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

\[
\begin{align*}
S & \rightarrow PV P \\
[noun \ fphrase] & \rightarrow RA N \\
[a]R[ticle] & \rightarrow a | the \\
A[djective] & \rightarrow \text{big | black} \\
N[oun] & \rightarrow \text{cat | black} \\
V[erb] & \rightarrow \text{bites | scratches}
\end{align*}
\]

Generating strings

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

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

\[
\begin{align*}
S \Rightarrow A B \$ \Rightarrow A a B \$ \Rightarrow a a B \$ \Rightarrow a a B b \$ \Rightarrow a a b b b \$
\end{align*}
\]

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

Strings are composed of symbols

- **A 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: **a b $** (what is the grammar for this?)

Strings are composed of symbols

- **A 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: **a b $** (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)**
  - 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

- **A → B | C** → **A → B**
- **A → C**
- **D → E {F}** → **D → E Ftail**
- **Ftail → F Ftail**
- **Ftail → λ**
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 → A B $
A → x a A
A → y a A
A → λ
B → b
```

Consider S → A B $ → x a A B $ → x a B $ → x a b $

Tokens that can follow A are called the follow set of A

First and follow sets

- First(α) = {a ∈ Vₐ | $ →* α aβ} ∪ {λ | if $ →* λ}
- Follow(A) = {a ∈ Vₐ | $ →* Aa ...} ∪ {$ | if $ →* A}$

<table>
<thead>
<tr>
<th>S</th>
<th>start symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>a terminal symbol</td>
</tr>
<tr>
<td>A</td>
<td>a non-terminal symbol</td>
</tr>
<tr>
<td>α, β</td>
<td>a string composed of terminals and non-terminals (typically, α is the RHS of a production)</td>
</tr>
</tbody>
</table>

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
  `S → A B $
  A → x a A
  A → y a A
  B → λ
  A → λ
  B → b`
  - First(A) = {x, y}
  - First(xA) = {x}
  - First (AB) = {x, y, b}
  - Follow(A) = {b}

<table>
<thead>
<tr>
<th>S</th>
<th>start symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>a terminal symbol</td>
</tr>
<tr>
<td>A</td>
<td>a non-terminal symbol</td>
</tr>
<tr>
<td>α, β</td>
<td>a string composed of terminals and non-terminals (typically, α is the RHS of a production)</td>
</tr>
</tbody>
</table>

Rightarrow: derived in 1 step
Rightarrow*: derived in 0 or more steps
Rightarrow**: derived in 1 or more steps
Computing first sets

- Terminal: First(a) = \{a\}
- Non-terminal: First(A)
  - Look at all productions for A
    \[ A \rightarrow X_1X_2...X_n \]
  - First(A) \supseteq (First(X_1) \setminus \lambda)
  - If \( \lambda \in \text{First}(X_i) \), First(A) \supseteq (First(X_2) \setminus \lambda)
  - If \( \lambda \) is in First(X_i) for all i, then \( \lambda \in \text{First}(A) \)
- Computing First(\( \lambda \)): 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. \( X \rightarrow \alpha A \beta \text{: Follow}(A) \supseteq (\text{First}(\beta) \setminus \lambda) \)
    2. \( X \rightarrow \alpha A \beta \text{: If } \lambda \in \text{First}(\beta), \text{Follow}(A) \supseteq \text{Follow}(X) \)
    3. \( X \rightarrow \alpha A \text{: Follow}(A) \supseteq \text{Follow}(X) \)
- Note: Follow(X) never has \( \lambda \) in it.

Exercise

- What are the follow sets for

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

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 \( A \rightarrow X_1X_2...X_m \)) applies

\[
\text{Predict}(P) =
\begin{cases}
\text{First}(X_1...X_m) & \text{if } \lambda \notin \text{First}(X_1...X_m) \\
(\text{First}(X_1...X_m) \setminus \lambda) \cup \text{Follow}(X_1...X_m) & \text{otherwise}
\end{cases}
\]
- 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

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:

  \(x a y a b\)

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

- 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 \(x = a + 3\)

```plaintext
statement ::= ID: id = expr = assign
expr ::= term + term = addop
term ::= ID: id = LITERAL = num
```
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
  \[
  \text{<stmt>} \rightarrow \text{<expr> then <stmt list> <if suffix>} \\
  \text{<if suffix>} \rightarrow \text{endif} \\
  \text{<if suffix>} \rightarrow \text{else <stmt list> endif}
  \]

Left recursion

- Left recursion is a problem for LL(1) parsers
- LHS is also the first symbol of the RHS
- Consider:
  \[
  \text{E} \rightarrow \text{E + T} \\
  \text{E} \rightarrow \text{E - E} \\
  \text{E} \rightarrow \text{x}
  \]
- What would happen with the stack-based algorithm?

Removing left recursion

\[
\text{E} \rightarrow \text{E + T} \\
\text{E} \rightarrow \text{T}
\]

Are all grammars LL(k)?

- No! Consider the following grammar:
  \[
  \text{S} \rightarrow \text{E} \\
  \text{E} \rightarrow \text{(E + E)} \\
  \text{E} \rightarrow \text{(E - E)} \\
  \text{E} \rightarrow \text{x}
  \]
- 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 \( \geq \) j)

\[
\text{S} \rightarrow \text{S C} \\
\text{S} \rightarrow \lambda \\
\text{C} \rightarrow [ ] \quad \text{[ ] can be parsed: } \text{SS\lambda C or SS\lambda} \\
\text{(it's ambiguous!)}
\]

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?

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

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

Simple example

1. P \rightarrow S
2. S \rightarrow x ; S
3. S \rightarrow e

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>Shift</td>
</tr>
<tr>
<td>S</td>
<td>Shift</td>
</tr>
<tr>
<td>x</td>
<td>Shift</td>
</tr>
<tr>
<td>:</td>
<td>Shift</td>
</tr>
<tr>
<td>e</td>
<td>Reduce 3</td>
</tr>
<tr>
<td>1</td>
<td>Reduce 2</td>
</tr>
<tr>
<td>2</td>
<td>Accept</td>
</tr>
<tr>
<td>3</td>
<td>4</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

<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>

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 → X₁ ... Xᵢ • Xᵢ₊₁ ...
- The “•” marks the point to which the production has been recognized. In this case, we have recognized X₁ ... Xᵢ
- Configuration set: all the configurations that can apply at a given point during the parse:
  - A → B • CD
  - A → B • GH
  - T → B • 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 → α, add configuration X → •α

Successor configuration set

- Starting with the initial configuration set
  - s₀ = closure₀({S → • E $})
  - 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' = go_to₀(s, X):
    - For each configuration in s of the form A → β • X → γ add
      - A → β • X • Y to t
    - s' = closure₀(t)
**CFSM**

- CFSM = Characteristic Finite State Machine
- Nodes are configuration sets (starting from s0)
- Arcs are 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>3</td>
<td></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
    
    \[ A \to \alpha \cdot a \beta \in s \text{ and } a \in V_t, \text{ else error} \]
  - We **reduce** production P if the * is at the end of a production
    
    \[ B \to \alpha \cdot \varepsilon \in s \text{ where production } P \text{ is } B \to \alpha \]
  - 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>Symbol</th>
<th>ID</th>
<th>$</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>State</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>S</td>
<td>$</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>R2</td>
<td>R2</td>
<td>R2</td>
</tr>
<tr>
<td>2</td>
<td>S</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>A</td>
<td>A</td>
<td>A</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:
  
  \[
  S \to A y \\
  A \to \lambda | x
  \]
- This leads to the following initial configuration set:
  
  \[
  S \to \cdot A y \\
  A \to \cdot x \\
  A \to \lambda \cdot
  \]
- 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 → X₁ ... Xᵢ • Xᵢ₊₁ ... Xⱼ , l (where l ∈ Vt ∪ 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 , l
- Add all configurations of the form A → • Y , 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

<table>
<thead>
<tr>
<th>closure₁(S → • E $, (λ)) =</th>
<th>S → • E $, (λ)</th>
</tr>
</thead>
<tbody>
<tr>
<td>S → E $, (λ)</td>
<td>E → • E + T , (λ)</td>
</tr>
<tr>
<td>E → • E + T , (λ)</td>
<td>E → • T , (λ)</td>
</tr>
<tr>
<td>T → • ID , (λ)</td>
<td>T → • (E) , (λ)</td>
</tr>
<tr>
<td>T → • (E) , (λ)</td>
<td>E → • E + T , (+)</td>
</tr>
<tr>
<td>E → • T , (+)</td>
<td>E → • T , (+)</td>
</tr>
<tr>
<td>T → • ID , (+)</td>
<td>T → • (E) , (+)</td>
</tr>
</tbody>
</table>
Building goto and action tables

- The function $\text{goto1}(\text{configuration-set}, \text{symbol})$ is analogous to $\text{goto0}(\text{configuration-set}, \text{symbol})$ 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 $\star$ is at end of configuration and $x \in$ lookahead set of configuration
    
    $A \rightarrow \alpha \cdot \{\ldots x \ldots\} \in s$
  
  - shift when $\star$ is before $x$
    
    $A \rightarrow \beta \cdot x y \in s$

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></th>
<th>begin</th>
<th>end</th>
<th>SimpleStmt</th>
<th>$&lt;$program$&gt;$</th>
<th>$&lt;$stmts$&gt;$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5/1</td>
<td></td>
<td></td>
<td>$&lt;$program$&gt;$</td>
<td>$&lt;$stmts$&gt;$</td>
</tr>
<tr>
<td>1</td>
<td>5/4</td>
<td>R4</td>
<td>$5/5$</td>
<td>$5/2$</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>5/4</td>
<td>R4</td>
<td>$5/5$</td>
<td>$5/7$</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>5/4</td>
<td>R4</td>
<td>$5/5$</td>
<td>$5/10$</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>5/4</td>
<td>R4</td>
<td>$5/6$</td>
<td>$5/11$</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</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>begin $5;5; \text{ and } $$</td>
<td>Shift 1</td>
</tr>
<tr>
<td>2</td>
<td>0 1</td>
<td>$5;5; \text{ and } $$</td>
<td>Shift 5</td>
</tr>
<tr>
<td>3</td>
<td>0 1 5 6</td>
<td>$;5; \text{ and } $$</td>
<td>Shift 6</td>
</tr>
<tr>
<td>4</td>
<td>0 1 5 6 6</td>
<td>$5; \text{ and } $$</td>
<td>Shift 5</td>
</tr>
<tr>
<td>5</td>
<td>0 1 5 6 6</td>
<td>;end $$</td>
<td>Shift 6</td>
</tr>
<tr>
<td>6</td>
<td>0 1 5 6 6</td>
<td>end $$</td>
<td>Reduce 4 (goto 10)</td>
</tr>
<tr>
<td>7</td>
<td>0 1 5 6 6 10</td>
<td>end $$</td>
<td>Reduce 2 (goto 10)</td>
</tr>
<tr>
<td>8</td>
<td>0 1 5 6 10</td>
<td>end $$</td>
<td>Reduce 2 (goto 2)</td>
</tr>
<tr>
<td>9</td>
<td>0 1 2</td>
<td>end $$</td>
<td>Shift 3</td>
</tr>
<tr>
<td>10</td>
<td>0 1 2 3</td>
<td>$$</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 Follow($A$)
    
    - LALR: merge LR states in certain cases (we won’t discuss this)