

# Compilers

## *CS414-2015-03*

### *Parsing*

David Galles

Department of Computer Science  
University of San Francisco

## 03-0: Parsing

---

- Once we have broken an input file into a sequence of tokens, the next step is to determine if that sequence of tokens forms a syntactically correct program – parsing
- *Parsing* a sequence of tokens == determining if the string of tokens could be generated by a Context-Free Grammar.

## 03-1: CFG Example

---

$S \rightarrow \text{print}(E);$

$S \rightarrow \text{while } (E) \ S$

$S \rightarrow \{ L \}$

$E \rightarrow \text{identifier}$

$E \rightarrow \text{integer\_literal}$

$L \rightarrow SL$

$L \rightarrow \epsilon$

Examples / Parse Trees

## 03-2: Recursive Descent Parser

---

- Write Java code that repeatedly calls getNextToken(), and determines if the stream of returned tokens can be generated by the CFG
  - If so, end normally
  - If not, call an error function

## 03-3: Recursive Descent Parser

---

A Recursive Descent Parser is implemented as a suite of recursive functions, one for each non-terminal in the grammar:

- ParseS will terminate normally if the next tokens in the input stream can be derived from the non-terminal  $S$
- ParseL will terminate normally if the next tokens in the input stream can be derived from the non-terminal  $L$
- ParseE will terminate normally if the next tokens in the input stream can be derived from the non-terminal  $E$

## 03-4: Recursive Descent Parser

---

$S \rightarrow \text{print}(E);$

$S \rightarrow \text{while } (E) \ S$

$S \rightarrow \{ L \}$

$E \rightarrow \text{identifier}$

$E \rightarrow \text{integer\_literal}$

$L \rightarrow SL$

$L \rightarrow \epsilon$

Code for Parser.java.html on web browser

## 03-5: LL(1) Parsers

---

These recursive descent parsers are also known as LL(1) parsers, for Left-to-right, Leftmost derivation, with 1 symbol lookahead

- The input file is read from left to right (starting with the first symbol in the input stream, and proceeding to the last symbol).
- The parser ensures that a string can be derived by the grammar by building a leftmost derivation.
- Which rule to apply at each step is decided upon after looking at just 1 symbol.

## 03-6: Building LL(1) Parsers

---

$$S' \rightarrow S\$$$

$$S \rightarrow AB$$

$$S \rightarrow Ch$$

$$A \rightarrow ef$$

$$A \rightarrow \epsilon$$

$$B \rightarrow hg$$

$$C \rightarrow DD$$

$$C \rightarrow fi$$

$$D \rightarrow g$$

ParseS    use rule  $S \rightarrow AB$  on e, h  
          use the rule  $S \rightarrow Ch$  on f, g



## 03-7: First sets

$\text{First}(S)$  is the set of all terminals that can start strings derived from  $S$  (plus  $\epsilon$ , if  $S$  can produce  $\epsilon$ )

$$S' \rightarrow S\$$$

$$S \rightarrow AB$$

$$S \rightarrow Ch$$

$$A \rightarrow ef$$

$$A \rightarrow \epsilon$$

$$B \rightarrow hg$$

$$C \rightarrow DD$$

$$C \rightarrow fi$$

$$D \rightarrow g$$

$$\text{First}(S') =$$

$$\text{First}(S) =$$

$$\text{First}(A) =$$

$$\text{First}(B) =$$

$$\text{First}(C) =$$

$$\text{First}(D) =$$

## 03-8: First sets

$\text{First}(S)$  is the set of all terminals that can start strings derived from  $S$  (plus  $\epsilon$ , if  $S$  can produce  $\epsilon$ )

$$S' \rightarrow S\$$$

$$\text{First}(S') = \{e, f, g, h\}$$

$$S \rightarrow AB$$

$$\text{First}(S) = \{e, f, g, h\}$$

$$S \rightarrow Ch$$

$$\text{First}(A) = \{e, \epsilon\}$$

$$A \rightarrow ef$$

$$\text{First}(B) = \{h\}$$

$$A \rightarrow \epsilon$$

$$\text{First}(C) = \{f, g\}$$

$$B \rightarrow hg$$

$$\text{First}(D) = \{g\}$$

$$C \rightarrow DD$$

$$C \rightarrow fi$$

$$D \rightarrow g$$

## 03-9: First sets

---

We can expand the definition of First sets to include strings of terminals and non-terminals

$$S' \rightarrow S\$$$

$$S \rightarrow AB$$

$$S \rightarrow Ch$$

$$A \rightarrow ef$$

$$A \rightarrow \epsilon$$

$$B \rightarrow hg$$

$$C \rightarrow DD$$

$$C \rightarrow fi$$

$$D \rightarrow g$$

$$\text{First}(aB) =$$

$$\text{First}(BC) =$$

$$\text{First}(AbC) =$$

$$\text{First}(AC) =$$

$$\text{First}(abS) =$$

$$\text{First}(DDA) =$$

## 03-10: First sets

We can expand the definition of First sets to include strings of terminals and non-terminals

$$S' \rightarrow S\$$$

$$S \rightarrow AB$$

$$S \rightarrow Ch$$

$$A \rightarrow ef$$

$$A \rightarrow \epsilon$$

$$B \rightarrow hg$$

$$C \rightarrow DD$$

$$C \rightarrow fi$$

$$D \rightarrow g$$

$$\text{First}(aB) = \{a\}$$

$$\text{First}(BC) = \{h\}$$

$$\text{First}(AbC) = \{e, b\}$$

$$\text{First}(AC) = \{e, f, g\}$$

$$\text{First}(abS) = \{a\}$$

$$\text{First}(DDA) = \{g\}$$

## 03-11: Calculating First sets

---

- For each non-terminal  $S$ , set  $\text{First}(S) = \{\}$
- For each rule of the form  $S \rightarrow \gamma$ , add  $\text{First}(\gamma)$  to  $\text{First}(S)$
- Repeat until no changes are made

## 03-12: Calculating First sets

---

Example:

$$S' \rightarrow S\$$$

$$S \rightarrow AB$$

$$S \rightarrow Ch$$

$$A \rightarrow ef$$

$$A \rightarrow \epsilon$$

$$B \rightarrow hg$$

$$C \rightarrow DD$$

$$C \rightarrow fi$$

$$D \rightarrow g$$

## 03-13: Follow Sets

$\text{Follow}(S)$  is the set of all terminals that can follow  $S$  in a (partial) derivation.

$$S' \rightarrow S\$$$

$$S \rightarrow AB$$

$$S \rightarrow Ch$$

$$A \rightarrow ef$$

$$A \rightarrow \epsilon$$

$$B \rightarrow hg$$

$$C \rightarrow DD$$

$$C \rightarrow fi$$

$$D \rightarrow g$$

$$\text{Follow}(S') =$$

$$\text{Follow}(S) =$$

$$\text{Follow}(A) =$$

$$\text{Follow}(B) =$$

$$\text{Follow}(C) =$$

$$\text{Follow}(D) =$$

## 03-14: Follow Sets

$\text{Follow}(S)$  is the set of all terminals that can follow  $S$  in a (partial) derivation.

$$S' \rightarrow S\$$$

$$\text{Follow}(S') = \{ \}$$

$$S \rightarrow AB$$

$$\text{Follow}(S) = \{\$ \}$$

$$S \rightarrow Ch$$

$$\text{Follow}(A) = \{h\}$$

$$A \rightarrow ef$$

$$\text{Follow}(B) = \{\$ \}$$

$$A \rightarrow \epsilon$$

$$\text{Follow}(C) = \{h\}$$

$$B \rightarrow hg$$

$$\text{Follow}(D) = \{h, g\}$$

$$C \rightarrow DD$$

$$C \rightarrow fi$$

$$D \rightarrow g$$



## 03-15: Calculating Follow sets

---

- For each non-terminal  $S$ , set  $\text{Follow}(S) = \{\}$
- For each rule of the form  $S \rightarrow \gamma$ 
  - For each non-terminal  $S_1$  in  $\gamma$ , where  $\gamma$  is of the form  $\alpha S_1 \beta$ 
    - If  $\text{First}(\beta)$  does not contain  $\epsilon$ , add all elements of  $\text{First}(\beta)$  to  $\text{Follow}(S_1)$ .
    - If  $\text{First}(\beta)$  does contain  $\epsilon$ , add all elements of  $\text{First}(\beta)$  *except*  $\epsilon$  to  $\text{Follow}(S_1)$ , and add all elements of  $\text{Follow}(S)$  to  $\text{Follow}(S_1)$ .
- If any changes were made, repeat.

## 03-16: Calculating Follow sets

---

Example:

$$S' \rightarrow S\$$$

$$S \rightarrow AB$$

$$S \rightarrow Ch$$

$$A \rightarrow ef$$

$$A \rightarrow \epsilon$$

$$B \rightarrow hg$$

$$C \rightarrow DD$$

$$C \rightarrow fi$$

$$D \rightarrow g$$

## 03-17: Parse Tables

---

- Each row in a parse table is labeled with a non-terminal
- Each column in a parse table is labeled with a terminal
- Each element in a parse table is either empty, or contains a grammar rule
  - Rule  $S \rightarrow \gamma$  goes in row  $S$ , in all columns of  $\text{First}(\gamma)$ .
  - If  $\text{First}(\gamma)$  contains  $\epsilon$ , then the rule  $S \rightarrow \gamma$  goes in row  $S$ , in all columns of  $\text{Follow}(S)$ .

## 03-18: Parse Table Example

$S' \rightarrow S\$$	$\text{First}(S') = \{e, f, g, h\}$	$\text{Follow}(S') = \{ \}$
$S \rightarrow AB$	$\text{First}(S) = \{e, f, g, h\}$	$\text{Follow}(S) = \{\$ \}$
$S \rightarrow Ch$	$\text{First}(A) = \{e, \epsilon\}$	$\text{Follow}(A) = \{h\}$
$A \rightarrow ef$	$\text{First}(B) = \{h\}$	$\text{Follow}(B) = \{\$ \}$
$A \rightarrow \epsilon$	$\text{First}(C) = \{f, g\}$	$\text{Follow}(C) = \{h\}$
$B \rightarrow hg$	$\text{First}(D) = \{g\}$	$\text{Follow}(D) = \{h, g\}$
$C \rightarrow DD$		
$C \rightarrow fi$		
$D \rightarrow g$		

## 03-19: Parse Table Example

	e	f	g	h	i
$S'$	$S' \rightarrow S\$$	$S' \rightarrow S\$$	$S' \rightarrow S\$$	$S' \rightarrow S\$$	
$S$	$S \rightarrow AB$	$S \rightarrow Ch$	$S \rightarrow Ch$	$S \rightarrow AB$	
$A$	$A \rightarrow ef$			$A \rightarrow \epsilon$	
$B$				$B \rightarrow hg$	
$C$		$C \rightarrow fi$	$C \rightarrow DD$		
$D$			$D \rightarrow g$		

# 03-20: Parse Table Example

---

```
void ParseA() {  
    switch(currentToken) {  
        case e:  
            checkToken(e);  
            checkToken(f);  
            break;  
        case h:  
            /* epsilon case */  
            break;  
        otherwise:  
            error("Parse Error");  
    }  
}  
  
void ParseC() {  
    switch(currentToken) {  
        case f:  
            checkToken(f);  
            checkToken(i);  
            break;  
        case g:  
            ParseD();  
            ParseD();  
            break;  
        otherwise:  
            error("Parse Error");  
    }  
}
```

## 03-21: LL(1) Parser Example

---

$$Z' \rightarrow Z\$$$

$$Z \rightarrow XYZ \mid d$$

$$X \rightarrow a \mid Y$$

$$Y \rightarrow \epsilon \mid c$$

(Initial Symbol =  $Z'$ )

## 03-22: LL(1) Parser Example

---

$Z' \rightarrow Z\$$        $\text{First}(Z') = \{a, c, d\}$

$Z \rightarrow XYZ \mid d$        $\text{First}(Z) = \{a, c, d\}$

$X \rightarrow a \mid Y$        $\text{First}(X) = \{a, c, \epsilon\}$

$Y \rightarrow \epsilon \mid c$        $\text{First}(Y) = \{c, \epsilon\}$

(Initial Symbol =  $Z'$ )



## 03-23: LL(1) Parser Example

---

$Z' \rightarrow Z\$$	$\text{First}(Z') = \{a, c, d\}$	$\text{Follow}(Z') = \{ \}$
$Z \rightarrow XYZ \mid d$	$\text{First}(Z) = \{a, c, d\}$	$\text{Follow}(Z) = \{\$ \}$
$X \rightarrow a \mid Y$	$\text{First}(X) = \{a, c, \epsilon\}$	$\text{Follow}(X) = \{a, c, d\}$
$Y \rightarrow \epsilon \mid c$	$\text{First}(Y) = \{c, \epsilon\}$	$\text{Follow}(Y) = \{a, c, d\}$

(Initial Symbol =  $Z'$ )

# 03-24: LL(1) Parser Example

$Z' \rightarrow Z\$$   
 $Z \rightarrow XYZ \mid d$   
 $X \rightarrow a \mid Y$   
 $Y \rightarrow \epsilon \mid c$

$\text{First}(Z') = \{a, c, d\}$   
 $\text{First}(Z) = \{a, c, d\}$   
 $\text{First}(X) = \{a, c, \epsilon\}$   
 $\text{First}(Y) = \{c, \epsilon\}$

$\text{Follow}(Z') = \{ \}$   
 $\text{Follow}(Z) = \{\$ \}$   
 $\text{Follow}(X) = \{a, c, d\}$   
 $\text{Follow}(Y) = \{a, c, d\}$

	a	c	d
$Z'$	$Z' \rightarrow Z\$$	$Z' \rightarrow Z\$$	$Z' \rightarrow Z\$$
$Z$	$Z \rightarrow XYZ$	$Z \rightarrow XYZ$	$Z \rightarrow XYZ$ $Z \rightarrow d$
$X$	$X \rightarrow a$ $X \rightarrow Y$	$X \rightarrow Y$	$X \rightarrow Y$
$Y$	$Y \rightarrow \epsilon$	$Y \rightarrow c$ $Y \rightarrow \epsilon$	$Y \rightarrow \epsilon$

## 03-25: non-LL(1) Grammars

---

- Not all grammars can be parsed by a LL(1) parser
- A grammar is LL(1) if the LL(1) parse table contains no duplicate entries
- Previous CFG is *not* LL(1)

## 03-26: non-LL(1) Grammars

---

- Not all grammars can be parsed by a LL(1) parser
- A grammar is LL(1) if the LL(1) parse table contains no duplicate entries
- Previous CFG is *not* LL(1)
  - No ambiguous grammar is LL(1)

## 03-27: LL(1) Parser Example

---

$$S' \rightarrow S\$$$

$$S \rightarrow \text{if } E \text{ then } S \text{ else } S$$

$$S \rightarrow \text{begin } L \text{ end}$$

$$S \rightarrow \text{print}(E)$$

$$L \rightarrow \epsilon$$

$$L \rightarrow SL'$$

$$L' \rightarrow ; SL'$$

$$L' \rightarrow \epsilon$$

$$E \rightarrow \text{num} = \text{num}$$

## 03-28: LL(1) Parser Example

Non-Terminal	First	Follow
$S'$	{if, begin, print}	{ }
$S$	{if, begin, print}	{\$, end, ;}
$L$	{ $\epsilon$ , if, begin, print}	{end}
$L'$	{ $\epsilon$ , ;}	{end}
$E$	{num}	{)}

# 03-29: LL(1) Parser Example

	if	then	else	begin	end	print
$S'$	$S' \rightarrow S\$$			$S' \rightarrow S\$$		$S' \rightarrow S\$$
$S$	$S \rightarrow \text{if } E \text{ then } S \text{ else } S$			$S \rightarrow \text{begin } L \text{ end}$		$S \rightarrow \text{print}(E)$
$L$	$L \rightarrow SL'$			$S' \rightarrow S\$$	$L \rightarrow \epsilon$	$S' \rightarrow S\$$
$L'$						
$E$						

	(	)	;	num	=
$S'$					
$S$					
$L$					
$L'$			$L' \rightarrow ; SL'$		
$E$				$E \rightarrow \text{num} = \text{num}$	

## 03-30: LL(1) Parser Example

---

$$S' \rightarrow S\$$$

$$S \rightarrow ABC$$

$$A \rightarrow a$$

$$A \rightarrow \epsilon$$

$$B \rightarrow b$$

$$B \rightarrow \epsilon$$

$$C \rightarrow c$$

$$C \rightarrow \epsilon$$



# 03-31: LL(1) Parser Example

Non-terminal	First	Follow
$S'$	{a, b, c}	{}
$S$	{a, b, c}	{ $\$$ }
$A$	{a}	{b, c, $\$$ }
$B$	{b}	{c, $\$$ }
$C$	{c}	{ $\$$ }

	a	b	c	$\$$
$S'$	$S' \rightarrow S\$$	$S' \rightarrow S\$$	$S' \rightarrow S\$$	
$S$	$S \rightarrow ABC$	$S \rightarrow ABC$	$S \rightarrow ABC$	$S \rightarrow ABC$
$A$	$A \rightarrow a$	$A \rightarrow \epsilon$	$A \rightarrow \epsilon$	$A \rightarrow \epsilon$
$B$		$B \rightarrow b$	$B \rightarrow \epsilon$	$B \rightarrow \epsilon$
$C$			$C \rightarrow c$	$C \rightarrow \epsilon$

## 03-32: Creating an LL(1) CFG

---

- Not all grammars are LL(1)
- We can often modify a CFG that is not LL(1)
  - New CFG generates the same language as the old CFG
  - New CFG is LL(1)

## 03-33: Creating an LL(1) CFG

---

- Remove Ambiguity
  - No ambiguous grammar is LL(1)
  - Grammar is ambiguous if there are two ways to generate the same string
  - If there are two ways to generate a string  $\alpha$ , modify the CFG so that one of the ways to generate the string is removed

## 03-34: Removing Ambiguity

---

- Often a grammar is ambiguous when there is a special case rule that can be generated by a general rule
- Solution: Remove the special case, let the general case handle it

$S \rightarrow V = E;$

$S \rightarrow V = \text{identifier};$

$E \rightarrow V$

$E \rightarrow \text{integer\_literal}$

$V \rightarrow \text{identifier}$

Structured variable definitions commonly have this problem

## 03-35: Removing Ambiguity

---

- This grammar, for describing variable accesses, is also ambiguous
  - Ambiguous in the same way as expression CFG
  - Can be made unambiguous in a similar fashion

$$V \rightarrow V . V$$
$$V \rightarrow \text{identifier}$$

## 03-36: Removing Ambiguity

---

- Some *Languages* are inherently ambiguous
- A Language  $L$  is ambiguous if:
  - For each CFG  $G$  that generates  $L$ ,  $G$  is ambiguous
- No programming languages are inherently ambiguous

## 03-37: Left Recursion

---

$$S \rightarrow S\alpha$$

$$S \rightarrow \beta$$

- Any CFG that contains these rules (where  $\alpha$  and  $\beta$  are any string of terminals and non-terminals) is *not* LL(1)
- Why?

## 03-38: Left Recursion

---

$S \rightarrow Sa$

$S \rightarrow b$

- What should ParseS() do on a b?



## 03-39: Left Recursion

$$S \rightarrow Sa$$

$$S \rightarrow b$$

	First	Follow
$S$	{b}	{a}

	a	b
$S$		$S \rightarrow Sa$ $S \rightarrow b$

## 03-40: Left Recursion

---

$$S \rightarrow Sa$$

$$S \rightarrow b$$

- What strings can be derived from this grammar?

## 03-41: Left Recursion

---

$$S \rightarrow Sa$$
$$S \rightarrow b$$

- What strings can be derived from this grammar?

$$S \Rightarrow b$$
$$S \Rightarrow Sa \Rightarrow ba$$
$$S \Rightarrow Sa \Rightarrow Saa \Rightarrow baa$$
$$S \Rightarrow Sa \Rightarrow Saa \Rightarrow Saaa \Rightarrow baaa$$
$$S \Rightarrow Sa \Rightarrow Saa \Rightarrow Saaa \Rightarrow Saaaa \Rightarrow baaaa$$

## 03-42: Removing Left Recursion

---

$$S \rightarrow Sa$$
$$S \rightarrow b$$

- What strings can be derived from this grammar?
  - A b, followed by zero or more a's.
- What is a CFG for this language that does not use Left Recursion?

## 03-43: Removing Left Recursion

---

$$S \rightarrow Sa$$

$$S \rightarrow b$$

- What strings can be derived from this grammar?
  - A b, followed by zero or more a's.
- What is a CFG for this language that does not use Left Recursion?

$$S \rightarrow bS'$$

$$S' \rightarrow aS'$$

$$S' \rightarrow \epsilon$$

## 03-44: Removing Left Recursion

---

$$S \rightarrow Sa$$
$$S \rightarrow b$$

- What strings can be derived from this grammar?
  - A b, followed by zero or more a's.
- What is an EBNF for this language that does not use Left Recursion?

## 03-45: Removing Left Recursion

---

$$S \rightarrow Sa$$
$$S \rightarrow b$$

- What strings can be derived from this grammar?
  - A b, followed by zero or more a's.
- What is an EBNF for this language that does not use Left Recursion?

$$S \rightarrow b(a)^*$$

## 03-46: Removing Left Recursion

---

- In General, if you have rules of the form:

$$S \rightarrow S\alpha$$

$$S \rightarrow \beta$$

- You can replace them with the CFG rules

$$S \rightarrow \beta S'$$

$$S' \rightarrow \alpha S'$$

$$S' \rightarrow \epsilon$$

or the EBNF rules

$$S \rightarrow \beta(\alpha)^*$$



## 03-47: Removing Left Recursion

---

- What about:

$$S \rightarrow S\alpha_1$$

$$S \rightarrow S\alpha_2$$

$$S \rightarrow \beta_1$$

$$S \rightarrow \beta_2$$

## 03-48: Removing Left Recursion

---

- What about:

$$S \rightarrow S\alpha_1$$

$$S \rightarrow S\alpha_2$$

$$S \rightarrow \beta_1$$

$$S \rightarrow \beta_2$$

$$S \rightarrow BA$$

$$B \rightarrow \beta_1$$

$$B \rightarrow \beta_2$$

$$A \rightarrow \alpha_1 A$$

$$A \rightarrow \alpha_2 A$$

$$A \rightarrow \epsilon$$

We can use the same method for arbitrarily complex grammars

## 03-49: Left Factoring

---

- Consider Fortran DO statements:

Fortran:

```
do
```

```
    var = intial, final
```

```
    loop body
```

```
end do
```

Java Equivalent:

```
for (var=initial; var <= final; var++) {
```

```
    loop body
```

```
}
```

## 03-50: Left Factoring

---

- Consider Fortran DO statements:

Fortran:

```
do  
    var = intial, final, inc  
    loop body  
end do
```

Java Equivalent:

```
for (var=initial; var <= final; var+=inc) {  
    loop body  
}
```

## 03-51: Left Factoring

---

- CFG for Fortran DO statements:  
 $S \rightarrow \text{do } L S$   
 $L \rightarrow \text{id} = \text{exp}, \text{exp}$   
 $L \rightarrow \text{id} = \text{exp}, \text{exp}, \text{exp}$
- Is this Grammar LL(1)?

## 03-52: Left Factoring

---

- CFG for Fortran DO statements:  
 $S \rightarrow \text{do } L S$   
 $L \rightarrow \text{id} = \text{exp}, \text{exp}$   
 $L \rightarrow \text{id} = \text{exp}, \text{exp}, \text{exp}$
- Is this Grammar LL(1)? No!
- The problem is in the rules for  $L$ 
  - Two rules for  $L$  that start out exactly the same
  - No way to know which rule to apply when looking at just 1 symbol

## 03-53: Left Factoring

---

- Factor out the similar sections from the rules:

$$S \rightarrow \text{do } L \ S$$

$$L \rightarrow \text{id} = \text{exp}, \text{exp}$$

$$L \rightarrow \text{id} = \text{exp}, \text{exp}, \text{exp}$$

## 03-54: Left Factoring

---

- Factor out the similar sections from the rules:

$$S \rightarrow \text{do } L S$$

$$L \rightarrow \text{id} = \text{exp}, \text{exp } L'$$

$$L' \rightarrow , \text{exp}$$

$$L' \rightarrow \epsilon$$

- We can also use EBNF:

$$S \rightarrow \text{do } L S$$

$$L \rightarrow \text{id} = \text{exp}, \text{exp } (, \text{exp})?$$



## 03-55: Left Factoring

---

- In general, if we have rules of the form:

$$S \rightarrow \alpha \beta_1$$

$$S \rightarrow \alpha \beta_2$$

...

$$S \rightarrow \alpha \beta_n$$

- We can left factor these rules to get:

$$S \rightarrow \alpha B$$

$$B \rightarrow \beta_1$$

$$B \rightarrow \beta_2$$

...

$$B \rightarrow \beta_n$$

## 03-56: Building an LL(1) Parser

---

- Create a CFG for the language
- Remove ambiguity from the CFG, remove left recursion, and left-factor it
- Find First/Follow sets for all non-terminals
- Build the LL(1) parse table
- Use the parse table to create a suite of mutually recursive functions

## 03-57: Building an LL(1) Parser

---

- Create an EBNF for the language
- Remove ambiguity from the EBNF, remove left recursion, and left-factor it
- ~~Find First/Follow sets for all non-terminals~~
- ~~Build the LL(1) parse table~~
- ~~Use the parse table to create a suite of mutually recursive functions~~
- Use a parser generator tool that converts the EBNF into parsing functions

# 03-58: Structure JavaCC file foo.jj

---

```
options{
    /* Code to set various options flags */
}

PARSER_BEGIN(foo)

public class foo {
    /* This segment is often empty */
}

PARSER_END(foo)

TOKEN_MGR_DECLS :
{
    /* Declarations used by lexical analyzer */
}

/* Token Rules & Actions */

/* JavaCC Rules and Actions -- EBNF for language*/
```

## 03-59: JavaCC Rules

---

- JavaCC rules correspond to EBNF rules
- JavaCC rules have the form:

```
void nonTerminalName() :  
{ /* Java Declarations */ }  
{ /* Rule definition */  
}
```

- For now, the Java Declarations section will be empty (we will use it later on, when building parse trees)
- Non terminals in JavaCC rules are followed by ()
- Terminals in JavaCC rules are between < and >

## 03-60: JavaCC Rules

---

- For example, the CFG rules:

$$S \rightarrow \text{while } (E) S$$
$$S \rightarrow V = E;$$

- Would be represented by the JavaCC rule:

```
void statement() :  
{}  
{  
    <WHILE> <LPAREN> expression() <RPAREN> statement()  
|   variable() <GETS> expression() <SEMICOLON>  
}
```

## 03-61: Example JavaCC Files

- We now examine a JavaCC file that parses prefix expressions
- Prefix expression examples:

Prefix expression	Equivalent infix expression
+ 3 4	3 + 4
+ - 2 1 5	(2 - 1) + 5
+ - 3 4 * 5 6	(3 - 4) + (5 * 6)
+ - * 3 4 5 6	((3 * 4) - 5) + 6
+ 3 - 4 * 5 6	3 + (4 - (5 * 6))

- Come up with a CFG for Prefix expressions

## 03-62: Example JavaCC Files

---

- CFG for prefix expressions:

$$E \rightarrow + E E$$

$$E \rightarrow - E E$$

$$E \rightarrow * E E$$

$$E \rightarrow / E E$$

$$E \rightarrow \text{num}$$

Pull up JavaCC file on other screen



## 03-63: Example JavaCC Files

---

- In javacc format:

```
void expression():  
{  
    {  
        <PLUS>    expression() expression()  
    |    <MINUS>   expression() expression()  
    |    <TIMES>   expression() expression()  
    |    <DIVIDE>  expression() expression()  
    |    <INTEGER_LITERAL>  
    }  
}
```

Put up driver file, do some examples

## 03-64: Lookahead

---

- Consider the following JavaCC fragment:

```
void S():  
{  
    "a" "b" "c"  
    |  "a" "d" "c"  
}
```

- Is this grammar LL(1)?

## 03-65: Lookahead

---

- A LOOKAHEAD directive, placed at a choice point, will allow JavaCC to use a lookahead  $> 1$ :

```
void S():  
{  
  { LOOKAHEAD(2) "a" "b" "c"  
    | "a" "d" "c"  
  }  
}
```

- ParseS will look at the next *two* symbols, before deciding which rule for S to use

## 03-66: Lookahead

---

- LOOKAHEAD directives are placed at “choice points” – places in the grammar where there is more than one possible rule that can match.

```
void S():  
{  
    "A" (("B" "C") | ("B" "D"))  
}
```

## 03-67: Lookahead

---

- LOOKAHEAD directives are placed at “choice points” – places in the grammar where there is more than one possible rule that can match.

```
void S():  
{  
    "A" ( LOOKAHEAD(2) ("B" "C") | ("B" "D"))  
}
```

## 03-68: Lookahead

---

- LOOKAHEAD directives are placed at “choice points” – places in the grammar where there is more than one possible rule that can match.

```
void S():  
{  
    {  
        "A" (("B" "C") | ("B" "D"))  
    }  
}
```

## 03-69: Lookahead

---

- LOOKAHEAD directives are placed at “choice points” – places in the grammar where there is more than one possible rule that can match.

```
void S():  
{  
    LOOKAHEAD(2) "A" (("B" "C") | ("B" "D"))  
}
```

- This is **not** a valid use of lookahead – the grammar will not be parsed correctly. Why not?

## 03-70: JavaCC & Non-LL(k)

---

- JavaCC will produce a parser for grammars that are not LL(1) (and even for grammars that are not LL(k), for any k)
- The parser that is produced is not guaranteed to correctly parse the language described by the grammar
- A warning will be issued when JavaCC is run on a non-LL(1) grammar



## 03-71: JavaCC & Non-LL(k)

---

- What does JavaCC do for a non-LL(1) grammar?
  - The rule that appears *first* will be used

```
void S() :  
{  
  {  
    "a" "b" "c"  
  |  "a" "b" "d"  
  }  
}
```

# 03-72: JavaCC & Non-LL(k)

---

- Infamous dangling else

```
void statement():  
{  
    <IF> expression() <THEN> statement()  
    | <IF> expression() <THEN> statement() <ELSE> statement()  
    | /* Other statement definitions */  
}
```

- Why doesn't this grammar work?

# 03-73: JavaCC & Non-LL(k)

---

```
void statement() :
{
    <IF> expression() <THEN> statement() optionalelse()
|
    /* Other statement definitions */
}

void optionalelse() :
{
    <ELSE> statement()
|
    /* nothing */ { }
}

if <e> then <S>
if <e> then <S> else <S>
if <e> then if <e> then <S> else <S>
```

# 03-74: JavaCC & Non-LL(k)

---

```
void statement() :
{
    <IF> expression() <THEN> statement() optionalelse()
|
    /* Other statement definitions */
}
void optionalelse() :
{
    /* nothing */ { }
|
    <ELSE> statement()
}
```

- What about this grammar?

# 03-75: JavaCC & Non-LL(k)

---

```
void statement() :
{
{
    <IF> expression() <THEN> statement() optionalelse()
|
    /* Other statement definitions */
}
}

void optionalelse() :
{
{
    /* nothing */ { }
|
    <ELSE> statement()
}
}
```

- What about this grammar?
- Doesn't work! (why?)

## 03-76: JavaCC & Non-LL(k)

---

```
void statement() :  
{  
  {  
    <IF> expression() <THEN> statement() (<ELSE> statement)?  
  |  
    /* Other statement definitions */  
  }  
}
```

- This grammar will also work correctly
- Also produces a warning

# 03-77: JavaCC & Non-LL(k)

---

```
void statement() :  
{  
  {  
    <IF> expression() <THEN> statement()  
      (LOOKAHEAD(1) <ELSE> statement)?  
  |  
    /* Other statement definitions */  
  }  
}
```

- This grammar also works correctly
- Produces no warnings
  - (not because it is any more safe – if you include a LOOKAHEAD directive, the system assumes you know what you are doing)

## 03-78: Parsing Project

---

- For your next project, you will write a parser for simpleJava using JavaCC.
- Provided Files:
  - ParseTest.java A main program to test your parser **You must use program as your starting non-terminal for ParseTest to work correctly!**
  - test\*.sjava Various simpleJava programs to test your parser. Some have parsing errors, some do not. **Be sure to test your parser on other test cases, too! These files are not meant to be exhaustive!!**



## 03-79: Parsing Project “Gotcha’s”

---

- Expressions can be tricky. Read the text for more examples and suggestions
- Structured variable accesses are similar to expressions, and have some of the same issues
- Avoid specific cases that can be handled by a more general case

## 03-80: Parsing Project “Gotcha’s”

---

- Procedure calls and assignment statements can be tricky for LL(1) parsers. You may need to left-factor, and/or use LOOKAHEAD directives
- LOOKAHEAD directives are useful, but can be dangerous (for instance, you will not get warnings for the sections that use LOOKAHEAD.) Try left-factoring, or other techniques, before resorting to LOOKAHEAD.
- This project is much more difficult than the lexical analyzer. **Start Early!**