

# Parsers

# Agenda

- Terminology
- LL(I) 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:  $V_n \rightarrow \lambda \mid (V_n \mid V_t)^+$
    - 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
  - $V_n = \{S, A, B\}$ 
    - 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

- 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

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

# Generating strings

$S \rightarrow A B \$$

$A \rightarrow A a$

$A \rightarrow a$

$B \rightarrow B b$

$B \rightarrow b$

- 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:

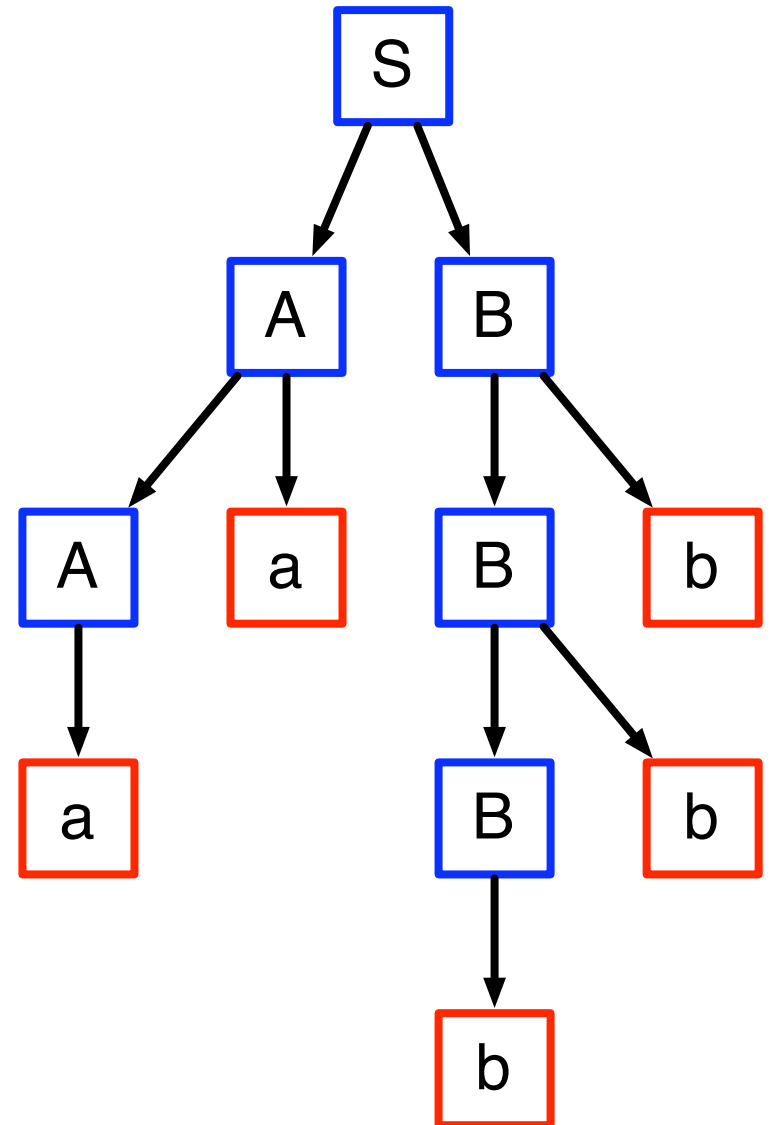
$S \Rightarrow A B \$ \Rightarrow A a B \$ \Rightarrow a a B \$ \Rightarrow a a B b \$ \Rightarrow$   
 $a a B b b \$ \Rightarrow a a b b b \$$

# 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:

E	→	Prefix (E)
E	→	V Tail
Prefix	→	F
Prefix	→	$\lambda$
Tail	→	+ E
Tail	→	$\lambda$

- 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?

$F(V + V)$

E	→	Prefix (E)
E	→	V Tail
Prefix	→	F
Prefix	→	$\lambda$
Tail	→	+ E
Tail	→	$\lambda$

# Simple conversions

$A \rightarrow B \mid C$



$A \rightarrow B$   
 $A \rightarrow C$

$D \rightarrow E \{F\}$



$D \rightarrow E \text{ Ftail}$   
 $\text{Ftail} \rightarrow F \text{ Ftail}$   
 $\text{Ftail} \rightarrow \lambda$

# 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

# 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

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

- A sentence in the grammar:

$$x a c c \$$$



# A simple example

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

• A sentence in the grammar:

$$x a c c \$$$

Current derivation: S

# A simple example

$S \rightarrow A B c \$$

$A \rightarrow x a A$

$A \rightarrow y a A$

$A \rightarrow c$

$B \rightarrow b$

$B \rightarrow \lambda$

• A sentence in the grammar:

$x a c c \$$

Current derivation:  $A B c \$$

Predict rule

# A simple example

$$S \rightarrow A B c \$$$

Choose based on  
*first set* of rules

$$\begin{array}{l} A \rightarrow x a A \\ A \rightarrow y a A \\ A \rightarrow c \end{array}$$

$$B \rightarrow b$$

• A sentence in the grammar:

$$B \rightarrow \lambda$$

$x a c c \$$

Current derivation:  $x a A B c \$$

Predict rule *based on next token*

# A simple example

$S \rightarrow A B c \$$

$A \rightarrow x a A$

$A \rightarrow y a A$

$A \rightarrow c$

$B \rightarrow b$

$B \rightarrow \lambda$

• A sentence in the grammar:

$x a c c \$$

Current derivation:  $x a A B c \$$

Match token

# A simple example

$S \rightarrow A B c \$$

$A \rightarrow x a A$

$A \rightarrow y a A$

$A \rightarrow c$

$B \rightarrow b$

$B \rightarrow \lambda$

• A sentence in the grammar:

$x a c c \$$

Current derivation:  $x a A B c \$$

Match token

# A simple example

$$S \rightarrow A B c \$$$

Choose based on  
*first set* of rules

$$\begin{array}{l} A \rightarrow x a A \\ A \rightarrow y a A \\ A \rightarrow c \end{array}$$

$$B \rightarrow b$$

• A sentence in the grammar:

$$B \rightarrow \lambda$$

$x a c c \$$

Current derivation:  $x a c B c \$$

Predict rule *based on next token*

# A simple example

$S \rightarrow A B c \$$

$A \rightarrow x a A$

$A \rightarrow y a A$

$A \rightarrow c$

$B \rightarrow b$

$B \rightarrow \lambda$

• A sentence in the grammar:

$x a c c \$$

Current derivation: **x a c** B c \$

**Match token**

# A simple example

$S \rightarrow A B c \$$

$A \rightarrow x a A$

Choose based on  
*follow set*

$A \rightarrow y a A$

$A \rightarrow c$

$B \rightarrow b$

$B \rightarrow \lambda$

• A sentence in the grammar:

$x a c c \$$

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

Predict rule *based on next token*



# A simple example

$S \rightarrow A B c \$$

$A \rightarrow x a A$

$A \rightarrow y a A$

$A \rightarrow c$

$B \rightarrow b$

$B \rightarrow \lambda$

• A sentence in the grammar:

$x a c c \$$

Current derivation: **x a c c \$**

**Match token**

# A simple example

$S \rightarrow A B c \$$

$A \rightarrow x a A$

$A \rightarrow y a A$

$A \rightarrow c$

$B \rightarrow b$

$B \rightarrow \lambda$

• A sentence in the grammar:

$x a c c \$$

Current derivation: **x a c c \$**

**Match token**

# First and follow sets

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

$$S \rightarrow A B \$$$

$$A \rightarrow x a A$$

$$A \rightarrow y a A$$

$$A \rightarrow \lambda$$

$$B \rightarrow b$$

# First and follow sets

- $\text{First}(\alpha) = \{a \in V_t \mid \alpha \Rightarrow^* a\beta\} \cup \{\lambda \mid \text{if } \alpha \Rightarrow^* \lambda\}$
- $\text{Follow}(A) = \{a \in V_t \mid S \Rightarrow^+ \dots Aa \dots\} \cup \{\$ \mid \text{if } S \Rightarrow^+ \dots A \$\}$

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

# Computing first sets

- Terminal:  $\text{First}(a) = \{a\}$
- Non-terminal:  $\text{First}(A)$ 
  - Look at all productions for  $A$   
$$A \rightarrow X_1 X_2 \dots X_k$$
  - $\text{First}(A) \supseteq (\text{First}(X_1) - \lambda)$
  - If  $\lambda \in \text{First}(X_1)$ ,  $\text{First}(A) \supseteq (\text{First}(X_2) - \lambda)$
  - If  $\lambda$  is in  $\text{First}(X_i)$  for all  $i$ , then  $\lambda \in \text{First}(A)$
- Computing  $\text{First}(\alpha)$ : similar procedure to computing  $\text{First}(A)$

# 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

- $\text{Follow}(S) = \{\}$
- To compute  $\text{Follow}(A)$ :
  - Find productions which have  $A$  on rhs. Three rules:
    1.  $X \rightarrow \alpha A \beta$ :  $\text{Follow}(A) \supseteq (\text{First}(\beta) - \lambda)$
    2.  $X \rightarrow \alpha A \beta$ : 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:  $\text{Follow}(X)$  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  $A \rightarrow X_1 X_2 \dots X_m$ ) applies

$\text{Predict}(P) =$

$$\begin{cases} \text{First}(X_1 \dots X_m) & \text{if } \lambda \notin \text{First}(X_1 \dots X_m) \\ (\text{First}(X_1 \dots X_m) - \lambda) \cup \text{Follow}(A) & \text{otherwise} \end{cases}$$

- 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:  $T[A][t] = \text{//initialize all fields to "error"}$

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.  $B \rightarrow b$

# 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

# An example

- How would a stack-based parser parse:

x a y a b

1.  $S \rightarrow A B \$$

2.  $A \rightarrow x a A$

3.  $A \rightarrow y a A$

4.  $A \rightarrow \lambda$

5.  $B \rightarrow b$

Parse stack	Remaining input	Parser action
S	x a y a b \$	predict 1

# An example

- How would a stack-based parser parse:

$x a y a b$

1.  $S \rightarrow A B \$$

2.  $A \rightarrow x a A$

3.  $A \rightarrow y a A$

4.  $A \rightarrow \lambda$

5.  $B \rightarrow b$

Parse stack	Remaining input	Parser action
S	$x a y a b \$$	predict 1
A B \$	$x a y a b \$$	predict 2

# An example

- How would a stack-based parser parse:

$x a y a b$

1.  $S \rightarrow A B \$$

2.  $A \rightarrow x a A$

3.  $A \rightarrow y a A$

4.  $A \rightarrow \lambda$

5.  $B \rightarrow b$

Parse stack	Remaining input	Parser action
S	x a y a b \$	predict 1
A B \$	x a y a b \$	predict 2
x a A B \$	x a y a b \$	match(x)

# An example

- How would a stack-based parser parse:

$x a y a b$

1.  $S \rightarrow A B \$$

2.  $A \rightarrow x a A$

3.  $A \rightarrow y a A$

4.  $A \rightarrow \lambda$

5.  $B \rightarrow b$

Parse stack	Remaining input	Parser action
S	x a y a b \$	predict 1
A B \$	x a y a b \$	predict 2
x a A B \$	x a y a b \$	match(x)
a A B \$	a y a b \$	match(a)



# An example

- How would a stack-based parser parse:

$x a y a b$

1.  $S \rightarrow A B \$$

2.  $A \rightarrow x a A$

3.  $A \rightarrow y a A$

4.  $A \rightarrow \lambda$

5.  $B \rightarrow b$

Parse stack	Remaining input	Parser action
$S$	$x a y a b \$$	predict 1
$A B \$$	$x a y a b \$$	predict 2
$x a A B \$$	$x a y a b \$$	match(x)
$a A B \$$	$a y a b \$$	match(a)
$A B \$$	$y a b \$$	predict 3

# An example

- How would a stack-based parser parse:

$x a y a b$

1.  $S \rightarrow A B \$$

2.  $A \rightarrow x a A$

3.  $A \rightarrow y a A$

4.  $A \rightarrow \lambda$

5.  $B \rightarrow b$

Parse stack	Remaining input	Parser action
$S$	$x a y a b \$$	predict 1
$A B \$$	$x a y a b \$$	predict 2
$x a A B \$$	$x a y a b \$$	match(x)
$a A B \$$	$a y a b \$$	match(a)
$A B \$$	$y a b \$$	predict 3
$y a A B \$$	$y a b \$$	match(y)

# An example

- How would a stack-based parser parse:

$x a y a b$

1.  $S \rightarrow A B \$$

2.  $A \rightarrow x a A$

3.  $A \rightarrow y a A$

4.  $A \rightarrow \lambda$

5.  $B \rightarrow b$

Parse stack	Remaining input	Parser action
$S$	$x a y a b \$$	predict 1
$A B \$$	$x a y a b \$$	predict 2
$x a A B \$$	$x a y a b \$$	match(x)
$a A B \$$	$a y a b \$$	match(a)
$A B \$$	$y a b \$$	predict 3
$y a A B \$$	$y a b \$$	match(y)
$a A B \$$	$a b \$$	match(a)

# An example

- How would a stack-based parser parse:

$x a y a b$

1.  $S \rightarrow A B \$$

2.  $A \rightarrow x a A$

3.  $A \rightarrow y a A$

4.  $A \rightarrow \lambda$

5.  $B \rightarrow b$

Parse stack	Remaining input	Parser action
$S$	$x a y a b \$$	predict 1
$A B \$$	$x a y a b \$$	predict 2
$x a A B \$$	$x a y a b \$$	match(x)
$a A B \$$	$a y a b \$$	match(a)
$A B \$$	$y a b \$$	predict 3
$y a A B \$$	$y a b \$$	match(y)
$a A B \$$	$a b \$$	match(a)
$A B \$$	$b \$$	predict 4

# An example

- How would a stack-based parser parse:

$x a y a b$

1.  $S \rightarrow A B \$$

2.  $A \rightarrow x a A$

3.  $A \rightarrow y a A$

4.  $A \rightarrow \lambda$

5.  $B \rightarrow b$

Parse stack	Remaining input	Parser action
$S$	$x a y a b \$$	predict 1
$A B \$$	$x a y a b \$$	predict 2
$x a A B \$$	$x a y a b \$$	match(x)
$a A B \$$	$a y a b \$$	match(a)
$A B \$$	$y a b \$$	predict 3
$y a A B \$$	$y a b \$$	match(y)
$a A B \$$	$a b \$$	match(a)
$A B \$$	$b \$$	predict 4
$B \$$	$b \$$	predict 5

# An example

- How would a stack-based parser parse:

$x a y a b$

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

Parse stack	Remaining input	Parser action
S	x a y a b \$	predict 1
A B \$	x a y a b \$	predict 2
x a A B \$	x a y a b \$	match(x)
a A B \$	a y a b \$	match(a)
A B \$	y a b \$	predict 3
y a A B \$	y a b \$	match(y)
a A B \$	a b \$	match(a)
A B \$	b \$	predict 4
B \$	b \$	predict 5
b \$	b \$	match(b)

# An example

- How would a stack-based parser parse:

$x a y a b$

1.  $S \rightarrow A B \$$

2.  $A \rightarrow x a A$

3.  $A \rightarrow y a A$

4.  $A \rightarrow \lambda$

5.  $B \rightarrow b$

Parse stack	Remaining input	Parser action
$S$	$x a y a b \$$	predict 1
$A B \$$	$x a y a b \$$	predict 2
$x a A B \$$	$x a y a b \$$	match(x)
$a A B \$$	$a y a b \$$	match(a)
$A B \$$	$y a b \$$	predict 3
$y a A B \$$	$y a b \$$	match(y)
$a A B \$$	$a b \$$	match(a)
$A B \$$	$b \$$	predict 4
$B \$$	$b \$$	predict 5
$b \$$	$b \$$	match(b)
$\$$	$\$$	Done!

# 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  $x = a + 3$

```
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

$\langle \text{stmt} \rangle \rightarrow \text{if } \langle \text{expr} \rangle \text{ then } \langle \text{stmt list} \rangle \text{ endif}$

$\langle \text{stmt} \rangle \rightarrow \text{if } \langle \text{expr} \rangle \text{ then } \langle \text{stmt list} \rangle \text{ else } \langle \text{stmt list} \rangle \text{ endif}$

- This is not LL(1) (why?)

- We can turn this in to

$\langle \text{stmt} \rangle \rightarrow \text{if } \langle \text{expr} \rangle \text{ then } \langle \text{stmt list} \rangle \langle \text{if suffix} \rangle$

$\langle \text{if suffix} \rangle \rightarrow \text{endif}$

$\langle \text{if suffix} \rangle \rightarrow \text{else } \langle \text{stmt list} \rangle \text{ endif}$

# 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

$E \rightarrow E + T$   
 $E \rightarrow T$



$E \rightarrow EI \text{ Etail}$   
 $EI \rightarrow T$   
 $\text{Etail} \rightarrow + T \text{ Etail}$   
 $\text{Etail} \rightarrow \lambda$

# 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:  
$$A \rightarrow ax \mid ay$$
- More lookahead leads to more powerful parsers
- What are the downsides?

# Are all grammars LL(k)?

- No! Consider the following grammar:

$$\begin{aligned} S &\rightarrow E \\ E &\rightarrow (E + E) \\ E &\rightarrow (E - E) \\ E &\rightarrow x \end{aligned}$$

- 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”:  $[^i ]^j$  ( $i \geq j$ )

S → [ S C  
S → λ  
C → ]  
C → λ

[ [ ] can be parsed: SSλC or SSCλ  
(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?

$$\begin{aligned} S &\rightarrow [ S \\ S &\rightarrow SI \\ SI &\rightarrow [ SI ] \\ SI &\rightarrow \lambda \end{aligned}$$

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”

$$\begin{array}{l} S \rightarrow [ S C \\ S \rightarrow \lambda \\ C \rightarrow ] \\ C \rightarrow \lambda \end{array}$$

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

$$\begin{array}{l} S \rightarrow \text{if } S \text{ E} \\ S \rightarrow \text{other} \\ E \rightarrow \text{else } S \text{ endif} \\ E \rightarrow \text{endif} \end{array}$$

# 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 **L**eft-to-right, **R**ight-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.  $S \rightarrow e$

		Symbol					Action
		x	;	e	P	S	
State	0	1		3		5	Shift
	1		2				Shift
	2	1		3		4	Shift
	3						Reduce 3
	4						Reduce 2
	5						Accept

# 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”

Step	Parse Stack	Remaining Input	Parser Action
1	0	x ; x ; e	Shift 1
2	0 1	; x ; e	Shift 2
3	0 1 2	x ; e	Shift 1
4	0 1 2 1	; e	Shift 2
5	0 1 2 1 2	e	Shift 3
6	0 1 2 1 2 3		Reduce 3 (goto 4)
7	0 1 2 1 2 4		Reduce 2 (goto 4)
8	0 1 2 4		Reduce 2 (goto 5)
9	0 5		Accept



# 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 “•”

$$A \rightarrow X_1 \dots X_i \cdot X_{i+1} \dots X_j$$

- The “•” marks the point to which the production has been recognized. In this case, we have recognized  $X_1 \dots X_i$

- 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  $X \rightarrow \bullet \alpha$

```
S → E $  
E → E + T | T  
T → ID | (E)
```

```
closure0({S → • E $}) = {  
  S → • E $  
  E → • E + T  
  E → • T  
  T → • ID  
  T → • (E  
}
```

# Successor configuration set

- Starting with the initial configuration set

$$s_0 = \text{closure}_0(\{S \rightarrow \cdot \alpha \$\})$$

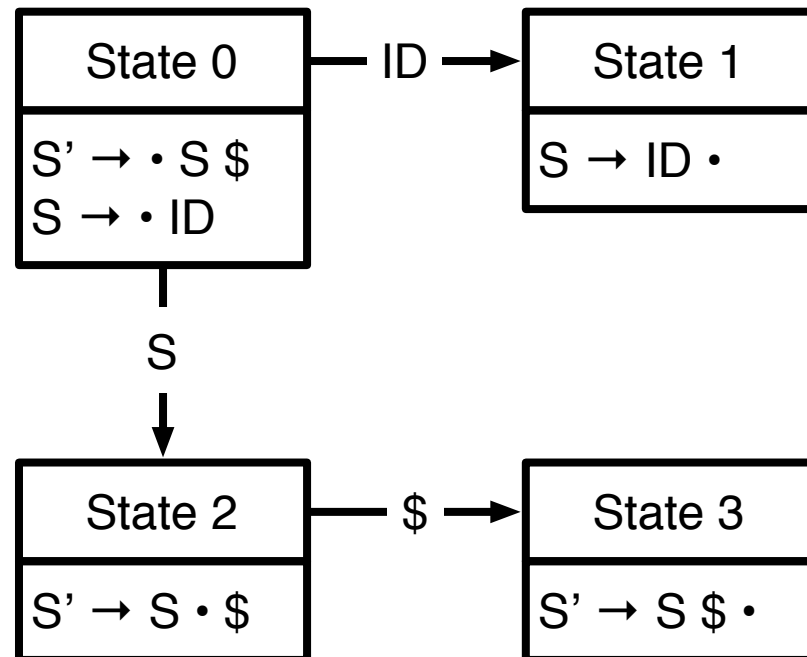
an LR(0) parser will find the successor given the next symbol  $X$

- $X$  can be either a terminal (the next token from the scanner) or a non-terminal (the result of applying a reduction)
- Determining the successor  $s' = \text{go\_to}_0(s, X)$ :
  - 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 s0)
- Arcs are go\_to relationships

$S' \rightarrow S \$$   
 $S \rightarrow ID$



# Building the goto table

- We can just read this off from the CFSM

		Symbol		
		ID	\$	S
State	0	1		2
	1			
	2		3	
	3			

# Building the action table

- Given the configuration set  $s$ :
  - We **shift** if the next token matches a terminal after the  $\bullet$  in some configuration
$$A \rightarrow \alpha \bullet a \beta \in s \text{ and } a \in V_t, \text{ else error}$$
  - We **reduce** production  $P$  if the  $\bullet$  is at the end of a production
$$B \rightarrow \alpha \bullet \in s \text{ where production } P \text{ is } B \rightarrow \alpha$$
  - Extra actions:
    - **shift** if goto table transitions between states on a non-terminal
    - **accept** if we have matched the goal production

# Action table

State	0	Shift
	1	Reduce 2
	2	Shift
	3	Accept



# 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”):

$$A \rightarrow x \cdot x$$

$$A \rightarrow x \cdot$$

- 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:

$$S \rightarrow \cdot A y$$

$$A \rightarrow \cdot x$$

$$A \rightarrow \lambda \cdot$$

- 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(I) parser**
  - Built using similar techniques as LR(0) parsers, but uses lookahead to distinguish states
  - LR(I) 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(I) and LALR(I)
    - Differ in how they resolve ambiguities
    - yacc and bison produce LALR(I) parsers

# LR(l) parsing

- Configurations in LR(l) look similar to LR(0), but they are extended to include a lookahead symbol

$$A \rightarrow X_1 \dots X_i \cdot X_{i+1} \dots X_j, l \text{ (where } l \in V_t \cup \lambda \text{)}$$

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

$$A \rightarrow X_1 \dots X_i \cdot X_{i+1} \dots X_j, \{l_1 \dots l_m\}$$

# 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

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

$S \rightarrow E \$$   
 $E \rightarrow E + T \mid T$   
 $T \rightarrow ID \mid (E)$

# Example

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

$S \rightarrow \cdot E \$, \{\lambda\}$

$S \rightarrow E \$$   
 $E \rightarrow E + T \mid T$   
 $T \rightarrow ID \mid (E)$

# Example

$S \rightarrow E \$$   
 $E \rightarrow E + T \mid T$   
 $T \rightarrow ID \mid (E)$

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

$S \rightarrow \cdot E \$, \{\lambda\}$

$E \rightarrow \cdot E + T, \{\$\}$

# Example

$S \rightarrow E \$$   
 $E \rightarrow E + T \mid T$   
 $T \rightarrow ID \mid (E)$

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

$S \rightarrow \cdot E \$, \{\lambda\}$

$E \rightarrow \cdot E + T, \{\$\}$

$E \rightarrow \cdot T, \{\$\}$

# Example

$S \rightarrow E \$$   
 $E \rightarrow E + T \mid T$   
 $T \rightarrow ID \mid (E)$

$\text{closure}(\{S \rightarrow \bullet E \$, \{\lambda\}\}) =$

$S \rightarrow \bullet E \$, \{\lambda\}$

$E \rightarrow \bullet E + T, \{\$\}$

$E \rightarrow \bullet T, \{\$\}$

$T \rightarrow \bullet ID, \{\$\}$

# Example

$S \rightarrow E \$$   
 $E \rightarrow E + T \mid T$   
 $T \rightarrow ID \mid (E)$

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

$S \rightarrow \cdot E \$, \{\lambda\}$

$E \rightarrow \cdot E + T, \{\$\}$

$E \rightarrow \cdot T, \{\$\}$

$T \rightarrow \cdot ID, \{\$\}$

$T \rightarrow \cdot (E), \{\$\}$

# Example

$S \rightarrow E \$$   
 $E \rightarrow E + T \mid T$   
 $T \rightarrow ID \mid (E)$

$\text{closure}(\{S \rightarrow \cdot E \$, \{\lambda\}\}) =$
$S \rightarrow \cdot E \$, \{\lambda\}$
$E \rightarrow \cdot E + T, \{\$\}$
$E \rightarrow \cdot T, \{\$\}$
$T \rightarrow \cdot ID, \{\$\}$
$T \rightarrow \cdot (E), \{\$\}$
$E \rightarrow \cdot E + T, \{+\}$

# Example

$S \rightarrow E \$$   
 $E \rightarrow E + T \mid T$   
 $T \rightarrow ID \mid (E)$

$\text{closure}(\{S \rightarrow \cdot E \$, \{\lambda\}\}) =$
$S \rightarrow \cdot E \$, \{\lambda\}$
$E \rightarrow \cdot E + T, \{\$\}$
$E \rightarrow \cdot T, \{\$\}$
$T \rightarrow \cdot ID, \{\$\}$
$T \rightarrow \cdot (E), \{\$\}$
$E \rightarrow \cdot E + T, \{+\}$
$E \rightarrow \cdot T, \{+\}$



# Example

$S \rightarrow E \$$   
 $E \rightarrow E + T \mid T$   
 $T \rightarrow ID \mid (E)$

$\text{closure}(\{S \rightarrow \cdot E \$, \{\lambda\}\}) =$
$S \rightarrow \cdot E \$, \{\lambda\}$
$E \rightarrow \cdot E + T, \{\$\}$
$E \rightarrow \cdot T, \{\$\}$
$T \rightarrow \cdot ID, \{\$\}$
$T \rightarrow \cdot (E), \{\$\}$
$E \rightarrow \cdot E + T, \{+\}$
$E \rightarrow \cdot T, \{+\}$
$T \rightarrow \cdot ID, \{+\}$

# Example

$S \rightarrow E \$$   
 $E \rightarrow E + T \mid T$   
 $T \rightarrow ID \mid (E)$

$\text{closure}(\{S \rightarrow \cdot E \$, \{\lambda\}\}) =$
$S \rightarrow \cdot E \$, \{\lambda\}$
$E \rightarrow \cdot E + T, \{\$\}$
$E \rightarrow \cdot T, \{\$\}$
$T \rightarrow \cdot ID, \{\$\}$
$T \rightarrow \cdot (E), \{\$\}$
$E \rightarrow \cdot E + T, \{+\}$
$E \rightarrow \cdot T, \{+\}$
$T \rightarrow \cdot ID, \{+\}$
$T \rightarrow \cdot (E), \{+\}$

# Building goto and action tables

- The function **goto** (configuration-set, symbol) is analogous to **goto** (configuration-set, symbol) for LR(0)
- Build goto table in the same way as for LR(0)
- Key difference: the action table.

action[s][x] =

- **reduce** when • is at end of configuration *and*  $x \in$  lookahead set of configuration

$$A \rightarrow \alpha \bullet, \{\dots x \dots\} \in s$$

- **shift** when • is before  $x$

$$A \rightarrow \beta \bullet x \gamma \in s$$

# Example

- Consider the simple grammar:

$\langle \text{program} \rangle \rightarrow \text{begin } \langle \text{stmts} \rangle \text{ end } \$$

$\langle \text{stmts} \rangle \rightarrow \text{SimpleStmt } ; \langle \text{stmts} \rangle$

$\langle \text{stmts} \rangle \rightarrow \text{begin } \langle \text{stmts} \rangle \text{ end } ; \langle \text{stmts} \rangle$

$\langle \text{stmts} \rangle \rightarrow \lambda$

# Action and goto tables

	begin	end	;	SimpleStmt	\$	<program>	<stmts>
0	S / 1						
1	S / 4	R4		S / 5			S / 2
2		S / 3					
3					A		
4	S / 4	R4		S / 5			S / 7
5			S / 6				
6	S / 4	R4		S / 5			S / 10
7		S / 8					
8			S / 9				
9	S / 4	R4		S / 6			S / 11
10		R2					
11		R3					

<program> → begin <stmts> end \$

<stmts> → SimpleStmt ; <stmts>

<stmts> → begin <stmts> end ; <stmts>

<stmts> →  $\lambda$

# Example

- Parse: begin SimpleStmt ; SimpleStmt ; end \$

Step	Parse Stack	Remaining Input	Parser Action
1	0	begin S ; S ; end \$	Shift 1
2	0 1	S ; S ; end \$	Shift 5
3	0 1 5	; S ; end \$	Shift 6
4	0 1 5 6	S ; end \$	Shift 5
5	0 1 5 6 5	; end \$	Shift 6
6	0 1 5 6 5 6	end \$	Reduce 4 (goto 10)
7	0 1 5 6 5 6 10	end \$	Reduce 2 (goto 10)
8	0 1 5 6 10	end \$	Reduce 2 (goto 2)
9	0 1 2	end \$	Shift 3
10	0 1 2 3	\$	Accept

# Problems with LR(I) parsers

- LR(I) 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)