Shift 5</td>
</tr>
<tr>
<td>5</td>
<td>0 1 5 6 5</td>
<td><code>; end $</code></td>
<td>Shift 6</td>
</tr>
<tr>
<td>6</td>
<td>0 1 5 6 5 6</td>
<td><code>end $</code></td>
<td>Reduce 4 (goto 10)</td>
</tr>
<tr>
<td>7</td>
<td>0 1 5 6 5 6 10</td>
<td><code>end $</code></td>
<td>Reduce 2 (goto 10)</td>
</tr>
<tr>
<td>8</td>
<td>0 1 5 6 10</td>
<td><code>end $</code></td>
<td>Reduce 2 (goto 2)</td>
</tr>
<tr>
<td>9</td>
<td>0 1 2</td>
<td><code>end $</code></td>
<td>Shift 3</td>
</tr>
<tr>
<td>10</td>
<td>0 1 2 3</td>
<td><code>$</code></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 in certain cases (we won’t discuss this)