# 02247 Compiler Construction
# Modules 6 & 7: Parsing

Andrea Vandin
(*Slides based on: Swami Iyer, UMass Boston, http://www.swamiiyer.net/cs451/slides.html*)

## Tentative Lecture Plan

| # | *Date* | *Topic* | *Chapter* | *Suggested project plan* |
|---|---|---|---|---|
| 1 | Feb 1 | Course Introduction & Compilers Overview | 1 | |
| 2 | Feb 8 | The *j*-- Compiler & Demo of Final Project | 1 | |
| 3 | Feb 15 | Lexical Analysis | 2 | |
| 4 | Feb 22 | Lexical Analysis 2 | 2 | |
| 5 | Mar 1 | Lab Day | | Complete Steps 0-1 |
| 6 | Mar 8 | Parsing | 3 | |
| 7 | Mar 15 | Parsing 2 | 3 | Begin Steps 2-3 |
| 8 | Mar 22 | Type Checking | 4 | |
| | Mar 29 | *Easter holiday* | | |
| 9 | Apr 5 | Type Checking 2 | 4 | Begin Step 4 |
| 10 | Apr 12 | Lab Day | | Complete Steps 2-4 |
| 11 | Apr 19 | JVM Code Generation | 5 | Begin Step 5 |
| 12 | Apr 26 | MIPS Code Generation | 6 | |
| 13 | May 3 | Register Allocation | 7 | |
| | ? | Translating functional languages | | |
| | May 10 | Deadline for final assignment | | Project and report ready |
| | May 18 | Exam | | |

# The j-- compiler

# The j-- compiler

# The j-- compiler

# The j-- compiler

# The j-- compiler

# The j-- compiler

# Where are we

Source
Language
Program
compilationUnit()
Scanner
nextToken()
Parser
Abstract
Syntax
Tree
(AST)
preAnalyze()
analyze()
codegen()
Main
JVM
.class
file(s)

# Where are we

Source
Language
Program
compilationUnit()
Scanner
nextToken()
Parser
Abstract
Syntax
Tree
(AST)
preAnalyze()
analyze()
codegen()
Main
JVM
.class
file(s)

# Outline


1. **Introduction**
2. **Context-free Grammars and Languages**
   - Backus-Naur Form (BNF)
   - Left- and right-most derivation, & Parse tree
   - Ambiguity
3. **Top-down Deterministic Parsing**
   - Example: Top-down deterministic parsing of a *j*-- program
   - Parsing by Recursive Descent
   - Parsing by the LL(1) technique
4. **Transform a non-LL(1) grammar into an equivalent LL(1) one**
5. **Parser Generation Using JavaCC**

## Outline


1. **Introduction**
2. **Context-free Grammars and Languages**
   - Backus-Naur Form (BNF)
   - Left- and right-most derivation, & Parse tree
   - Ambiguity
3. **Top-down Deterministic Parsing**
   - Example: Top-down deterministic parsing of a *j*-- program
   - Parsing by Recursive Descent
   - Parsing by the LL(1) technique
4. **Transform a non-LL(1) grammar into an equivalent LL(1) one**
5. **Parser Generation Using JavaCC**

## Parsing

Once we have identified the tokens in our program, we then want to determine its syntactic structure. This process is called parsing

We wish to make sure the program is syntactically valid, ie, it conforms to the grammar that describes its syntax

As the parser parses the program it should identify syntax errors and report them and the line numbers they appear on

More *semantic* errors (e.g. type mismatches), will be ignored by the parser, and considered in later compilation phases.

When the parser does find a syntax error, it should not just stop, but it should report the error, and gracefully recover so that it may go on looking for additional errors

The parser should produce some representation of the parsed program that is suitable for semantic analysis; in the *j*-- compiler, we produce an abstract syntax tree (AST)

## Parsing

Once we have identified the tokens in our program, we then want to determine its syntactic structure. This process is called parsing

We wish to make sure the program is syntactically valid, ie, it conforms to the grammar that describes its syntax

As the parser parses the program it should identify syntax errors and report them and the line numbers they appear on

More *semantic* errors (e.g. type mismatches), will be ignored by the parser, and considered in later compilation phases.

When the parser does find a syntax error, it should not just stop, but it should report the error, and gracefully recover so that it may go on looking for additional errors

The parser should produce some representation of the parsed program that is suitable for semantic analysis; in the *j*-- compiler, we produce an abstract syntax tree (AST)

## Parsing

Once we have identified the tokens in our program, we then want to determine its syntactic structure. This process is called parsing

We wish to make sure the program is syntactically valid, ie, it conforms to the grammar that describes its syntax

As the parser parses the program it should identify syntax errors and report them and the line numbers they appear on

More *semantic* errors (e.g. type mismatches), will be ignored by the parser, and considered in later compilation phases.

When the parser does find a syntax error, it should not just stop, but it should report the error, and gracefully recover so that it may go on looking for additional errors

The parser should produce some representation of the parsed program that is suitable for semantic analysis; in the *j*-- compiler, we produce an abstract syntax tree (AST)

## Parsing

Once we have identified the tokens in our program, we then want to determine its syntactic structure. This process is called parsing

We wish to make sure the program is syntactically valid, ie, it conforms to the grammar that describes its syntax

As the parser parses the program it should identify syntax errors and report them and the line numbers they appear on

More *semantic* errors (e.g. type mismatches), will be ignored by the parser, and considered in later compilation phases.

When the parser does find a syntax error, it should not just stop, but it should report the error, and gracefully recover so that it may go on looking for additional errors

The parser should produce some representation of the parsed program that is suitable for semantic analysis; in the *j*-- compiler, we produce an abstract syntax tree (AST)

## Parsing

Once we have identified the tokens in our program, we then want to determine its syntactic structure. This process is called parsing

We wish to make sure the program is syntactically valid, ie, it conforms to the grammar that describes its syntax

As the parser parses the program it should identify syntax errors and report them and the line numbers they appear on

More *semantic* errors (e.g. type mismatches), will be ignored by the parser, and considered in later compilation phases.

When the parser does find a syntax error, it should not just stop, but it should report the error, and gracefully recover so that it may go on looking for additional errors

The parser should produce some representation of the parsed program that is suitable for semantic analysis; in the *j*-- compiler, we produce an abstract syntax tree (AST)

## Parsing

Once we have identified the tokens in our program, we then want to determine its syntactic structure. This process is called parsing

We wish to make sure the program is syntactically valid, ie, it conforms to the grammar that describes its syntax

As the parser parses the program it should identify syntax errors and report them and the line numbers they appear on

More *semantic* errors (e.g. type mismatches), will be ignored by the parser, and considered in later compilation phases.

When the parser does find a syntax error, it should not just stop, but it should report the error, and gracefully recover so that it may go on looking for additional errors

The parser should produce some representation of the parsed program that is suitable for semantic analysis; in the *j*-- compiler, we produce an abstract syntax tree (AST)

## The Factorial program and its AST

For example, given the following *j*-- program

```
package pass;

import java.lang.System;

public class Factorial {
    // Two methods and a field
    public static int factorial(int n) {
        if (n <= 0)
            return 1;
        else
            return n * factorial(n - 1);
    }

    public static void main(String[] args) {
        int x = n;
        System.out.println(x + "! = " + factorial(x));
    }

    static int n = 5;
}
```

we would like to produce an AST shown in the following slide

# The `Factorial` program and its AST

## The Factorial program and its AST

```
package pass;
import java.lang.System;

public class Factorial {
    public static int factorial(int n) {
        if (n <= 0)
            return 1;
        else
            return n * factorial(n - 1);
    }

    public static void main(String[] args) {
        int x = n;
        System.out.println(x + "! = " + factorial(x));
    }

    static int n = 5;
}
```

## The Abstract Syntax Tree

The nodes in the AST represent syntactic objects

The AST is rooted at a `JCompilationUnit`, the syntactic object representing the program that we are compiling

The directed edges are labeled by the names of the fields they represent; for example, `JCompilationUnit` has a package name, a list (an `ArrayList`) of imported types, and a list (an `ArrayList`) of class declarations

We are interested in a tree representation for our program because it is easier to analyze and decorate (with type information) a tree than it is to do the same with text

The syntax in the program text is implicit. The AST makes such syntax explicit. This is the purpose of parsing.

## The Abstract Syntax Tree

The nodes in the AST represent syntactic objects

The AST is rooted at a `JCompilationUnit`, the syntactic object representing the program that we are compiling

The directed edges are labeled by the names of the fields they represent; for example, `JCompilationUnit` has a package name, a list (an `ArrayList`) of imported types, and a list (an `ArrayList`) of class declarations

We are interested in a tree representation for our program because it is easier to analyze and decorate (with type information) a tree than it is to do the same with text

The syntax in the program text is implicit. The AST makes such syntax explicit. This is the purpose of parsing.

## The Abstract Syntax Tree

The nodes in the AST represent syntactic objects

The AST is rooted at a `JCompilationUnit`, the syntactic object representing the program that we are compiling

The directed edges are labeled by the names of the fields they represent; for example, `JCompilationUnit` has a package name, a list (an `ArrayList`) of imported types, and a list (an `ArrayList`) of class declarations

We are interested in a tree representation for our program because it is easier to analyze and decorate (with type information) a tree than it is to do the same with text

The syntax in the program text is implicit. The AST makes such syntax explicit. This is the purpose of parsing.

## The Abstract Syntax Tree

The nodes in the AST represent syntactic objects

The AST is rooted at a `JCompilationUnit`, the syntactic object representing the program that we are compiling

The directed edges are labeled by the names of the fields they represent; for example, `JCompilationUnit` has a package name, a list (an `ArrayList`) of imported types, and a list (an `ArrayList`) of class declarations

We are interested in a tree representation for our program because it is easier to analyze and decorate (with type information) a tree than it is to do the same with text

The syntax in the program text is implicit. The AST makes such syntax explicit. This is the purpose of parsing.

## The Abstract Syntax Tree

The nodes in the AST represent syntactic objects

The AST is rooted at a `JCompilationUnit`, the syntactic object representing the program that we are compiling

The directed edges are labeled by the names of the fields they represent; for example, `JCompilationUnit` has a package name, a list (an `ArrayList`) of imported types, and a list (an `ArrayList`) of class declarations

We are interested in a tree representation for our program because it is easier to analyze and decorate (with type information) a tree than it is to do the same with text

The syntax in the program text is implicit. The AST makes such syntax explicit. This is the purpose of parsing.

## Outline

1. Introduction
2. Context-free Grammars and Languages
   - Backus-Naur Form (BNF)
   - Left- and right-most derivation, & Parse tree
   - Ambiguity
3. Top-down Deterministic Parsing
   - Example: Top-down deterministic parsing of a *j*-- program
   - Parsing by Recursive Descent
   - Parsing by the LL(1) technique
4. Transform a non-LL(1) grammar into an equivalent LL(1) one
5. Parser Generation Using JavaCC

## Context-free Grammars

The grammars that we use to describe programming languages are inherently recursive and are best described by what we call context-free grammars, using a notation called Backus-Naur Form (BNF)

For example, the context-free rule

$$S ::= \texttt{if}\ (E)\ S \qquad\qquad S \text{ “can be written as” } \texttt{if}\ (E)\ S$$

says that, if “$E$” is an *expression* and “$S$” is a *statement*, then

$$\texttt{if}\ (E)\ S$$

is also a *statement*

There are abbreviations possible in the notation; for example, we can write

$$\begin{aligned} S ::= & \ \texttt{if}\ (E)\ S \\ & \mid \texttt{if}\ (E)\ S\ \texttt{else}\ S \end{aligned}$$

as shorthand for

$$\begin{aligned} S &::= \texttt{if}\ (E)\ S \\ S &::= \texttt{if}\ (E)\ S\ \texttt{else}\ S \end{aligned}$$

## Context-free Grammars

The grammars that we use to describe programming languages are inherently recursive and are best described by what we call context-free grammars, using a notation called Backus-Naur Form (BNF)

For example, the context-free rule

$$S ::= \texttt{if (}E\texttt{)}\ S$$

$S$ "can be written as" $\texttt{if (}E\texttt{)}\ S$

says that, if "$E$" is an *expression* and "$S$" is a *statement*, then

$$\texttt{if (}E\texttt{)}\ S$$

is also a *statement*

There are abbreviations possible in the notation; for example, we can write

$$\begin{aligned} S ::= & \texttt{if (}E\texttt{)}\ S \\ & \mid \texttt{if (}E\texttt{)}\ S\ \texttt{else}\ S \end{aligned}$$

as shorthand for

$$\begin{aligned} S &::= \texttt{if (}E\texttt{)}\ S \\ S &::= \texttt{if (}E\texttt{)}\ S\ \texttt{else}\ S \end{aligned}$$

## Context-free Grammars

The grammars that we use to describe programming languages are inherently recursive and are best described by what we call context-free grammars, using a notation called Backus-Naur Form (BNF)

For example, the context-free rule

$$S ::= \texttt{if (}E\texttt{)}\ S$$

$S$ "can be written as" $\texttt{if (}E\texttt{)}\ S$

says that, if "$E$" is an *expression* and "$S$" is a *statement*, then

$$\texttt{if (}E\texttt{)}\ S$$

is also a *statement*

There are abbreviations possible in the notation; for example, we can write

$$\begin{aligned} S ::= & \texttt{if (}E\texttt{)}\ S \\ & \mid \texttt{if (}E\texttt{)}\ S\ \texttt{else}\ S \end{aligned}$$

as shorthand for

$$\begin{aligned} S &::= \texttt{if (}E\texttt{)}\ S \\ S &::= \texttt{if (}E\texttt{)}\ S\ \texttt{else}\ S \end{aligned}$$

## Context-free Grammars

The grammars that we use to describe programming languages are inherently recursive and are best described by what we call context-free grammars, using a notation called Backus-Naur Form (BNF)

For example, the context-free rule

$$S ::= \texttt{if (}E\texttt{)}\ S \qquad\qquad S \text{ ``can be written as'' } \texttt{if (}E\texttt{)}\ S$$

says that, if "$E$" is an *expression* and "$S$" is a *statement*, then

$$\texttt{if (}E\texttt{)}\ S$$

is also a *statement*

There are abbreviations possible in the notation; for example, we can write

$$\begin{aligned} S ::= & \texttt{if (}E\texttt{)}\ S \\ & \mid \texttt{if (}E\texttt{)}\ S\ \texttt{else}\ S \end{aligned}$$

as shorthand for

$$\begin{aligned} S ::= & \texttt{if (}E\texttt{)}\ S \\ S ::= & \texttt{if (}E\texttt{)}\ S\ \texttt{else}\ S \end{aligned}$$

## Context-free Grammars

The grammars that we use to describe programming languages are inherently recursive and are best described by what we call context-free grammars, using a notation called Backus-Naur Form (BNF)

For example, the context-free rule

$$S ::= \texttt{if (}E\texttt{)}\ S \qquad\qquad S \text{ ``can be written as'' } \texttt{if (}E\texttt{)}\ S$$

says that, if "$E$" is an *expression* and "$S$" is a *statement*, then

$$\texttt{if (}E\texttt{)}\ S$$

is also a *statement*

There are abbreviations possible in the notation; for example, we can write

$$\begin{aligned} S ::= & \texttt{if (}E\texttt{)}\ S \\ & \mid \texttt{if (}E\texttt{)}\ S\ \texttt{else}\ S \end{aligned}$$

as shorthand for

$$\begin{aligned} S ::= & \texttt{if (}E\texttt{)}\ S \\ S ::= & \texttt{if (}E\texttt{)}\ S\ \texttt{else}\ S \end{aligned}$$

## Context-free Grammars

$[\cdot]$ indicate that a phrase is optional; e.g., the two rules from above can be written as

$$S ::= \texttt{if (}E\texttt{)}\ S\ [\texttt{else}\ S]$$

$\{\cdot\}$ denote the Kleene closure: the phrase may appear zero or more times. For example

$$E ::= T\ \{\texttt{+}\ T\}$$

says that an expression $E$ may be written as a term $T$, followed by zero or more occurrences of + followed by a term $T$, such as $T$ + $T$ + $T$ ...

One may use the alternation sign | inside right-hand-sides, using parentheses for grouping; for example

$$E ::= T\ \{(\texttt{+} \mid \texttt{-})\ T\}$$

means that the additive operator may be either + or -, such as $T$ + $T$ - $T$ + $T$

## Context-free Grammars

$[\cdot]$ indicate that a phrase is optional; e.g., the two rules from above can be written as

$$S ::= \texttt{if (}E\texttt{)}\ S\ [\texttt{else}\ S]$$

$\{\cdot\}$ denote the Kleene closure: the phrase may appear zero or more times. For example

$$E ::= T\ \{\texttt{+}\ T\}$$

says that an expression $E$ may be written as a term $T$, followed by zero or more occurrences of + followed by a term $T$, such as $T$ + $T$ + $T$ ...

One may use the alternation sign | inside right-hand-sides, using parentheses for grouping; for example

$$E ::= T\ \{(\texttt{+} \mid \texttt{-})\ T\}$$

means that the additive operator may be either + or -, such as $T$ + $T$ - $T$ + $T$

## Context-free Grammars

$[\cdot]$ indicate that a phrase is optional; e.g., the two rules from above can be written as

$$S ::= \texttt{if (}E\texttt{)}\ S\ [\texttt{else}\ S]$$

$\{\cdot\}$ denote the Kleene closure: the phrase may appear zero or more times. For example

$$E ::= T\ \{\texttt{+}\ T\}$$

says that an expression $E$ may be written as a term $T$, followed by zero or more occurrences of + followed by a term $T$, such as $T$ + $T$ + $T$ ...

One may use the alternation sign | inside right-hand-sides, using parentheses for grouping; for example

$$E ::= T\ \{(\texttt{+} \mid \texttt{-})\ T\}$$

means that the additive operator may be either + or -, such as $T$ + $T$ - $T$ + $T$

## Backus-Naur Form (BNF)

BNF allows us to describe such syntax as that for a *j*-- compilation unit

$$\begin{aligned}\text{compilationUnit} ::= {} & [\texttt{package}\ \text{qualifiedIdentifier}\ \texttt{;}] \\ & \{\texttt{import}\ \ \text{qualifiedIdentifier}\ \texttt{;}\} \\ & \{\text{typeDeclaration}\}\ \texttt{EOF}\end{aligned}$$

A context-free grammar is a tuple, $G = (NonTerm, Term, S, Prod)$ where

- $NonTerm$ is a set of non-terminal symbols, sometimes called non-terminals,
- $Term$ is a set of terminal symbols, sometimes called terminals,
- $S \in NonTerm$ is a designated non-terminal, called the start symbol, and
- $Prod$ is a set of production rules, sometimes called productions or rules

For example, a context-free grammar that describes simple arithmetic expressions is $G = (NonTerm, Term, S, Prod)$ where $NonTerm = \{E, T, F\}$ is the set of non-terminals, $Term = \{\texttt{+}, \texttt{*}, \texttt{(}, \texttt{)}, \texttt{id}\}$ is the set of terminals, $S = E$ is the start symbol, and

$$\begin{aligned}Prod = \{ & E ::= E \texttt{+} T, \\ & E ::= T, \\ & T ::= T \texttt{*} F, \\ & T ::= F, \\ & F ::= \texttt{(} E \texttt{)}, \\ & F ::= \texttt{id}\}\end{aligned}$$

## Backus-Naur Form (BNF)

BNF allows us to describe such syntax as that for a *j*-- compilation unit

$$\begin{aligned}\text{compilationUnit} ::= {} & [\texttt{package}\ \text{qualifiedIdentifier}\ \texttt{;}] \\ & \{\texttt{import}\ \ \text{qualifiedIdentifier}\ \texttt{;}\} \\ & \{\text{typeDeclaration}\}\ \texttt{EOF}\end{aligned}$$

A context-free grammar is a tuple, $G = (NonTerm, Term, S, Prod)$ where

- $NonTerm$ is a set of non-terminal symbols, sometimes called non-terminals,
- $Term$ is a set of terminal symbols, sometimes called terminals,
- $S \in NonTerm$ is a designated non-terminal, called the start symbol, and
- $Prod$ is a set of production rules, sometimes called productions or rules

For example, a context-free grammar that describes simple arithmetic expressions is $G = (NonTerm, Term, S, Prod)$ where $NonTerm = \{E, T, F\}$ is the set of non-terminals, $Term = \{\texttt{+}, \texttt{*}, \texttt{(}, \texttt{)}, \texttt{id}\}$ is the set of terminals, $S = E$ is the start symbol, and

$$\begin{aligned}Prod = \{ & E ::= E \texttt{+} T, \\ & E ::= T, \\ & T ::= T \texttt{*} F, \\ & T ::= F, \\ & F ::= \texttt{(} E \texttt{)}, \\ & F ::= \texttt{id}\}\end{aligned}$$

## Backus-Naur Form (BNF)

BNF allows us to describe such syntax as that for a *j*-- compilation unit

$$\begin{aligned}\text{compilationUnit} ::= {} & [\texttt{package}\ \text{qualifiedIdentifier}\ \texttt{;}] \\ & \{\texttt{import}\ \ \text{qualifiedIdentifier}\ \texttt{;}\} \\ & \{\text{typeDeclaration}\}\ \texttt{EOF}\end{aligned}$$

A context-free grammar is a tuple, $G = (NonTerm, Term, S, Prod)$ where

- $NonTerm$ is a set of non-terminal symbols, sometimes called non-terminals,
- $Term$ is a set of terminal symbols, sometimes called terminals,
- $S \in NonTerm$ is a designated non-terminal, called the start symbol, and
- $Prod$ is a set of production rules, sometimes called productions or rules

For example, a context-free grammar that describes simple arithmetic expressions is $G = (NonTerm, Term, S, Prod)$ where $NonTerm = \{E, T, F\}$ is the set of non-terminals, $Term = \{\texttt{+}, \texttt{*}, \texttt{(}, \texttt{)}, \texttt{id}\}$ is the set of terminals, $S = E$ is the start symbol, and

$$\begin{aligned}Prod = \{ & E ::= E\ \texttt{+}\ T, \\ & E ::= T, \\ & T ::= T\ \texttt{*}\ F, \\ & T ::= F, \\ & F ::= \texttt{(}E\texttt{)}, \\ & F ::= \texttt{id}\}\end{aligned}$$

## Backus-Naur Form (BNF)

We can denote the above grammar less formally, simply as a sequence of production rules

$$
\begin{aligned}
E &::= E \texttt{ + } T \\
E &::= T \\
T &::= T \texttt{ * } \mathrm{F} \\
T &::= F \\
F &::= \texttt{(} E \texttt{)} \\
F &::= \texttt{id}
\end{aligned}
$$

- Non-terminals: symbols appearing also on LHS of rules
- Terminals: symbols appearing only on RHS of rules
- Start symbol: first non-terminal symbol defined

The start symbol is important because it is from this symbol, using the production rules, that we can generate strings in a language

For example, we can record a sequence of applications of the production rules, starting from $E$ (starting symbol in the above grammar) to the sentence `id + id * id` as follows

$$
\begin{aligned}
E &\Rightarrow E \texttt{ + } T \\
&\Rightarrow T \texttt{ + } T \\
&\Rightarrow F \texttt{ + } T \\
&\Rightarrow \mathrm{id} \texttt{ + } T \\
&\Rightarrow \mathrm{id} \texttt{ + } T \texttt{ * } F \\
&\Rightarrow \mathrm{id} \texttt{ + } F \texttt{ * } F \\
&\Rightarrow \texttt{id + id * } F \\
&\Rightarrow \texttt{id + id * id}
\end{aligned}
$$

## Backus-Naur Form (BNF)

We can denote the above grammar less formally, simply as a sequence of production rules

$$
\begin{aligned}
E &::= E \texttt{ + } T \\
E &::= T \\
T &::= T \texttt{ * } \mathrm{F} \\
T &::= F \\
F &::= \texttt{(} E \texttt{)} \\
F &::= \texttt{id}
\end{aligned}
$$

- Non-terminals: symbols appearing also on LHS of rules
- Terminals: symbols appearing only on RHS of rules
- Start symbol: first non-terminal symbol defined

The start symbol is important because it is from this symbol, using the production rules, that we can generate strings in a language

For example, we can record a sequence of applications of the production rules, starting from $E$ (starting symbol in the above grammar) to the sentence `id + id * id` as follows

$$
\begin{aligned}
E &\Rightarrow E \texttt{ + } T \\
&\Rightarrow T \texttt{ + } T \\
&\Rightarrow F \texttt{ + } T \\
&\Rightarrow \mathrm{id} \texttt{ + } T \\
&\Rightarrow \mathrm{id} \texttt{ + } T \texttt{ * } F \\
&\Rightarrow \mathrm{id} \texttt{ + } F \texttt{ * } F \\
&\Rightarrow \texttt{id + id * } F \\
&\Rightarrow \texttt{id + id * id}
\end{aligned}
$$

## Backus-Naur Form (BNF)

We can denote the above grammar less formally, simply as a sequence of production rules

$$
\begin{aligned}
E &::= E \texttt{ + } T \\
E &::= T \\
T &::= T \texttt{ * } \mathrm{F} \\
T &::= F \\
F &::= \texttt{(} E \texttt{)} \\
F &::= \texttt{id}
\end{aligned}
$$

- Non-terminals: symbols appearing also on LHS of rules
- Terminals: symbols appearing only on RHS of rules
- Start symbol: first non-terminal symbol defined

The start symbol is important because it is from this symbol, using the production rules, that we can generate strings in a language

For example, we can record a sequence of applications of the production rules, starting from $E$ (starting symbol in the above grammar) to the sentence `id + id * id` as follows

$$
\begin{aligned}
E &\Rightarrow E \texttt{ + } T \\
&\Rightarrow T \texttt{ + } T \\
&\Rightarrow F \texttt{ + } T \\
&\Rightarrow \mathrm{id} \texttt{ + } T \\
&\Rightarrow \mathrm{id} \texttt{ + } T \texttt{ * } F \\
&\Rightarrow \mathrm{id} \texttt{ + } F \texttt{ * } F \\
&\Rightarrow \texttt{id + id * } F \\
&\Rightarrow \texttt{id + id * id}
\end{aligned}
$$

## Backus-Naur Form (BNF)

We can denote the above grammar less formally, simply as a sequence of production rules

$$
\begin{aligned}
E &::= E \texttt{ + } T \\
E &::= T \\
T &::= T \texttt{ * } \mathrm{F} \\
T &::= F \\
F &::= \texttt{(} E \texttt{)} \\
F &::= \texttt{id}
\end{aligned}
$$

- Non-terminals: symbols appearing also on LHS of rules
- Terminals: symbols appearing only on RHS of rules
- Start symbol: first non-terminal symbol defined

The start symbol is important because it is from this symbol, using the production rules, that we can generate strings in a language

For example, we can record a sequence of applications of the production rules, starting from $E$ (starting symbol in the above grammar) to the sentence `id + id * id` as follows

$$
\begin{aligned}
E &\Rightarrow E \texttt{ + } T \\
&\Rightarrow T \texttt{ + } T \\
&\Rightarrow F \texttt{ + } T \\
&\Rightarrow \mathrm{id} \texttt{ + } T \\
&\Rightarrow \mathrm{id} \texttt{ + } T \texttt{ * } F \\
&\Rightarrow \mathrm{id} \texttt{ + } F \texttt{ * } F \\
&\Rightarrow \texttt{id + id * } F \\
&\Rightarrow \texttt{id + id * id}
\end{aligned}
$$

## Derivation

When one string can be re-written as another string, using zero or more production rules from the grammar, we say the first string derives ($\overset{*}{\Rightarrow}$) the second string.

For example

$E \overset{*}{\Rightarrow} E$ (in zero steps)
$E \overset{*}{\Rightarrow}$ `id + F * F`
$T$ `+` $T \overset{*}{\Rightarrow}$ `id + id * id`

A grammar $G$ describes a language $L(G)$. $L(G)$ contains all strings (sentences) made of only terminal symbols, that can be derived from the start symbol: $L(G) = \{w | S \overset{*}{\Rightarrow} w$ and $w \in T*\}$

For example, in the grammar above

$E \overset{*}{\Rightarrow}$ `id + id * id`
$E \overset{*}{\Rightarrow}$ `id`
$E \overset{*}{\Rightarrow}$ `(id + id) * id`

Hence, $L(G)$ includes

- `id + id * id`, `id`, `(id + id) * id`,
- and infinitely more finite sentences

## Derivation

When one string can be re-written as another string, using zero or more production rules from the grammar, we say the first string derives ($\overset{*}{\Rightarrow}$) the second string.

For example

$E \overset{*}{\Rightarrow} E$ (in zero steps)
$E \overset{*}{\Rightarrow}$ `id + F * F`
$T$ `+` $T \overset{*}{\Rightarrow}$ `id + id * id`

A grammar $G$ describes a language $L(G)$. $L(G)$ contains all strings (sentences) made of only terminal symbols, that can be derived from the start symbol: $L(G) = \{w | S \overset{*}{\Rightarrow} w$ and $w \in T*\}$

For example, in the grammar above

$E \overset{*}{\Rightarrow}$ `id + id * id`
$E \overset{*}{\Rightarrow}$ `id`
$E \overset{*}{\Rightarrow}$ `(id + id) * id`

Hence, $L(G)$ includes

- `id + id * id`, `id`, `(id + id) * id`,
- and infinitely more finite sentences

## Derivation

When one string can be re-written as another string, using zero or more production rules from the grammar, we say the first string derives ($\overset{*}{\Rightarrow}$) the second string.

For example

$E \overset{*}{\Rightarrow} E$ (in zero steps)
$E \overset{*}{\Rightarrow}$ `id + F * F`
$T$ `+` $T \overset{*}{\Rightarrow}$ `id + id * id`

A grammar $G$ describes a language $L(G)$. $L(G)$ contains all strings (sentences) made of only terminal symbols, that can be derived from the start symbol: $L(G) = \{w | S \overset{*}{\Rightarrow} w \text{ and } w \in T*\}$

For example, in the grammar above

$E \overset{*}{\Rightarrow}$ `id + id * id`
$E \overset{*}{\Rightarrow}$ `id`
$E \overset{*}{\Rightarrow}$ `(id + id) * id`

Hence, $L(G)$ includes

- `id + id * id`, `id`, `(id + id) * id`,
- and infinitely more finite sentences

## Derivation

When one string can be re-written as another string, using zero or more production rules from the grammar, we say the first string derives ($\overset{*}{\Rightarrow}$) the second string.

For example

$E \overset{*}{\Rightarrow} E$ (in zero steps)
$E \overset{*}{\Rightarrow}$ `id + F * F`
$T$ `+` $T \overset{*}{\Rightarrow}$ `id + id * id`

A grammar $G$ describes a language $L(G)$. $L(G)$ contains all strings (sentences) made of only terminal symbols, that can be derived from the start symbol: $L(G) = \{w | S \overset{*}{\Rightarrow} w \text{ and } w \in T*\}$

For example, in the grammar above

$E \overset{*}{\Rightarrow}$ `id + id * id`
$E \overset{*}{\Rightarrow}$ `id`
$E \overset{*}{\Rightarrow}$ `(id + id) * id`

Hence, $L(G)$ includes

- `id + id * id`, `id`, `(id + id) * id`,
- and infinitely more finite sentences

## Derivation

When one string can be re-written as another string, using zero or more production rules from the grammar, we say the first string derives ($\overset{*}{\Rightarrow}$) the second string.

For example

$E \overset{*}{\Rightarrow} E$ (in zero steps)
$E \overset{*}{\Rightarrow}$ `id + F * F`
$T$ `+` $T \overset{*}{\Rightarrow}$ `id + id * id`

A grammar $G$ describes a language $L(G)$. $L(G)$ contains all strings (sentences) made of only terminal symbols, that can be derived from the start symbol: $L(G) = \{w | S \overset{*}{\Rightarrow} w$ and $w \in T*\}$

For example, in the grammar above

$E \overset{*}{\Rightarrow}$ `id + id * id`
$E \overset{*}{\Rightarrow}$ `id`
$E \overset{*}{\Rightarrow}$ `(id + id) * id`

Hence, $L(G)$ includes

- `id + id * id,` `id,` `(id + id) * id,`
- and infinitely more finite sentences

## Left-most and right-most derivation

There are two kinds of derivation that will be important to us when we go about parsing these languages: left-most derivations and right-most derivations

A left-most derivation is a derivation in which at each step, the next string is derived by applying a production rule for rewriting the left-most non-terminal

For example

$$
\begin{aligned}
\underline{E} &\Rightarrow \underline{E} + T \\
&\Rightarrow \underline{T} + T \\
&\Rightarrow \underline{F} + T \\
&\Rightarrow \mathrm{id} + \underline{T} \\
&\Rightarrow \mathrm{id} + \underline{T} * F \\
&\Rightarrow \mathrm{id} + \underline{F} * F \\
&\Rightarrow \texttt{id} + \texttt{id} * \underline{F} \\
&\Rightarrow \texttt{id} + \texttt{id} * \texttt{id}
\end{aligned}
$$

## Left-most and right-most derivation

There are two kinds of derivation that will be important to us when we go about parsing these languages: left-most derivations and right-most derivations

A left-most derivation is a derivation in which at each step, the next string is derived by applying a production rule for rewriting the left-most non-terminal

For example

$$
\begin{aligned}
\underline{E} &\Rightarrow \underline{E} + T \\
&\Rightarrow \underline{T} + T \\
&\Rightarrow \underline{F} + T \\
&\Rightarrow \mathrm{id} + \underline{T} \\
&\Rightarrow \mathrm{id} + \underline{T} * F \\
&\Rightarrow \mathrm{id} + \underline{F} * F \\
&\Rightarrow \texttt{id} + \texttt{id} * \underline{F} \\
&\Rightarrow \texttt{id} + \texttt{id} * \texttt{id}
\end{aligned}
$$

## Left-most and right-most derivation

There are two kinds of derivation that will be important to us when we go about parsing these languages: left-most derivations and right-most derivations

A left-most derivation is a derivation in which at each step, the next string is derived by applying a production rule for rewriting the left-most non-terminal

For example

$$
\begin{aligned}
\underline{E} &\Rightarrow \underline{E} + T \\
&\Rightarrow \underline{T} + T \\
&\Rightarrow \underline{F} + T \\
&\Rightarrow \mathrm{id} + \underline{T} \\
&\Rightarrow \mathrm{id} + \underline{T} * F \\
&\Rightarrow \mathrm{id} + \underline{F} * F \\
&\Rightarrow \texttt{id} + \texttt{id} * \underline{F} \\
&\Rightarrow \texttt{id} + \texttt{id} * \texttt{id}
\end{aligned}
$$

## Left-most and right-most derivation

A right-most derivation is a derivation in which at each step, the next string is derived by applying a production rule for rewriting the right-most non-terminal

For example, the right-most derivation of `id + id * id` would go as follows

$$
\begin{aligned}
\underline{E} &\Rightarrow E + \underline{T} \\
&\Rightarrow E + T * \underline{F} \\
&\Rightarrow E + \underline{T} * \texttt{id} \\
&\Rightarrow E + \underline{F} * \texttt{id} \\
&\Rightarrow \underline{E} + \texttt{id} * \texttt{id} \\
&\Rightarrow \underline{T} + \texttt{id} * \texttt{id} \\
&\Rightarrow \underline{F} + \texttt{id} * \texttt{id} \\
&\Rightarrow \texttt{id} + \texttt{id} * \texttt{id}
\end{aligned}
$$

We use the term sentential form to refer to any string of (terminal and non-terminal) symbols that can be derived from the start symbol
In the previous derivation we have the following sententials:

$E$
$E + T$
$E + T * F$
$E + T * \texttt{id}$
$E + F * \texttt{id}$
$E + \texttt{id} * \texttt{id}$
$T + \texttt{id} * \texttt{id}$
$F + \texttt{id} * \texttt{id}$
$\texttt{id} + \texttt{id} * \texttt{id}$

## Left-most and right-most derivation

A right-most derivation is a derivation in which at each step, the next string is derived by applying a production rule for rewriting the right-most non-terminal

For example, the right-most derivation of `id + id * id` would go as follows

$$
\begin{aligned}
\underline{E} &\Rightarrow E \texttt{ + } \underline{T} \\
&\Rightarrow E \texttt{ + } T \texttt{ * } \underline{F} \\
&\Rightarrow E \texttt{ + } \underline{T} \texttt{ * id} \\
&\Rightarrow E \texttt{ + } \underline{F} \texttt{ * id} \\
&\Rightarrow \underline{E} \texttt{ + id * id} \\
&\Rightarrow \underline{T} \texttt{ + id * id} \\
&\Rightarrow \underline{F} \texttt{ + id * id} \\
&\Rightarrow \texttt{id + id * id}
\end{aligned}
$$

We use the term sentential form to refer to any string of (terminal and non-terminal) symbols that can be derived from the start symbol

In the previous derivation we have the following sententials:

$E$

$E \texttt{ + } T$

$E \texttt{ + } T \texttt{ * } F$

$E \texttt{ + } T \texttt{ * id}$

$E \texttt{ + } F \texttt{ * id}$

$E \texttt{ + id * id}$

$T \texttt{ + id * id}$

$F \texttt{ + id * id}$

$\texttt{id + id * id}$

## Left-most and right-most derivation

A right-most derivation is a derivation in which at each step, the next string is derived by applying a production rule for rewriting the right-most non-terminal

For example, the right-most derivation of `id + id * id` would go as follows

$$
\begin{aligned}
\underline{E} &\Rightarrow E \texttt{ + } \underline{T} \\
&\Rightarrow E \texttt{ + } T \texttt{ * } \underline{F} \\
&\Rightarrow E \texttt{ + } \underline{T} \texttt{ * id} \\
&\Rightarrow E \texttt{ + } \underline{F} \texttt{ * id} \\
&\Rightarrow \underline{E} \texttt{ + id * id} \\
&\Rightarrow \underline{T} \texttt{ + id * id} \\
&\Rightarrow \underline{F} \texttt{ + id * id} \\
&\Rightarrow \texttt{id + id * id}
\end{aligned}
$$

We use the term sentential form to refer to any string of (terminal and non-terminal) symbols that can be derived from the start symbol

In the previous derivation we have the following sententials:

$E$

$E \texttt{ + } T$

$E \texttt{ + } T \texttt{ * } F$

$E \texttt{ + } T \texttt{ * id}$

$E \texttt{ + } F \texttt{ * id}$

$E \texttt{ + id * id}$

$T \texttt{ + id * id}$

$F \texttt{ + id * id}$

$\texttt{id + id * id}$

## Left-most and right-most derivation

A right-most derivation is a derivation in which at each step, the next string is derived by applying a production rule for rewriting the right-most non-terminal

For example, the right-most derivation of `id + id * id` would go as follows

$$
\begin{aligned}
\underline{E} &\Rightarrow E \texttt{ + } \underline{T} \\
&\Rightarrow E \texttt{ + } T \texttt{ * } \underline{F} \\
&\Rightarrow E \texttt{ + } \underline{T} \texttt{ * id} \\
&\Rightarrow E \texttt{ + } \underline{F} \texttt{ * id} \\
&\Rightarrow \underline{E} \texttt{ + id * id} \\
&\Rightarrow \underline{T} \texttt{ + id * id} \\
&\Rightarrow \underline{F} \texttt{ + id * id} \\
&\Rightarrow \texttt{id + id * id}
\end{aligned}
$$

We use the term sentential form to refer to any string of (terminal and non-terminal) symbols that can be derived from the start symbol
In the previous derivation we have the following sententials:

$E$
$E \texttt{ + } T$
$E \texttt{ + } T \texttt{ * } F$
$E \texttt{ + } T \texttt{ * id}$
$E \texttt{ + } F \texttt{ * id}$
$E \texttt{ + id * id}$
$T \texttt{ + id * id}$
$F \texttt{ + id * id}$
$\texttt{id + id * id}$

## Parse Tree

A derivation can be represented using a parse tree. It shows the derivation and the structure

- of an input string (at the leaves)
- starting from a start symbol (at the root)

$$
\begin{aligned}
\underline{E} &\Rightarrow E \texttt{ + } \underline{T} \\
&\Rightarrow E \texttt{ + } T \texttt{ * } \underline{F} \\
&\Rightarrow E \texttt{ + } \underline{T} \texttt{ * id} \\
&\Rightarrow E \texttt{ + } \underline{F} \texttt{ * id} \\
&\Rightarrow \underline{E} \texttt{ + id * id} \\
&\Rightarrow \underline{T} \texttt{ + id * id} \\
&\Rightarrow \underline{F} \texttt{ + id * id} \\
&\Rightarrow \texttt{id + id * id}
\end{aligned}
$$

The derivation (left) and the parse tree (right) of `id + id * id`

## Ambiguity

Given a grammar $G$, if there exists a sentence $s$ in $L(G)$ for which:

- there is more than one left-most derivation, or equivalently
- there is more than one right-most derivation, or equivalently
- there is more than one parse tree in $G$

we say that the sentence $s$ is ambiguous

- Ambiguous grammar: a grammar that describes at least one ambiguous sentence
- Unambiguous grammar: a grammar that does not describe any ambiguous sentences

## Ambiguity

Given a grammar $G$, if there exists a sentence $s$ in $L(G)$ for which:

- there is more than one left-most derivation, or equivalently
- there is more than one right-most derivation, or equivalently
- there is more than one parse tree in $G$

we say that the sentence $s$ is ambiguous

- Ambiguous grammar: a grammar that describes at least one ambiguous sentence
- Unambiguous grammar: a grammar that does not describe any ambiguous sentences

## Ambiguity: an example

### Using left-most derivations

Consider the grammar

$$E ::= E \texttt{ + } E \mid E \texttt{ * } E \mid \texttt{(} E \texttt{)} \mid \texttt{id}$$

Is this grammar ambiguous?

## Ambiguity: an example

### Using left-most derivations

Consider the grammar

$$E ::= E \texttt{ + } E \mid E \texttt{ * } E \mid \texttt{(} E \texttt{)} \mid \texttt{id}$$

Is this grammar ambiguous? **Hint**: consider the sentence `id + id * id`

## Ambiguity: an example

### Using left-most derivations

Consider the grammar

$$E ::= E \texttt{ + } E \mid E \texttt{ * } E \mid \texttt{(}E\texttt{)} \mid \texttt{id}$$

Is this grammar ambiguous? Hint: consider the sentence `id + id * id`

One left-most derivation for this sentence is

$$\begin{aligned}
\underline{E} &\Rightarrow \underline{E} \texttt{ + } E \\
&\Rightarrow \texttt{id + } \underline{E} \\
&\Rightarrow \texttt{id + } \underline{E} \texttt{ * } E \\
&\Rightarrow \texttt{id + id * } \underline{E} \\
&\Rightarrow \texttt{id + id * id}
\end{aligned}$$

Another left-most derivation of the same sentence is

$$\begin{aligned}
\underline{E} &\Rightarrow \underline{E} \texttt{ * } E \\
&\Rightarrow \underline{E} \texttt{ + } E \texttt{ * } E \\
&\Rightarrow \texttt{id + } \underline{E} \texttt{ * } E \\
&\Rightarrow \texttt{id + id * } \underline{E} \\
&\Rightarrow \texttt{id + id * id}
\end{aligned}$$

Therefore, the grammar is ambiguous!

## Ambiguity: an example

### Using left-most derivations

Consider the grammar

$$E ::= E \texttt{ + } E \mid E \texttt{ * } E \mid \texttt{(}E\texttt{)} \mid \texttt{id}$$

Is this grammar ambiguous? Hint: consider the sentence `id + id * id`

One left-most derivation for this sentence is

$$\begin{aligned}
\underline{E} &\Rightarrow \underline{E} \texttt{ + } E \\
&\Rightarrow \texttt{id + } \underline{E} \\
&\Rightarrow \texttt{id + } \underline{E} \texttt{ * } E \\
&\Rightarrow \texttt{id + id * } \underline{E} \\
&\Rightarrow \texttt{id + id * id}
\end{aligned}$$

Another left-most derivation of the same sentence is

$$\begin{aligned}
\underline{E} &\Rightarrow \underline{E} \texttt{ * } E \\
&\Rightarrow \underline{E} \texttt{ + } E \texttt{ * } E \\
&\Rightarrow \texttt{id + } \underline{E} \texttt{ * } E \\
&\Rightarrow \texttt{id + id * } \underline{E} \\
&\Rightarrow \texttt{id + id * id}
\end{aligned}$$

Therefore, the grammar is ambiguous!

## Ambiguity: an example

### Using left-most derivations

Consider the grammar

$$E ::= E \texttt{ + } E \mid E \texttt{ * } E \mid \texttt{(}E\texttt{)} \mid \texttt{id}$$

Is this grammar ambiguous? Hint: consider the sentence `id + id * id`

One left-most derivation for this sentence is

$$\begin{aligned}
\underline{E} &\Rightarrow \underline{E} \texttt{ + } E \\
&\Rightarrow \texttt{id + } \underline{E} \\
&\Rightarrow \texttt{id + } \underline{E} \texttt{ * } E \\
&\Rightarrow \texttt{id + id * } \underline{E} \\
&\Rightarrow \texttt{id + id * id}
\end{aligned}$$

Another left-most derivation of the same sentence is

$$\begin{aligned}
\underline{E} &\Rightarrow \underline{E} \texttt{ * } E \\
&\Rightarrow \underline{E} \texttt{ + } E \texttt{ * } E \\
&\Rightarrow \texttt{id + } \underline{E} \texttt{ * } E \\
&\Rightarrow \texttt{id + id * } \underline{E} \\
&\Rightarrow \texttt{id + id * id}
\end{aligned}$$

Therefore, the grammar is ambiguous!

## Ambiguity: an example

### The same problem arises using right-most derivations & parse trees

Consider again the grammar

$$E ::= E \texttt{ + } E \mid E \texttt{ * } E \mid \texttt{(} E \texttt{)} \mid \texttt{id}$$

The sentence `id + id * id` also has two right-most derivations in the grammar:

$$\begin{aligned} \underline{E} &\Rightarrow E \texttt{ + } \underline{E} \\ &\Rightarrow E \texttt{ + } E \texttt{ * } \underline{E} \\ &\Rightarrow E \texttt{ + } \underline{E} \texttt{ * id} \\ &\Rightarrow \underline{E} \texttt{ + id * id} \\ &\Rightarrow \texttt{id + id * id} \end{aligned}$$

$$\begin{aligned} \underline{E} &\Rightarrow E \texttt{ * } \underline{E} \\ &\Rightarrow \underline{E} \texttt{ * id} \\ &\Rightarrow E \texttt{ + } \underline{E} \texttt{ * id} \\ &\Rightarrow \underline{E} \texttt{ + id * id} \\ &\Rightarrow \texttt{id + id * id} \end{aligned}$$

with the following two parse trees

## Ambiguity: an example

### The same problem arises using right-most derivations & parse trees

Consider again the grammar

$$E ::= E \texttt{ + } E \mid E \texttt{ * } E \mid \texttt{(}E\texttt{)} \mid \texttt{id}$$

The sentence `id + id * id` also has two right-most derivations in the grammar:

$$\begin{aligned} \underline{E} &\Rightarrow E \texttt{ + } \underline{E} \\ &\Rightarrow E \texttt{ + } E \texttt{ * } \underline{E} \\ &\Rightarrow E \texttt{ + } \underline{E} \texttt{ * id} \\ &\Rightarrow \underline{E} \texttt{ + id * id} \\ &\Rightarrow \texttt{id + id * id} \end{aligned}$$

$$\begin{aligned} \underline{E} &\Rightarrow E \texttt{ * } \underline{E} \\ &\Rightarrow \underline{E} \texttt{ * id} \\ &\Rightarrow E \texttt{ + } \underline{E} \texttt{ * id} \\ &\Rightarrow \underline{E} \texttt{ + id * id} \\ &\Rightarrow \texttt{id + id * id} \end{aligned}$$

with the following two parse trees

## Ambiguity: an example

### The same problem arises using right-most derivations & parse trees

Consider again the grammar

$$E ::= E \texttt{ + } E \mid E \texttt{ * } E \mid \texttt{(}E\texttt{)} \mid \texttt{id}$$

The sentence `id + id * id` also has two right-most derivations in the grammar:

$$\begin{aligned} \underline{E} &\Rightarrow E \texttt{ + } \underline{E} \\ &\Rightarrow E \texttt{ + } E \texttt{ * } \underline{E} \\ &\Rightarrow E \texttt{ + } \underline{E} \texttt{ * id} \\ &\Rightarrow \underline{E} \texttt{ + id * id} \\ &\Rightarrow \texttt{id + id * id} \end{aligned}$$

$$\begin{aligned} \underline{E} &\Rightarrow E \texttt{ * } \underline{E} \\ &\Rightarrow \underline{E} \texttt{ * id} \\ &\Rightarrow E \texttt{ + } \underline{E} \texttt{ * id} \\ &\Rightarrow \underline{E} \texttt{ + id * id} \\ &\Rightarrow \texttt{id + id * id} \end{aligned}$$

with the following two parse trees

## Ambiguity: an example

### The same problem arises using right-most derivations & parse trees

Consider again the grammar

$$E ::= E \texttt{ + } E \mid E \texttt{ * } E \mid \texttt{(}E\texttt{)} \mid \texttt{id}$$

The sentence `id + id * id` also has two right-most derivations in the grammar:

$$\begin{aligned}
\underline{E} &\Rightarrow E \texttt{ + } \underline{E} \\
&\Rightarrow E \texttt{ + } E \texttt{ * } \underline{E} \\
&\Rightarrow E \texttt{ + } \underline{E} \texttt{ * id} \\
&\Rightarrow \underline{E} \texttt{ + id * id} \\
&\Rightarrow \texttt{id + id * id}
\end{aligned}$$

$$\begin{aligned}
\underline{E} &\Rightarrow E \texttt{ * } \underline{E} \\
&\Rightarrow \underline{E} \texttt{ * id} \\
&\Rightarrow E \texttt{ + } \underline{E} \texttt{ * id} \\
&\Rightarrow \underline{E} \texttt{ + id * id} \\
&\Rightarrow \texttt{id + id * id}
\end{aligned}$$

with the following two parse trees

## Is this ambiguous?

### if then VS if then else

As another example, consider the grammar describing conditional statements

$$
\begin{aligned}
S &::= \texttt{if (}E\texttt{)}\ S \\
&\quad \mid \texttt{if (}E\texttt{)}\ S\ \texttt{else}\ S \\
&\quad \mid \texttt{s} \\
E &::= \texttt{e}
\end{aligned}
$$

and consider the sentence `if (e) if (e) s else s`

## Ambiguity: a real example

### if then VS if then else

As another example, consider the grammar describing conditional statements

$$
\begin{aligned}
S &::= \texttt{if (}E\texttt{)}\ S \\
&\quad \mid \texttt{if (}E\texttt{)}\ S\ \texttt{else}\ S \\
&\quad \mid \texttt{s} \\
E &::= \texttt{e}
\end{aligned}
$$

and consider the sentence `if (e) if (e) s else s`

There exist two left-most derivations for the above sentence in the grammar

$$
\begin{aligned}
\underline{S} &\Rightarrow \texttt{if (}\underline{E}\texttt{)}\ S\ \texttt{else}\ S \\
&\Rightarrow \texttt{if (e)}\ \underline{S}\ \texttt{else}\ S \\
&\Rightarrow \texttt{if}\ (\texttt{e})\ \texttt{if (}\underline{E}\texttt{)}\ S\ \texttt{else}\ S \\
&\Rightarrow \texttt{if}\ (\texttt{e})\ \texttt{if (e)}\ \underline{S}\ \texttt{else}\ S \\
&\Rightarrow \texttt{if (e) if (e) s else}\ \underline{S} \\
&\Rightarrow \texttt{if (e) if (e) s else s}
\end{aligned}
$$

$$
\begin{aligned}
\underline{S} &\Rightarrow \texttt{if (}\underline{E}\texttt{)}\ S \\
&\Rightarrow \texttt{if (e)}\ \underline{S} \\
&\Rightarrow \texttt{if (e) if (}\underline{E}\texttt{)}\ S\ \texttt{else}\ S \\
&\Rightarrow \texttt{if (e) if (e)}\ \underline{S}\ \texttt{else}\ S \\
&\Rightarrow \texttt{if (e) if (e) s else}\ \underline{S} \\
&\Rightarrow \texttt{if (e) if (e) s else s}
\end{aligned}
$$

with the following parse trees

## Ambiguity: a real example

### if then VS if then else

As another example, consider the grammar describing conditional statements

$$
\begin{aligned}
S &::= \texttt{if (}E\texttt{)}\ S \\
&\quad\mid \texttt{if (}E\texttt{)}\ S\ \texttt{else}\ S \\
&\quad\mid \texttt{s} \\
E &::= \texttt{e}
\end{aligned}
$$

and consider the sentence `if (e) if (e) s else s`

There exist two left-most derivations for the above sentence in the grammar

$$
\begin{aligned}
\underline{S} &\Rightarrow \texttt{if (}\underline{E}\texttt{)}\ S\ \texttt{else}\ S \\
&\Rightarrow \texttt{if (e)}\ \underline{S}\ \texttt{else}\ S \\
&\Rightarrow \texttt{if}\ (\texttt{e})\ \texttt{if (}\underline{E}\texttt{)}\ S\ \texttt{else}\ S \\
&\Rightarrow \texttt{if}\ (\texttt{e})\ \texttt{if (e)}\ \underline{S}\ \texttt{else}\ S \\
&\Rightarrow \texttt{if (e) if (e) s else}\ \underline{S} \\
&\Rightarrow \texttt{if (e) if (e) s else s}
\end{aligned}
$$

$$
\begin{aligned}
\underline{S} &\Rightarrow \texttt{if (}\underline{E}\texttt{)}\ S \\
&\Rightarrow \texttt{if (e)}\ \underline{S} \\
&\Rightarrow \texttt{if (e) if (}\underline{E}\texttt{)}\ S\ \texttt{else}\ S \\
&\Rightarrow \texttt{if (e) if (e)}\ \underline{S}\ \texttt{else}\ S \\
&\Rightarrow \texttt{if (e) if (e) s else}\ \underline{S} \\
&\Rightarrow \texttt{if (e) if (e) s else s}
\end{aligned}
$$

with the following parse trees

## Ambiguity: a real example

### Whose is this else?

One could easily modify the syntax of conditionals to remove the ambiguity

$$
\begin{aligned}
S &::= \texttt{if (}E\texttt{)}\ S \\
&\mid \texttt{if (}E\texttt{)}\ S\ \texttt{else}\ S \\
&\mid \texttt{s} \\
E &::= \texttt{e}
\end{aligned}
$$

$$
\begin{aligned}
S &::= \texttt{if}\ E\ \texttt{do}\ S \\
&\mid \texttt{if}\ E\ \texttt{then}\ S\ \texttt{else}\ S \\
&\mid \texttt{s} \\
E &::= \texttt{e}
\end{aligned}
$$

But programmers have become both accustomed to (and fond of) the ambiguous conditional

Compiler writers handle the rule as a special case in the compiler's parser, making sure that an `else` is grouped along with the closest preceding `if`

## Ambiguity: a real example

### Whose is this else?

One could easily modify the syntax of conditionals to remove the ambiguity

$$
\begin{aligned}
S &::= \texttt{if (}E\texttt{)}\ S \\
&\mid \texttt{if (}E\texttt{)}\ S\ \texttt{else}\ S \\
&\mid \texttt{s} \\
E &::= \texttt{e}
\end{aligned}
$$

$$
\begin{aligned}
S &::= \texttt{if}\ E\ \texttt{do}\ S \\
&\mid \texttt{if}\ E\ \texttt{then}\ S\ \texttt{else}\ S \\
&\mid \texttt{s} \\
E &::= \texttt{e}
\end{aligned}
$$

But programmers have become both accustomed to (and fond of) the ambiguous conditional

Compiler writers handle the rule as a special case in the compiler's parser, making sure that an `else` is grouped along with the closest preceding `if`

## Ambiguity: a real example

### Whose is this else?

One could easily modify the syntax of conditionals to remove the ambiguity

$$
\begin{aligned}
S &::= \texttt{if (}E\texttt{)}\ S \\
&\mid \texttt{if (}E\texttt{)}\ S\ \texttt{else}\ S \\
&\mid \texttt{s} \\
E &::= \texttt{e}
\end{aligned}
\qquad\qquad
\begin{aligned}
S &::= \texttt{if}\ E\ \texttt{do}\ S \\
&\mid \texttt{if}\ E\ \texttt{then}\ S\ \texttt{else}\ S \\
&\mid \texttt{s} \\
E &::= \texttt{e}
\end{aligned}
$$

But programmers have become both accustomed to (and fond of) the ambiguous conditional

Compiler writers handle the rule as a special case in the compiler's parser, making sure that an `else` is grouped along with the closest preceding `if`

## Ambiguity: a real example

### Whose is this else?

One could easily modify the syntax of conditionals to remove the ambiguity

$$
\begin{aligned}
S &::= \texttt{if (}E\texttt{)}\ S \\
&\mid \texttt{if (}E\texttt{)}\ S\ \texttt{else}\ S \\
&\mid \texttt{s} \\
E &::= \texttt{e}
\end{aligned}
\qquad\qquad
\begin{aligned}
S &::= \texttt{if}\ E\ \texttt{do}\ S \\
&\mid \texttt{if}\ E\ \texttt{then}\ S\ \texttt{else}\ S \\
&\mid \texttt{s} \\
E &::= \texttt{e}
\end{aligned}
$$

But programmers have become both accustomed to (and fond of) the ambiguous conditional

Compiler writers handle the rule as a special case in the compiler's parser, making sure that an `else` is grouped along with the closest preceding `if`

## Ambiguity: another real example

### x.y.z.w

The *j*-- (and Java) grammar have another ambiguity, which is even more difficult; consider the problem of parsing the expression `x.y.z.w`

- Clearly `w` is a field; if it were a method, then we would had `x.y.z.w()`

So far so good. But what about `x.y.z`? It depends on what are `x`, `y` and `z`.

- If `x` is a local variable, it might refer to an object with a field `y`, referring to another object with a field `z`, referring to yet another object having our field `w`; in this case, the expression would be parsed as a cascade of field selection expressions
- If `x.y` is a package, and `z` a class therein defined, then we would parse `x.y.z` as a fully qualified class, and `w` as a static field selection of class `z`
- Other possibilities, parsing various permutations of (possibly qualified) class names and field selection operations, also exist

The parser cannot determine how the expression `x.y.z` is parsed because types are not decided until after it has parsed the program and constructed its AST.

Hence `x.y.z` is represented in the AST by an `AmbiguousName` node; later on, after type declarations have been processed, the compiler re-writes this node as the appropriate sub-tree

## Ambiguity: another real example

### x.y.z.w

The *j*-- (and Java) grammar have another ambiguity, which is even more difficult; consider the problem of parsing the expression `x.y.z.w`

- Clearly `w` is a field; if it were a method, then we would had `x.y.z.w()`

So far so good. But what about `x.y.z`? It depends on what are `x`, `y` and `z`.

- If `x` is a local variable, it might refer to an object with a field `y`, referring to another object with a field `z`, referring to yet another object having our field `w`; in this case, the expression would be parsed as a cascade of field selection expressions
- If `x.y` is a package, and `z` a class therein defined, then we would parse `x.y.z` as a fully qualified class, and `w` as a static field selection of class `z`
- Other possibilities, parsing various permutations of (possibly qualified) class names and field selection operations, also exist

The parser cannot determine how the expression `x.y.z` is parsed because types are not decided until after it has parsed the program and constructed its AST.

Hence `x.y.z` is represented in the AST by an `AmbiguousName` node; later on, after type declarations have been processed, the compiler re-writes this node as the appropriate sub-tree

## Ambiguity: another real example

### x.y.z.w

The *j*-- (and Java) grammar have another ambiguity, which is even more difficult; consider the problem of parsing the expression `x.y.z.w`

- Clearly `w` is a field; if it were a method, then we would had `x.y.z.w()`

So far so good. But what about `x.y.z`? It depends on what are `x`, `y` and `z`.

- If `x` is a local variable, it might refer to an object with a field `y`, referring to another object with a field `z`, referring to yet another object having our field `w`; in this case, the expression would be parsed as a cascade of field selection expressions
- If `x.y` is a package, and `z` a class therein defined, then we would parse `x.y.z` as a fully qualified class, and `w` as a static field selection of class `z`
- Other possibilities, parsing various permutations of (possibly qualified) class names and field selection operations, also exist

The parser cannot determine how the expression `x.y.z` is parsed because types are not decided until after it has parsed the program and constructed its AST.

Hence `x.y.z` is represented in the AST by an `AmbiguousName` node; later on, after type declarations have been processed, the compiler re-writes this node as the appropriate sub-tree

## Ambiguity: another real example

### x.y.z.w

The *j*-- (and Java) grammar have another ambiguity, which is even more difficult; consider the problem of parsing the expression `x.y.z.w`

- Clearly `w` is a field; if it were a method, then we would had `x.y.z.w()`

So far so good. But what about `x.y.z`? It depends on what are `x`, `y` and `z`.

- If `x` is a local variable, it might refer to an object with a field `y`, referring to another object with a field `z`, referring to yet another object having our field `w`; in this case, the expression would be parsed as a cascade of field selection expressions
- If `x.y` is a package, and `z` a class therein defined, then we would parse `x.y.z` as a fully qualified class, and `w` as a static field selection of class `z`
- Other possibilities, parsing various permutations of (possibly qualified) class names and field selection operations, also exist

The parser cannot determine how the expression `x.y.z` is parsed because types are not decided until after it has parsed the program and constructed its AST.

Hence `x.y.z` is represented in the AST by an `AmbiguousName` node; later on, after type declarations have been processed, the compiler re-writes this node as the appropriate sub-tree

## Ambiguity: another real example

### x.y.z.w

The *j*-- (and Java) grammar have another ambiguity, which is even more difficult; consider the problem of parsing the expression `x.y.z.w`

- Clearly `w` is a field; if it were a method, then we would had `x.y.z.w()`

So far so good. But what about `x.y.z`? It depends on what are `x`, `y` and `z`.

- If `x` is a local variable, it might refer to an object with a field `y`, referring to another object with a field `z`, referring to yet another object having our field `w`; in this case, the expression would be parsed as a cascade of field selection expressions
- If `x.y` is a package, and `z` a class therein defined, then we would parse `x.y.z` as a fully qualified class, and `w` as a static field selection of class `z`
- Other possibilities, parsing various permutations of (possibly qualified) class names and field selection operations, also exist

The parser cannot determine how the expression `x.y.z` is parsed because types are not decided until after it has parsed the program and constructed its AST.

Hence `x.y.z` is represented in the AST by an `AmbiguousName` node; later on, after type declarations have been processed, the compiler re-writes this node as the appropriate sub-tree

## Ambiguity: another real example

### x.y.z.w

The *j*-- (and Java) grammar have another ambiguity, which is even more difficult; consider the problem of parsing the expression `x.y.z.w`

- Clearly `w` is a field; if it were a method, then we would had `x.y.z.w()`

So far so good. But what about `x.y.z`? It depends on what are `x`, `y` and `z`.

- If `x` is a local variable, it might refer to an object with a field `y`, referring to another object with a field `z`, referring to yet another object having our field `w`; in this case, the expression would be parsed as a cascade of field selection expressions
- If `x.y` is a package, and `z` a class therein defined, then we would parse `x.y.z` as a fully qualified class, and `w` as a static field selection of class `z`
- Other possibilities, parsing various permutations of (possibly qualified) class names and field selection operations, also exist

The parser cannot determine how the expression `x.y.z` is parsed because types are not decided until after it has parsed the program and constructed its AST.

Hence `x.y.z` is represented in the AST by an `AmbiguousName` node; later on, after type declarations have been processed, the compiler re-writes this node as the appropriate sub-tree

## Ambiguity: another real example

### x.y.z.w

The *j*-- (and Java) grammar have another ambiguity, which is even more difficult; consider the problem of parsing the expression `x.y.z.w`

- Clearly `w` is a field; if it were a method, then we would had `x.y.z.w()`

So far so good. But what about `x.y.z`? It depends on what are `x`, `y` and `z`.

- If `x` is a local variable, it might refer to an object with a field `y`, referring to another object with a field `z`, referring to yet another object having our field `w`; in this case, the expression would be parsed as a cascade of field selection expressions
- If `x.y` is a package, and `z` a class therein defined, then we would parse `x.y.z` as a fully qualified class, and `w` as a static field selection of class `z`
- Other possibilities, parsing various permutations of (possibly qualified) class names and field selection operations, also exist

The parser cannot determine how the expression `x.y.z` is parsed because types are not decided until after it has parsed the program and constructed its AST.

Hence `x.y.z` is represented in the AST by an `AmbiguousName` node; later on, after type declarations have been processed, the compiler re-writes this node as the appropriate sub-tree

## Ambiguity: another real example

### x.y.z.w

The *j*-- (and Java) grammar have another ambiguity, which is even more difficult; consider the problem of parsing the expression `x.y.z.w`

- Clearly `w` is a field; if it were a method, then we would had `x.y.z.w()`

So far so good. But what about `x.y.z`? It depends on what are `x`, `y` and `z`.

- If `x` is a local variable, it might refer to an object with a field `y`, referring to another object with a field `z`, referring to yet another object having our field `w`; in this case, the expression would be parsed as a cascade of field selection expressions
- If `x.y` is a package, and `z` a class therein defined, then we would parse `x.y.z` as a fully qualified class, and `w` as a static field selection of class `z`
- Other possibilities, parsing various permutations of (possibly qualified) class names and field selection operations, also exist

The parser cannot determine how the expression `x.y.z` is parsed because types are not decided until after it has parsed the program and constructed its AST.

Hence `x.y.z` is represented in the AST by an `AmbiguousName` node; later on, after type declarations have been processed, the compiler re-writes this node as the appropriate sub-tree

## Outline

1 Introduction

2 Context-free Grammars and Languages
- Backus-Naur Form (BNF)
- Left- and right-most derivation, & Parse tree
- Ambiguity

3 Top-down Deterministic Parsing
- Example: Top-down deterministic parsing of a *j*-- program
- Parsing by Recursive Descent
- Parsing by the LL(1) technique

4 Transform a non-LL(1) grammar into an equivalent LL(1) one

5 Parser Generation Using JavaCC

## What is top-down Deterministic Parsing?

### Example: Top-down deterministic parsing of a j-- program

A top-down deterministic parsing strategy scan the input from left to right, looking at and scanning just one symbol at a time.

The parser starts with the grammar's start symbol as an initial goal in the parsing process, which is then rewritten using a BNF rule replacing the symbol with the right-hand-side sequence of symbols

For example, in parsing a *j*-- program, our initial goal is to parse the start symbol, compilationUnit, which is defined by the following BNF rule

$$\begin{aligned}\text{compilationUnit} ::= {} & [\texttt{package}\ \text{qualifiedIdentifier}\ \texttt{;}] \\ & \{\texttt{import}\ \ \text{qualifiedIdentifier}\ \texttt{;}\} \\ & \{\text{typeDeclaration}\}\ \texttt{EOF}\end{aligned}$$

The goal of parsing a compilationUnit in the input can be rewritten as a number of sub-goals

1. If there is a `package` statement in the input sentence, then we parse that
2. If there are `import` statements in the input, then we parse them
3. If there are any type declarations, then we parse them
4. Finally, we parse the terminating `EOF` token

## What is top-down Deterministic Parsing?

### Example: Top-down deterministic parsing of a j-- program

A top-down deterministic parsing strategy scan the input from left to right, looking at and scanning just one symbol at a time.

The parser starts with the grammar's start symbol as an initial goal in the parsing process, which is then rewritten using a BNF rule replacing the symbol with the right-hand-side sequence of symbols

For example, in parsing a *j*-- program, our initial goal is to parse the start symbol, compilationUnit, which is defined by the following BNF rule

$$\begin{aligned}\text{compilationUnit} ::= {} & [\texttt{package}\ \text{qualifiedIdentifier}\ \texttt{;}] \\ & \{\texttt{import}\ \ \text{qualifiedIdentifier}\ \texttt{;}\} \\ & \{\text{typeDeclaration}\}\ \texttt{EOF}\end{aligned}$$

The goal of parsing a compilationUnit in the input can be rewritten as a number of sub-goals

1. If there is a `package` statement in the input sentence, then we parse that
2. If there are `import` statements in the input, then we parse them
3. If there are any type declarations, then we parse them
4. Finally, we parse the terminating `EOF` token

## What is top-down Deterministic Parsing?

### Example: Top-down deterministic parsing of a j-- program

A top-down deterministic parsing strategy scan the input from left to right, looking at and scanning just one symbol at a time.

The parser starts with the grammar's start symbol as an initial goal in the parsing process, which is then rewritten using a BNF rule replacing the symbol with the right-hand-side sequence of symbols

For example, in parsing a *j*-- program, our initial goal is to parse the start symbol, compilationUnit, which is defined by the following BNF rule

$$\begin{aligned}\text{compilationUnit} ::= & [\texttt{package}\ \text{qualifiedIdentifier}\ ;] \\ & \{\texttt{import}\ \ \text{qualifiedIdentifier}\ ;\} \\ & \{\text{typeDeclaration}\}\ \texttt{EOF}\end{aligned}$$

The goal of parsing a compilationUnit in the input can be rewritten as a number of sub-goals

1. If there is a `package` statement in the input sentence, then we parse that
2. If there are `import` statements in the input, then we parse them
3. If there are any type declarations, then we parse them
4. Finally, we parse the terminating `EOF` token

## What is top-down Deterministic Parsing?

### Example: Top-down deterministic parsing of a j-- program

A top-down deterministic parsing strategy scan the input from left to right, looking at and scanning just one symbol at a time.

The parser starts with the grammar's start symbol as an initial goal in the parsing process, which is then rewritten using a BNF rule replacing the symbol with the right-hand-side sequence of symbols

For example, in parsing a *j*-- program, our initial goal is to parse the start symbol, compilationUnit, which is defined by the following BNF rule

$$\begin{aligned}\text{compilationUnit} ::= & [\texttt{package}\ \text{qualifiedIdentifier}\ ;] \\ & \{\texttt{import}\ \ \text{qualifiedIdentifier}\ ;\} \\ & \{\text{typeDeclaration}\}\ \texttt{EOF}\end{aligned}$$

The goal of parsing a compilationUnit in the input can be rewritten as a number of sub-goals

1. If there is a `package` statement in the input sentence, then we parse that
2. If there are `import` statements in the input, then we parse them
3. If there are any type declarations, then we parse them
4. Finally, we parse the terminating `EOF` token

## What is top-down Deterministic Parsing?

### Example: Top-down deterministic parsing of a j-- program

Parsing a token, like `package`, is simple enough; if we see it, we simply scan it

Parsing a non-terminal is treated as another parsing (sub-) goal

For example, in the `package` statement, once we have scanned the `package` token, we are left with parsing a `qualifiedIdentifier`, which is defined by the following BNF rule

qualifiedIdentifier ::= <identifier> {. <identifier>}

We scan an <identifier> (treated by the lexical scanner as a token), and, so long as we see another period in the input, we scan that period and scan another <identifier>

This technique is known as top-down parsing because:

- starts at the start symbol
- continually rewrites non-terminals using BNF rules until reaching the leaves (tokens)

Since at each step in parsing a non-terminal we replace a parsing goal with a sequence of sub-goals, we often call this a goal-oriented parsing technique

## What is top-down Deterministic Parsing?

### Example: Top-down deterministic parsing of a j-- program

Parsing a token, like `package`, is simple enough; if we see it, we simply scan it

Parsing a non-terminal is treated as another parsing (sub-) goal

For example, in the `package` statement, once we have scanned the `package` token, we are left with parsing a `qualifiedIdentifier`, which is defined by the following BNF rule

qualifiedIdentifier ::= <identifier> {. <identifier>}

We scan an <identifier> (treated by the lexical scanner as a token), and, so long as we see another period in the input, we scan that period and scan another <identifier>

This technique is known as top-down parsing because:

- starts at the start symbol
- continually rewrites non-terminals using BNF rules until reaching the leaves (tokens)

Since at each step in parsing a non-terminal we replace a parsing goal with a sequence of sub-goals, we often call this a goal-oriented parsing technique

## What is top-down Deterministic Parsing?

### Example: Top-down deterministic parsing of a j-- program

Parsing a token, like `package`, is simple enough; if we see it, we simply scan it

Parsing a non-terminal is treated as another parsing (sub-) goal

For example, in the `package` statement, once we have scanned the `package` token, we are left with parsing a `qualifiedIdentifier`, which is defined by the following BNF rule

qualifiedIdentifier ::= `<identifier>` {. `<identifier>`}

We scan an `<identifier>` (treated by the lexical scanner as a token), and, so long as we see another period in the input, we scan that period and scan another `<identifier>`

This technique is known as top-down parsing because:

- starts at the start symbol
- continually rewrites non-terminals using BNF rules until reaching the leaves (tokens)

Since at each step in parsing a non-terminal we replace a parsing goal with a sequence of sub-goals, we often call this a goal-oriented parsing technique

## What is top-down Deterministic Parsing?

### Example: Top-down deterministic parsing of a j-- program

Parsing a token, like `package`, is simple enough; if we see it, we simply scan it

Parsing a non-terminal is treated as another parsing (sub-) goal

For example, in the `package` statement, once we have scanned the `package` token, we are left with parsing a `qualifiedIdentifier`, which is defined by the following BNF rule

$$\text{qualifiedIdentifier} ::= \texttt{<identifier>}\ \{\texttt{.}\ \texttt{<identifier>}\}$$

We scan an `<identifier>` (treated by the lexical scanner as a token), and, so long as we see another period in the input, we scan that period and scan another `<identifier>`

This technique is known as top-down parsing because:

- starts at the start symbol
- continually rewrites non-terminals using BNF rules until reaching the leaves (tokens)

Since at each step in parsing a non-terminal we replace a parsing goal with a sequence of sub-goals, we often call this a goal-oriented parsing technique

## What is top-down Deterministic Parsing?

### Example: Top-down deterministic parsing of a j-- program

Parsing a token, like `package`, is simple enough; if we see it, we simply scan it

Parsing a non-terminal is treated as another parsing (sub-) goal

For example, in the `package` statement, once we have scanned the `package` token, we are left with parsing a `qualifiedIdentifier`, which is defined by the following BNF rule

qualifiedIdentifier ::= `<identifier>` {. `<identifier>`}

We scan an `<identifier>` (treated by the lexical scanner as a token), and, so long as we see another period in the input, we scan that period and scan another `<identifier>`

This technique is known as top-down parsing because:

- starts at the start symbol
- continually rewrites non-terminals using BNF rules until reaching the leaves (tokens)

Since at each step in parsing a non-terminal we replace a parsing goal with a sequence of sub-goals, we often call this a goal-oriented parsing technique

## What is top-down Deterministic Parsing?

### Example: Top-down deterministic parsing of a j-- program

Parsing a token, like `package`, is simple enough; if we see it, we simply scan it

Parsing a non-terminal is treated as another parsing (sub-) goal

For example, in the `package` statement, once we have scanned the `package` token, we are left with parsing a `qualifiedIdentifier`, which is defined by the following BNF rule

$$\text{qualifiedIdentifier} ::= \texttt{<identifier>}\ \{\texttt{.}\ \texttt{<identifier>}\}$$

We scan an `<identifier>` (treated by the lexical scanner as a token), and, so long as we see another period in the input, we scan that period and scan another `<identifier>`

This technique is known as top-down parsing because:

- starts at the start symbol
- continually rewrites non-terminals using BNF rules until reaching the leaves (tokens)

Since at each step in parsing a non-terminal we replace a parsing goal with a sequence of sub-goals, we often call this a goal-oriented parsing technique

## How do we decide what next step to take?

For example, how do we decide whether or not there are more `import` statements to parse?

We decide by looking at the next un-scanned input token; if it is an `import`, we have another `import` statement to parse; otherwise we go on to parsing type declarations

As another example, consider the definition for the statement non-terminal

```
statement ::= block
            | if parExpression statement [else statement]
            | while parExpression statement
            | return [expression] ;
            | ;
            | statementExpression ;
```

We have six alternatives to choose from, depending on the next un-scanned input token

1. if the next token is a `{`, we parse a block
2. if the next token is an `if`, we parse an `if` statement
3. if the next token is a `while`, we parse a `while` statement
4. if the next token is a `return`, we parse a `return` statement
5. if the next token is a semicolon, we parse an empty statement
6. otherwise, we parse a statementExpression

Decisions are made looking at the next un-scanned token. Hence this is a LL(1) grammar

## How do we decide what next step to take?

For example, how do we decide whether or not there are more `import` statements to parse?

We decide by looking at the next un-scanned input token; if it is an `import`, we have another `import` statement to parse; otherwise we go on to parsing type declarations

As another example, consider the definition for the statement non-terminal

```
statement ::= block
            | if parExpression statement [else statement]
            | while parExpression statement
            | return [expression] ;
            | ;
            | statementExpression ;
```

We have six alternatives to choose from, depending on the next un-scanned input token

1. if the next token is a `{`, we parse a block
2. if the next token is an `if`, we parse an `if` statement
3. if the next token is a `while`, we parse a `while` statement
4. if the next token is a `return`, we parse a `return` statement
5. if the next token is a semicolon, we parse an empty statement
6. otherwise, we parse a statementExpression

Decisions are made looking at the next un-scanned token. Hence this is a LL(1) grammar

## How do we decide what next step to take?

For example, how do we decide whether or not there are more `import` statements to parse?

We decide by looking at the next un-scanned input token; if it is an `import`, we have another `import` statement to parse; otherwise we go on to parsing type declarations

As another example, consider the definition for the statement non-terminal

$$
\begin{aligned}
\text{statement} ::= \; & \text{block} \\
& \mid \texttt{if}\ \text{parExpression statement}\ [\texttt{else}\ \text{statement}] \\
& \mid \texttt{while}\ \text{parExpression statement} \\
& \mid \texttt{return}\ [\text{expression}]\ \texttt{;} \\
& \mid \texttt{;} \\
& \mid \text{statementExpression}\ \texttt{;}
\end{aligned}
$$

We have six alternatives to choose from, depending on the next un-scanned input token

1. if the next token is a `{`, we parse a block
2. if the next token is an `if`, we parse an `if` statement
3. if the next token is a `while`, we parse a `while` statement
4. if the next token is a `return`, we parse a `return` statement
5. if the next token is a semicolon, we parse an empty statement
6. otherwise, we parse a statementExpression

Decisions are made looking at the next un-scanned token. Hence this is a LL(1) grammar

## How do we decide what next step to take?

For example, how do we decide whether or not there are more `import` statements to parse?

We decide by looking at the next un-scanned input token; if it is an `import`, we have another `import` statement to parse; otherwise we go on to parsing type declarations

As another example, consider the definition for the statement non-terminal

$$
\begin{aligned}
\text{statement} ::= \; & \text{block} \\
& \mid \texttt{if}\ \text{parExpression statement}\ [\texttt{else}\ \text{statement}] \\
& \mid \texttt{while}\ \text{parExpression statement} \\
& \mid \texttt{return}\ [\text{expression}]\ \texttt{;} \\
& \mid \texttt{;} \\
& \mid \text{statementExpression}\ \texttt{;}
\end{aligned}
$$

We have six alternatives to choose from, depending on the next un-scanned input token

1. if the next token is a `{`, we parse a block
2. if the next token is an `if`, we parse an `if` statement
3. if the next token is a `while`, we parse a `while` statement
4. if the next token is a `return`, we parse a `return` statement
5. if the next token is a semicolon, we parse an empty statement
6. otherwise, we parse a statementExpression

Decisions are made looking at the next un-scanned token. Hence this is a LL(1) grammar

## How do we decide what next step to take?

For example, how do we decide whether or not there are more `import` statements to parse?

We decide by looking at the next un-scanned input token; if it is an `import`, we have another `import` statement to parse; otherwise we go on to parsing type declarations

As another example, consider the definition for the statement non-terminal

$$
\begin{aligned}
\text{statement} ::= \; & \text{block} \\
& \mid \texttt{if}\ \text{parExpression statement}\ [\texttt{else}\ \text{statement}] \\
& \mid \texttt{while}\ \text{parExpression statement} \\
& \mid \texttt{return}\ [\text{expression}]\ \texttt{;} \\
& \mid \texttt{;} \\
& \mid \text{statementExpression}\ \texttt{;}
\end{aligned}
$$

We have six alternatives to choose from, depending on the next un-scanned input token

1. if the next token is a `{`, we parse a block
2. if the next token is an `if`, we parse an `if` statement
3. if the next token is a `while`, we parse a `while` statement
4. if the next token is a `return`, we parse a `return` statement
5. if the next token is a semicolon, we parse an empty statement
6. otherwise, we parse a statementExpression

Decisions are made looking at the next un-scanned token. Hence this is a LL(1) grammar

## How do we decide what next step to take?

For example, how do we decide whether or not there are more `import` statements to parse?

We decide by looking at the next un-scanned input token; if it is an `import`, we have another `import` statement to parse; otherwise we go on to parsing type declarations

As another example, consider the definition for the statement non-terminal

$$
\begin{aligned}
\text{statement} ::= \; & \text{block} \\
& \mid \texttt{if}\ \text{parExpression statement}\ [\texttt{else}\ \text{statement}] \\
& \mid \texttt{while}\ \text{parExpression statement} \\
& \mid \texttt{return}\ [\text{expression}]\ \texttt{;} \\
& \mid \texttt{;} \\
& \mid \text{statementExpression}\ \texttt{;}
\end{aligned}
$$

We have six alternatives to choose from, depending on the next un-scanned input token

1. if the next token is a `{`, we parse a block
2. if the next token is an `if`, we parse an `if` statement
3. if the next token is a `while`, we parse a `while` statement
4. if the next token is a `return`, we parse a `return` statement
5. if the next token is a semicolon, we parse an empty statement
6. otherwise, we parse a statementExpression

Decisions are made looking at the next un-scanned token. Hence this is a LL(1) grammar

## How do we decide what next step to take?

For example, how do we decide whether or not there are more `import` statements to parse?

We decide by looking at the next un-scanned input token; if it is an `import`, we have another `import` statement to parse; otherwise we go on to parsing type declarations

As another example, consider the definition for the statement non-terminal

$$
\begin{aligned}
\text{statement} ::= \; & \text{block} \\
& \mid \texttt{if}\ \text{parExpression statement}\ [\texttt{else}\ \text{statement}] \\
& \mid \texttt{while}\ \text{parExpression statement} \\
& \mid \texttt{return}\ [\text{expression}]\ \texttt{;} \\
& \mid \texttt{;} \\
& \mid \text{statementExpression}\ \texttt{;}
\end{aligned}
$$

We have six alternatives to choose from, depending on the next un-scanned input token

1. if the next token is a `{`, we parse a block
2. if the next token is an `if`, we parse an `if` statement
3. if the next token is a `while`, we parse a `while` statement
4. if the next token is a `return`, we parse a `return` statement
5. if the next token is a semicolon, we parse an empty statement
6. otherwise, we parse a statementExpression

Decisions are made looking at the next un-scanned token. Hence this is a LL(1) grammar

## How do we decide what next step to take?

For example, how do we decide whether or not there are more `import` statements to parse?

We decide by looking at the next un-scanned input token; if it is an `import`, we have another `import` statement to parse; otherwise we go on to parsing type declarations

As another example, consider the definition for the statement non-terminal

$$
\begin{aligned}
\text{statement} ::= \; & \text{block} \\
& \mid \texttt{if}\ \text{parExpression statement}\ [\texttt{else}\ \text{statement}] \\
& \mid \texttt{while}\ \text{parExpression statement} \\
& \mid \texttt{return}\ [\text{expression}]\ \texttt{;} \\
& \mid \texttt{;} \\
& \mid \text{statementExpression}\ \texttt{;}
\end{aligned}
$$

We have six alternatives to choose from, depending on the next un-scanned input token

1. if the next token is a `{`, we parse a block
2. if the next token is an `if`, we parse an `if` statement
3. if the next token is a `while`, we parse a `while` statement
4. if the next token is a `return`, we parse a `return` statement
5. if the next token is a semicolon, we parse an empty statement
6. otherwise, we parse a statementExpression

Decisions are made looking at the next un-scanned token. Hence this is a LL(1) grammar

## How do we decide what next step to take?

For example, how do we decide whether or not there are more `import` statements to parse?

We decide by looking at the next un-scanned input token; if it is an `import`, we have another `import` statement to parse; otherwise we go on to parsing type declarations

As another example, consider the definition for the statement non-terminal

$$
\begin{aligned}
\text{statement} ::= \; & \text{block} \\
& \mid \texttt{if}\ \text{parExpression statement}\ [\texttt{else}\ \text{statement}] \\
& \mid \texttt{while}\ \text{parExpression statement} \\
& \mid \texttt{return}\ [\text{expression}]\ \texttt{;} \\
& \mid \texttt{;} \\
& \mid \text{statementExpression}\ \texttt{;}
\end{aligned}
$$

We have six alternatives to choose from, depending on the next un-scanned input token

1. if the next token is a `{`, we parse a block
2. if the next token is an `if`, we parse an `if` statement
3. if the next token is a `while`, we parse a `while` statement
4. if the next token is a `return`, we parse a `return` statement
5. if the next token is a semicolon, we parse an empty statement
6. otherwise, we parse a statementExpression

Decisions are made looking at the next un-scanned token. Hence this is a LL(1) grammar

## How do we decide what next step to take?

For example, how do we decide whether or not there are more `import` statements to parse?

We decide by looking at the next un-scanned input token; if it is an `import`, we have another `import` statement to parse; otherwise we go on to parsing type declarations

As another example, consider the definition for the statement non-terminal

$$
\begin{aligned}
\text{statement} ::= \; & \text{block} \\
& \mid \texttt{if}\ \text{parExpression statement}\ [\texttt{else}\ \text{statement}] \\
& \mid \texttt{while}\ \text{parExpression statement} \\
& \mid \texttt{return}\ [\text{expression}]\ \texttt{;} \\
& \mid \texttt{;} \\
& \mid \text{statementExpression}\ \texttt{;}
\end{aligned}
$$

We have six alternatives to choose from, depending on the next un-scanned input token

1. if the next token is a `{`, we parse a block
2. if the next token is an `if`, we parse an `if` statement
3. if the next token is a `while`, we parse a `while` statement
4. if the next token is a `return`, we parse a `return` statement
5. if the next token is a semicolon, we parse an empty statement
6. otherwise, we parse a statementExpression

Decisions are made looking at the next un-scanned token. Hence this is a LL(1) grammar

## How do we decide what next step to take?

For example, how do we decide whether or not there are more `import` statements to parse?

We decide by looking at the next un-scanned input token; if it is an `import`, we have another `import` statement to parse; otherwise we go on to parsing type declarations

As another example, consider the definition for the statement non-terminal

$$
\begin{aligned}
\text{statement} ::= \; & \text{block} \\
& \mid \texttt{if}\ \text{parExpression statement}\ [\texttt{else}\ \text{statement}] \\
& \mid \texttt{while}\ \text{parExpression statement} \\
& \mid \texttt{return}\ [\text{expression}]\ \texttt{;} \\
& \mid \texttt{;} \\
& \mid \text{statementExpression}\ \texttt{;}
\end{aligned}
$$

We have six alternatives to choose from, depending on the next un-scanned input token

1. if the next token is a `{`, we parse a block
2. if the next token is an `if`, we parse an `if` statement
3. if the next token is a `while`, we parse a `while` statement
4. if the next token is a `return`, we parse a `return` statement
5. if the next token is a semicolon, we parse an empty statement
6. otherwise, we parse a statementExpression

Decisions are made looking at the next un-scanned token. Hence this is a LL(1) grammar

## LL(1) , LL(2), . . . , LL($k$) grammars

In some cases, one must lookahead several tokens in the input to decide which alternative to take: LL(2), . . . , LL($k$).

In all cases, because we can predict which of several alternative right-hand-sides of a BNF rule to apply, based on the next input token(s), we say this is a predictive parsing technique

There are two principal top-down (goal-oriented, or predictive) parsing techniques:

1. Parsing by recursive descent
   - ⋆ This is the strategy we use in parsing *j*-- programs
2. LL(1) or LL($k$) parsing

## LL(1) , LL(2), . . . , LL($k$) grammars

In some cases, one must lookahead several tokens in the input to decide which alternative to take: LL(2), . . . , LL($k$).

In all cases, because we can predict which of several alternative right-hand-sides of a BNF rule to apply, based on the next input token(s), we say this is a predictive parsing technique

There are two principal top-down (goal-oriented, or predictive) parsing techniques:

1. Parsing by recursive descent
   - ⋆ This is the strategy we use in parsing *j*-- programs
2. LL(1) or LL($k$) parsing

## LL(1) , LL(2), . . . , LL($k$) grammars

In some cases, one must lookahead several tokens in the input to decide which alternative to take: LL(2), . . . , LL($k$).

In all cases, because we can predict which of several alternative right-hand-sides of a BNF rule to apply, based on the next input token(s), we say this is a predictive parsing technique

There are two principal top-down (goal-oriented, or predictive) parsing techniques:

1. Parsing by recursive descent
   - ⋆ This is the strategy we use in parsing *j*-- programs
2. LL(1) or LL($k$) parsing

## Outline

1 Introduction

2 Context-free Grammars and Languages
- Backus-Naur Form (BNF)
- Left- and right-most derivation, & Parse tree
- Ambiguity

3 Top-down Deterministic Parsing
- Example: Top-down deterministic parsing of a *j*-- program
- Parsing by Recursive Descent
- Parsing by the LL(1) technique

4 Transform a non-LL(1) grammar into an equivalent LL(1) one

5 Parser Generation Using JavaCC

## Recursive descent

Parsing by recursive descent involves writing a method (or procedure) for parsing each non-terminal according to the production rules that define that non-terminal

Depending on the next un-scanned input symbol, the method

1. decides which rule to apply and then
2. scans any terminals (tokens) on the right hand side by calling upon the `Scanner`, and then
3. parses any non-terminals by recursively invoking the methods that parse them

This is the strategy we use in parsing *j*-- programs

We already saw the method `compilationUnit()` that parses a *j*-- `compilationUnit`

- Slides 02_compilation.pdf

As another example, consider the rule defining `qualifiedIdentifier`

qualifiedIdentifier ::= `<identifier>` {. `<identifier>`}

## Recursive descent

Parsing by recursive descent involves writing a method (or procedure) for parsing each non-terminal according to the production rules that define that non-terminal

Depending on the next un-scanned input symbol, the method

1. decides which rule to apply and then
2. scans any terminals (tokens) on the right hand side by calling upon the `Scanner`, and then
3. parses any non-terminals by recursively invoking the methods that parse them

This is the strategy we use in parsing *j*-- programs

We already saw the method `compilationUnit()` that parses a *j*-- `compilationUnit`

- Slides 02_compilation.pdf

As another example, consider the rule defining `qualifiedIdentifier`

qualifiedIdentifier ::= `<identifier>` {. `<identifier>`}

## Recursive descent

Parsing by recursive descent involves writing a method (or procedure) for parsing each non-terminal according to the production rules that define that non-terminal

Depending on the next un-scanned input symbol, the method

1. decides which rule to apply and then
2. scans any terminals (tokens) on the right hand side by calling upon the `Scanner`, and then
3. parses any non-terminals by recursively invoking the methods that parse them

This is the strategy we use in parsing *j*-- programs

We already saw the method `compilationUnit()` that parses a *j*-- `compilationUnit`

- Slides 02_compilation.pdf

As another example, consider the rule defining `qualifiedIdentifier`

qualifiedIdentifier ::= `<identifier>` {. `<identifier>`}

## Recursive descent

### Example: parsing a qualifiedIdentifier

Parsing by recursive descent involves writing a method (or procedure) for parsing each non-terminal according to the production rules that define that non-terminal

Depending on the next un-scanned input symbol, the method

1. decides which rule to apply and then
2. scans any terminals (tokens) on the right hand side by calling upon the `Scanner`, and then
3. parses any non-terminals by recursively invoking the methods that parse them

This is the strategy we use in parsing *j*-- programs

We already saw the method `compilationUnit()` that parses a *j*-- `compilationUnit`

- Slides 02_compilation.pdf

As another example, consider the rule defining `qualifiedIdentifier`

qualifiedIdentifier ::= `<identifier>` {. `<identifier>`}

## Recursive descent

### Example: parsing a qualifiedIdentifier

Parsing by recursive descent involves writing a method (or procedure) for parsing each non-terminal according to the production rules that define that non-terminal

Depending on the next un-scanned input symbol, the method

1. decides which rule to apply and then
2. scans any terminals (tokens) on the right hand side by calling upon the `Scanner`, and then
3. parses any non-terminals by recursively invoking the methods that parse them

This is the strategy we use in parsing *j*-- programs

We already saw the method `compilationUnit()` that parses a *j*-- `compilationUnit`

- Slides 02_compilation.pdf

As another example, consider the rule defining `qualifiedIdentifier`

qualifiedIdentifier ::= `<identifier>` {. `<identifier>`}

## Recursive descent

### Example: parsing a qualifiedIdentifier

Parsing by recursive descent involves writing a method (or procedure) for parsing each non-terminal according to the production rules that define that non-terminal

Depending on the next un-scanned input symbol, the method

1. decides which rule to apply and then
2. scans any terminals (tokens) on the right hand side by calling upon the `Scanner`, and then
3. parses any non-terminals by recursively invoking the methods that parse them

This is the strategy we use in parsing *j*-- programs

We already saw the method `compilationUnit()` that parses a *j*-- `compilationUnit`

- Slides 02_compilation.pdf

As another example, consider the rule defining `qualifiedIdentifier`

qualifiedIdentifier ::= `<identifier>` {. `<identifier>`}

Parsing a qualifiedIdentifier such as `java.lang.Class` is straightforward

1. We look at the next incoming token and if it is an identifier, we scan it; if it is not an identifier, then we raise an error
2. Repeatedly, so long as the next incoming token is a period
   a. We scan the period
   b. We look at the next token and if it is an identifier, we scan it; otherwise, we raise an error

## Recursive descent

### Example: parsing a qualifiedIdentifier

The method for parsing `qualifiedIdentifier` not only parses the input, but also constructs an AST node for recording the fully qualified name as a string

```
private TypeName qualifiedIdentifier() {
    int line = scanner.token().line();
    mustBe(IDENTIFIER);
    String qualifiedIdentifier = scanner.previousToken().image();
    while (have(DOT)) {
        mustBe(IDENTIFIER);
        qualifiedIdentifier += "." + scanner.previousToken().image();
    }
    return new TypeName(line, qualifiedIdentifier);
}
```

The helper predicate method `have()` looks at the next incoming token (supplied by the `Scanner`), and if that token matches its argument, then it scans the token in the input and returns `true`; otherwise, it scans nothing and returns `false`

The helper method `mustBe()` requires that the next incoming token match its argument; it looks at the next token and if it matches its argument, it scans that token in the input, and it raises an error otherwise

`mustBe()` provides certain amount of error recovery as well

## Recursive descent

### Example: parsing a qualifiedIdentifier

The method for parsing `qualifiedIdentifier` not only parses the input, but also constructs an AST node for recording the fully qualified name as a string

```
private TypeName qualifiedIdentifier() {
    int line = scanner.token().line();
    mustBe(IDENTIFIER);
    String qualifiedIdentifier = scanner.previousToken().image();
    while (have(DOT)) {
        mustBe(IDENTIFIER);
        qualifiedIdentifier += "." + scanner.previousToken().image();
    }
    return new TypeName(line, qualifiedIdentifier);
}
```

The helper predicate method `have()` looks at the next incoming token (supplied by the `Scanner`), and if that token matches its argument, then it scans the token in the input and returns `true`; otherwise, it scans nothing and returns `false`

The helper method `mustBe()` requires that the next incoming token match its argument; it looks at the next token and if it matches its argument, it scans that token in the input, and it raises an error otherwise

`mustBe()` provides certain amount of error recovery as well

## Recursive descent

### Example: parsing a qualifiedIdentifier

The method for parsing `qualifiedIdentifier` not only parses the input, but also constructs an AST node for recording the fully qualified name as a string

```
private TypeName qualifiedIdentifier() {
    int line = scanner.token().line();
    mustBe(IDENTIFIER);
    String qualifiedIdentifier = scanner.previousToken().image();
    while (have(DOT)) {
        mustBe(IDENTIFIER);
        qualifiedIdentifier += "." + scanner.previousToken().image();
    }
    return new TypeName(line, qualifiedIdentifier);
}
```

The helper predicate method `have()` looks at the next incoming token (supplied by the `Scanner`), and if that token matches its argument, then it scans the token in the input and returns `true`; otherwise, it scans nothing and returns `false`

The helper method `mustBe()` requires that the next incoming token match its argument; it looks at the next token and if it matches its argument, it scans that token in the input, and it raises an error otherwise

`mustBe()` provides certain amount of error recovery as well

## Recursive descent

### Example: parsing a qualifiedIdentifier

The method for parsing `qualifiedIdentifier` not only parses the input, but also constructs an AST node for recording the fully qualified name as a string

```
private TypeName qualifiedIdentifier() {
    int line = scanner.token().line();
    mustBe(IDENTIFIER);
    String qualifiedIdentifier = scanner.previousToken().image();
    while (have(DOT)) {
        mustBe(IDENTIFIER);
        qualifiedIdentifier += "." + scanner.previousToken().image();
    }
    return new TypeName(line, qualifiedIdentifier);
}
```

The helper predicate method `have()` looks at the next incoming token (supplied by the `Scanner`), and if that token matches its argument, then it scans the token in the input and returns `true`; otherwise, it scans nothing and returns `false`

The helper method `mustBe()` requires that the next incoming token match its argument; it looks at the next token and if it matches its argument, it scans that token in the input, and it raises an error otherwise

`mustBe()` provides certain amount of error recovery as well

## Recursive descent

### Example: parsing a statement

As another example, consider our syntax for statements

statement ::= block
| `if` parExpression statement [`else` statement]
| `while` parExpression statement
| `return` [expression] ;
| ;
| statementExpression ;

The method for parsing a statement decides which rule to apply when looking at the next un-scanned token

```
private JStatement statement() {
    int line = scanner.token().line();
    if (see(LCURLY)) {
        return block();
    } else if (have(IF)) {
        JExpression test = parExpression();
        JStatement consequent = statement();
        JStatement alternate = have(ELSE) ? statement() : null;
        return new JIfStatement(line, test, consequent, alternate);
    ...
```

The helper predicate method `see()` simply looks at the incoming token and returns `true` if it is the token that we are looking for, and `false` otherwise; in no case is anything scanned

# Recursive descent

## Example: parsing a statement

As another example, consider our syntax for statements

$$
\begin{aligned}
\text{statement} ::= \; & \text{block} \\
& \mid \texttt{if}\ \text{parExpression statement}\ [\texttt{else}\ \text{statement}] \\
& \mid \texttt{while}\ \text{parExpression statement} \\
& \mid \texttt{return}\ [\text{expression}]\ \texttt{;} \\
& \mid \texttt{;} \\
& \mid \text{statementExpression}\ \texttt{;}
\end{aligned}
$$

The method for parsing a statement decides which rule to apply when looking at the next un-scanned token

```
private JStatement statement() {
    int line = scanner.token().line();
    if (see(LCURLY)) {
        return block();
    } else if (have(IF)) {
        JExpression test = parExpression();
        JStatement consequent = statement();
        JStatement alternate = have(ELSE) ? statement() : null;
        return new JIfStatement(line, test, consequent, alternate);
    ...
```

The helper predicate method `see()` simply looks at the incoming token and returns `true` if it is the token that we are looking for, and `false` otherwise; in no case is anything scanned

# Recursive descent

## Example: parsing a statement

```
        ...
    } else if (have(WHILE)) {
        JExpression test = parExpression();
        JStatement statement = statement();
        return new JWhileStatement(line, test, statement);
    } else if (have(RETURN)) {
        if (have(SEMI)) {
            return new JReturnStatement(line, null);
        } else {
            JExpression expr = expression();
            mustBe(SEMI);
            return new JReturnStatement(line, expr);
        }
    } else if (have(SEMI)) {
        return new JEmptyStatement(line);
    } else { // Must be a statementExpression
        JStatement statement = statementExpression();
        mustBe(SEMI);
        return statement;
    }
}
```

## Recursive descent with lookahead

The parsing of statements works because we can determine which rule to follow in parsing the statement based only on the next un-scanned symbol in the input source

Unfortunately, this is not always the case

Sometimes we must consider the next few symbols, ie, we must lookahead in the input stream of tokens to decide which rule to apply

For example, consider the syntax for a simple unary expression

$$
\begin{aligned}
\text{simpleUnaryExpression} ::= \; & \texttt{!}\ \text{unaryExpression} \\
| \; & \texttt{(}\ \text{basicType}\ \texttt{)}\ \ \text{unaryExpression}\ \text{//cast} \\
| \; & \texttt{(}\ \text{referenceType}\ \texttt{)}\ \text{simpleUnaryExpression}\ \text{// cast} \\
| \; & \text{postfixExpression}
\end{aligned}
$$

For this, we must not only differentiate between the two kinds (basic type and reference type) of casts, but we must also distinguish a cast from a postfix expression (i.e., a parenthesized expression like `(x)`)

## Recursive descent with lookahead

The parsing of statements works because we can determine which rule to follow in parsing the statement based only on the next un-scanned symbol in the input source

Unfortunately, this is not always the case

Sometimes we must consider the next few symbols, ie, we must lookahead in the input stream of tokens to decide which rule to apply

For example, consider the syntax for a simple unary expression

```
simpleUnaryExpression ::= ! unaryExpression
                        | ( basicType ) unaryExpression //cast
                        | ( referenceType ) simpleUnaryExpression // cast
                        | postfixExpression
```

For this, we must not only differentiate between the two kinds (basic type and reference type) of casts, but we must also distinguish a cast from a postfix expression (i.e., a parenthesized expression like `(x)`)

## Recursive descent with lookahead

The parsing of statements works because we can determine which rule to follow in parsing the statement based only on the next un-scanned symbol in the input source

Unfortunately, this is not always the case

Sometimes we must consider the next few symbols, ie, we must lookahead in the input stream of tokens to decide which rule to apply

For example, consider the syntax for a simple unary expression

```
simpleUnaryExpression ::= ! unaryExpression
                        | ( basicType ) unaryExpression //cast
                        | ( referenceType ) simpleUnaryExpression // cast
                        | postfixExpression
```

For this, we must not only differentiate between the two kinds (basic type and reference type) of casts, but we must also distinguish a cast from a postfix expression (i.e., a parenthesized expression like `(x)`)

## Recursive descent with lookahead

The parsing of statements works because we can determine which rule to follow in parsing the statement based only on the next un-scanned symbol in the input source

Unfortunately, this is not always the case

Sometimes we must consider the next few symbols, ie, we must lookahead in the input stream of tokens to decide which rule to apply

For example, consider the syntax for a simple unary expression

$$
\begin{aligned}
\text{simpleUnaryExpression} ::= \; & \texttt{!}\ \text{unaryExpression} \\
| \; & \texttt{(}\ \text{basicType}\ \texttt{)}\ \ \text{unaryExpression}\ \text{//cast} \\
| \; & \texttt{(}\ \text{referenceType}\ \texttt{)}\ \text{simpleUnaryExpression}\ \text{// cast} \\
| \; & \text{postfixExpression}
\end{aligned}
$$

For this, we must not only differentiate between the two kinds (basic type and reference type) of casts, but we must also distinguish a cast from a postfix expression (i.e., a parenthesized expression like `(x)`)

## Recursive descent with lookahead

The parsing of statements works because we can determine which rule to follow in parsing the statement based only on the next un-scanned symbol in the input source

Unfortunately, this is not always the case

Sometimes we must consider the next few symbols, ie, we must lookahead in the input stream of tokens to decide which rule to apply

For example, consider the syntax for a simple unary expression

$$
\begin{aligned}
\text{simpleUnaryExpression} ::= \; & \texttt{!}\ \text{unaryExpression} \\
| \; & \texttt{(}\ \text{basicType}\ \texttt{)}\ \ \text{unaryExpression}\ \text{//cast} \\
| \; & \texttt{(}\ \text{referenceType}\ \texttt{)}\ \text{simpleUnaryExpression}\ \text{// cast} \\
| \; & \text{postfixExpression}
\end{aligned}
$$

For this, we must not only differentiate between the two kinds (basic type and reference type) of casts, but we must also distinguish a cast from a postfix expression (i.e., a parenthesized expression like `(x)`)

## Recursive descent with lookahead: the case of `simpleUnaryExpression`

Consider the code for parsing a simple unary expression

```
private JExpression simpleUnaryExpression() {
    int line = scanner.token().line();
    if (have(LNOT)) {
        return new JLogicalNotOp(line, unaryExpression());
    } else if (seeCast()) {
        mustBe(LPAREN);
        boolean isBasicType = seeBasicType();
        Type type = type();
        mustBe(RPAREN);
        JExpression expr = isBasicType
            ? unaryExpression()
            : simpleUnaryExpression();
        return new JCastOp(line, type, expr);
    } else {
        return postfixExpression();
    }
}
```

We use the predicate `seeCast()` to distinguish casts from parenthesized expressions, and `seeBasicType()` to distinguish between casts to basic types from casts to reference types

Now, consider the two predicates

```
private boolean seeBasicType() {
    if (see(BOOLEAN) || see(CHAR) || see(INT)) {
        return true;
    } else {
        return false;
    }
}
```

## Recursive descent with lookahead: the case of `simpleUnaryExpression`

Consider the code for parsing a simple unary expression

```
private JExpression simpleUnaryExpression() {
    int line = scanner.token().line();
    if (have(LNOT)) {
        return new JLogicalNotOp(line, unaryExpression());
    } else if (seeCast()) {
        mustBe(LPAREN);
        boolean isBasicType = seeBasicType();
        Type type = type();
        mustBe(RPAREN);
        JExpression expr = isBasicType
            ? unaryExpression()
            : simpleUnaryExpression();
        return new JCastOp(line, type, expr);
    } else {
        return postfixExpression();
    }
}
```

We use the predicate `seeCast()` to distinguish casts from parenthesized expressions, and `seeBasicType()` to distinguish between casts to basic types from casts to reference types

Now, consider the two predicates

```
private boolean seeBasicType() {
    if (see(BOOLEAN) || see(CHAR) || see(INT)) {
        return true;
    } else {
        return false;
    }
}
```

# seeCast() needs to look ahead

```
private boolean seeCast() {
    scanner.recordPosition();
    if (!have(LPAREN)) {
        scanner.returnToPosition();
        return false;
    }
    if (seeBasicType()) {
        scanner.returnToPosition();
        return true;
    }
    if (!see(IDENTIFIER)) {
        scanner.returnToPosition();
        return false;
    } else {
        scanner.next(); // Scan the IDENTIFIER; a qualified identifier is ok
        while (have(DOT)) {
            if (!have(IDENTIFIER)) {
                scanner.returnToPosition();
                return false;
            }
        }
    }
    while (have(LBRACK)) {
        if (!have(RBRACK)) {
            scanner.returnToPosition();
            return false;
        }
    }
    if (!have(RPAREN)) {
        scanner.returnToPosition();
        return false;
    }
    scanner.returnToPosition();
    return true;
}
```

## Sometimes we need to look ahead

`seeCast()` must look ahead in the token stream to consider a sequence of tokens in making its decision, but our lexical analyzer `Scanner` keeps track of only the single incoming symbol

For this reason, we define a second lexical analyzer `LookaheadScanner` which encapsulates `Scanner` and provides extra machinery that allows us to look ahead in the token stream

`LookaheadScanner` includes a means of remembering tokens (and their images) that have been scanned, a method `recordPosition()` for marking a position in the token stream, and `returnToPosition()` for returning the lexical analyzer to that recorded position (backtracking)

Of course, calls to the two methods may be nested, so that one predicate (for example, `seeCast()`) may make use of another (for example, `seeBasicType()`), and therefore, all of this information must be recorded on a pushdown stack

## Sometimes we need to look ahead

`seeCast()` must look ahead in the token stream to consider a sequence of tokens in making its decision, but our lexical analyzer `Scanner` keeps track of only the single incoming symbol

For this reason, we define a second lexical analyzer `LookaheadScanner` which encapsulates `Scanner` and provides extra machinery that allows us to look ahead in the token stream

`LookaheadScanner` includes a means of remembering tokens (and their images) that have been scanned, a method `recordPosition()` for marking a position in the token stream, and `returnToPosition()` for returning the lexical analyzer to that recorded position (backtracking)

Of course, calls to the two methods may be nested, so that one predicate (for example, `seeCast()`) may make use of another (for example, `seeBasicType()`), and therefore, all of this information must be recorded on a pushdown stack

## Sometimes we need to look ahead

`seeCast()` must look ahead in the token stream to consider a sequence of tokens in making its decision, but our lexical analyzer `Scanner` keeps track of only the single incoming symbol

For this reason, we define a second lexical analyzer `LookaheadScanner` which encapsulates `Scanner` and provides extra machinery that allows us to look ahead in the token stream

`LookaheadScanner` includes a means of remembering tokens (and their images) that have been scanned, a method `recordPosition()` for marking a position in the token stream, and `returnToPosition()` for returning the lexical analyzer to that recorded position (backtracking)

Of course, calls to the two methods may be nested, so that one predicate (for example, `seeCast()`) may make use of another (for example, `seeBasicType()`), and therefore, all of this information must be recorded on a pushdown stack

## Sometimes we need to look ahead

`seeCast()` must look ahead in the token stream to consider a sequence of tokens in making its decision, but our lexical analyzer `Scanner` keeps track of only the single incoming symbol

For this reason, we define a second lexical analyzer `LookaheadScanner` which encapsulates `Scanner` and provides extra machinery that allows us to look ahead in the token stream

`LookaheadScanner` includes a means of remembering tokens (and their images) that have been scanned, a method `recordPosition()` for marking a position in the token stream, and `returnToPosition()` for returning the lexical analyzer to that recorded position (backtracking)

Of course, calls to the two methods may be nested, so that one predicate (for example, `seeCast()`) may make use of another (for example, `seeBasicType()`), and therefore, all of this information must be recorded on a pushdown stack

## Error recovery: How to recover from an error

What happens when the parser detects an error?

This will happen when `mustBe()` comes across a token that it is not expecting

The parser could simply report the error and quit

But we would rather have the parser report the error, and then continue parsing so that it might detect any additional syntax errors

The facility for continuing after an error is detected is called error recovery

Error recovery can be difficult; the parser must not only detect syntax errors but it must sufficiently recover its state so as to continue parsing without introducing additional spurious error messages

Many parser generators provide elaborate machinery for programming effective error recovery

For the *j*-- parser, we provide limited error recovery in the `mustBe()` method

## Error recovery: How to recover from an error

What happens when the parser detects an error?

This will happen when `mustBe()` comes across a token that it is not expecting

The parser could simply report the error and quit

But we would rather have the parser report the error, and then continue parsing so that it might detect any additional syntax errors

The facility for continuing after an error is detected is called error recovery

Error recovery can be difficult; the parser must not only detect syntax errors but it must sufficiently recover its state so as to continue parsing without introducing additional spurious error messages

Many parser generators provide elaborate machinery for programming effective error recovery

For the *j*-- parser, we provide limited error recovery in the `mustBe()` method

## Error recovery: How to recover from an error

What happens when the parser detects an error?

This will happen when `mustBe()` comes across a token that it is not expecting

The parser could simply report the error and quit

But we would rather have the parser report the error, and then continue parsing so that it might detect any additional syntax errors

The facility for continuing after an error is detected is called error recovery

Error recovery can be difficult; the parser must not only detect syntax errors but it must sufficiently recover its state so as to continue parsing without introducing additional spurious error messages

Many parser generators provide elaborate machinery for programming effective error recovery

For the *j*-- parser, we provide limited error recovery in the `mustBe()` method

## Error recovery: How to recover from an error

What happens when the parser detects an error?

This will happen when `mustBe()` comes across a token that it is not expecting

The parser could simply report the error and quit

But we would rather have the parser report the error, and then continue parsing so that it might detect any additional syntax errors

The facility for continuing after an error is detected is called error recovery

Error recovery can be difficult; the parser must not only detect syntax errors but it must sufficiently recover its state so as to continue parsing without introducing additional spurious error messages

Many parser generators provide elaborate machinery for programming effective error recovery

For the *j*-- parser, we provide limited error recovery in the `mustBe()` method

## Error recovery: How to recover from an error

What happens when the parser detects an error?

This will happen when `mustBe()` comes across a token that it is not expecting

The parser could simply report the error and quit

But we would rather have the parser report the error, and then continue parsing so that it might detect any additional syntax errors

The facility for continuing after an error is detected is called error recovery

Error recovery can be difficult; the parser must not only detect syntax errors but it must sufficiently recover its state so as to continue parsing without introducing additional spurious error messages

Many parser generators provide elaborate machinery for programming effective error recovery

For the *j*-- parser, we provide limited error recovery in the `mustBe()` method

## Error recovery: How to recover from an error

What happens when the parser detects an error?

This will happen when `mustBe()` comes across a token that it is not expecting

The parser could simply report the error and quit

But we would rather have the parser report the error, and then continue parsing so that it might detect any additional syntax errors

The facility for continuing after an error is detected is called error recovery

Error recovery can be difficult; the parser must not only detect syntax errors but it must sufficiently recover its state so as to continue parsing without introducing additional spurious error messages

Many parser generators provide elaborate machinery for programming effective error recovery

For the *j*-- parser, we provide limited error recovery in the `mustBe()` method

## Error recovery: How to recover from an error

What happens when the parser detects an error?

This will happen when `mustBe()` comes across a token that it is not expecting

The parser could simply report the error and quit

But we would rather have the parser report the error, and then continue parsing so that it might detect any additional syntax errors

The facility for continuing after an error is detected is called error recovery

Error recovery can be difficult; the parser must not only detect syntax errors but it must sufficiently recover its state so as to continue parsing without introducing additional spurious error messages

Many parser generators provide elaborate machinery for programming effective error recovery

For the *j*-- parser, we provide limited error recovery in the `mustBe()` method

## Error recovery: How to recover from an error

What happens when the parser detects an error?

This will happen when `mustBe()` comes across a token that it is not expecting

The parser could simply report the error and quit

But we would rather have the parser report the error, and then continue parsing so that it might detect any additional syntax errors

The facility for continuing after an error is detected is called error recovery

Error recovery can be difficult; the parser must not only detect syntax errors but it must sufficiently recover its state so as to continue parsing without introducing additional spurious error messages

Many parser generators provide elaborate machinery for programming effective error recovery

For the *j*-- parser, we provide limited error recovery in the `mustBe()` method

## Error recovery in the j-- compiler

First, consider the definitions for `see()` and `have()`

```
private boolean see(TokenKind sought) {
    return (sought == scanner.token().kind());
}

private boolean have(TokenKind sought) {
    if (see(sought)) {
        scanner.next();
        return true;
    } else {
        return false;
    }
}
```

The helper method `mustBe()`, defined in the next slide, makes use of a boolean flag, `isRecovered`, which is `true` if either no error has been detected or if the parser has recovered from a previous syntax error; it takes on the value `false` when it is in a state in which it has not yet recovered from a syntax error

## Error recovery in the j-- compiler

First, consider the definitions for `see()` and `have()`

```
private boolean see(TokenKind sought) {
    return (sought == scanner.token().kind());
}

private boolean have(TokenKind sought) {
    if (see(sought)) {
        scanner.next();
        return true;
    } else {
        return false;
    }
}
```

The helper method `mustBe()`, defined in the next slide, makes use of a boolean flag, `isRecovered`, which is `true` if either no error has been detected or if the parser has recovered from a previous syntax error; it takes on the value `false` when it is in a state in which it has not yet recovered from a syntax error

```
boolean isRecovered = true;

private void mustBe(TokenKind sought) {
    if (scanner.token().kind() == sought) {
        scanner.next();
        isRecovered = true;
    } else if (isRecovered) {
        isRecovered = false;
        reportParserError("%s found where %s sought", scanner
            .token().image(), sought.image());
    } else {
        // Do not report the (possibly spurious) error,
        // but rather attempt to recover by forcing a match.
        while (!see(sought) && !see(EOF)) {
            scanner.next();
        }
        if (see(sought)) {
            scanner.next();
            isRecovered = true;
        }
    }
}
```

When `mustBe()` first comes across an input token that it is not looking for (it is in the recovered state) it reports an error and goes into an un-recovered state

If, in a subsequent use of `mustBe()`, it finds another syntax error, it does not report the error, but rather it attempts to get back into a recovered state by repeatedly scanning tokens until it comes across the one it is seeking

If it succeeds in finding that token, it goes back into a recovered state; it may not succeed but instead scan to the end of the file, in which case the parsing stops

```
    boolean isRecovered = true;

    private void mustBe(TokenKind sought) {
        if (scanner.token().kind() == sought) {
            scanner.next();
            isRecovered = true;
        } else if (isRecovered) {
            isRecovered = false;
            reportParserError("%s found where %s sought", scanner
                .token().image(), sought.image());
        } else {
            // Do not report the (possibly spurious) error,
            // but rather attempt to recover by forcing a match.
            while (!see(sought) && !see(EOF)) {
                scanner.next();
            }
            if (see(sought)) {
                scanner.next();
                isRecovered = true;
            }
        }
    }
```

When `mustBe()` first comes across an input token that it is not looking for (it is in the recovered state) it reports an error and goes into an un-recovered state

If, in a subsequent use of `mustBe()`, it finds another syntax error, it does not report the error, but rather it attempts to get back into a recovered state by repeatedly scanning tokens until it comes across the one it is seeking

If it succeeds in finding that token, it goes back into a recovered state; it may not succeed but instead scan to the end of the file, in which case the parsing stops

```
    boolean isRecovered = true;

    private void mustBe(TokenKind sought) {
        if (scanner.token().kind() == sought) {
            scanner.next();
            isRecovered = true;
        } else if (isRecovered) {
            isRecovered = false;
            reportParserError("%s found where %s sought", scanner
                .token().image(), sought.image());
        } else {
            // Do not report the (possibly spurious) error,
            // but rather attempt to recover by forcing a match.
            while (!see(sought) && !see(EOF)) {
                scanner.next();
            }
            if (see(sought)) {
                scanner.next();
                isRecovered = true;
            }
        }
    }
```

When `mustBe()` first comes across an input token that it is not looking for (it is in the recovered state) it reports an error and goes into an un-recovered state

If, in a subsequent use of `mustBe()`, it finds another syntax error, it does not report the error, but rather it attempts to get back into a recovered state by repeatedly scanning tokens until it comes across the one it is seeking

If it succeeds in finding that token, it goes back into a recovered state; it may not succeed but instead scan to the end of the file, in which case the parsing stops

```
    boolean isRecovered = true;

    private void mustBe(TokenKind sought) {
        if (scanner.token().kind() == sought) {
            scanner.next();
            isRecovered = true;
        } else if (isRecovered) {
            isRecovered = false;
            reportParserError("%s found where %s sought", scanner
                .token().image(), sought.image());
        } else {
            // Do not report the (possibly spurious) error,
            // but rather attempt to recover by forcing a match.
            while (!see(sought) && !see(EOF)) {
                scanner.next();
            }
            if (see(sought)) {
                scanner.next();
                isRecovered = true;
            }
        }
    }
```

When `mustBe()` first comes across an input token that it is not looking for (it is in the recovered state) it reports an error and goes into an un-recovered state

If, in a subsequent use of `mustBe()`, it finds another syntax error, it does not report the error, but rather it attempts to get back into a recovered state by repeatedly scanning tokens until it comes across the one it is seeking

If it succeeds in finding that token, it goes back into a recovered state; it may not succeed but instead scan to the end of the file, in which case the parsing stops

```
    boolean isRecovered = true;

    private void mustBe(TokenKind sought) {
        if (scanner.token().kind() == sought) {
            scanner.next();
            isRecovered = true;
        } else if (isRecovered) {
            isRecovered = false;
            reportParserError("%s found where %s sought", scanner
                .token().image(), sought.image());
        } else {
            // Do not report the (possibly spurious) error,
            // but rather attempt to recover by forcing a match.
            while (!see(sought) && !see(EOF)) {
                scanner.next();
            }
            if (see(sought)) {
                scanner.next();
                isRecovered = true;
            }
        }
    }
```

When `mustBe()` first comes across an input token that it is not looking for (it is in the recovered state) it reports an error and goes into an un-recovered state

If, in a subsequent use of `mustBe()`, it finds another syntax error, it does not report the error, but rather it attempts to get back into a recovered state by repeatedly scanning tokens until it comes across the one it is seeking

If it succeeds in finding that token, it goes back into a recovered state; it may not succeed but instead scan to the end of the file, in which case the parsing stops

## Outline

1 Introduction

2 Context-free Grammars and Languages
- Backus-Naur Form (BNF)
- Left- and right-most derivation, & Parse tree
- Ambiguity

3 Top-down Deterministic Parsing
- Example: Top-down deterministic parsing of a *j*-- program
- Parsing by Recursive Descent
- Parsing by the LL(1) technique

4 Transform a non-LL(1) grammar into an equivalent LL(1) one

5 Parser Generation Using JavaCC

## The LL(1) parsing technique

Like recursive descent, the LL(1) technique is top-down, goal-oriented, and predictive

- Recursive descent uses the program stack for keeping track of recursive methods calls
- LL(1) makes this stack explicit

LL(1) stands for:

L indicates a left-to-right scan of the input token stream;

L indicates that it produces a left parse, which is a left-most derivation

1 indicates we just look ahead at the next one symbol in the input to make a decision

An LL(1) parser works in a top-down fashion:

- At the start, the start symbol is pushed onto the stack as the initial goal,
- depending on the first token in the source program, the start symbol is replaced by a sequence of symbols from the right-hand-side of a rule defining the start symbol
- Parsing continues by parsing each symbol as it is removed from the top of the stack
  - ⋆ If the top symbol is a terminal, it scans a terminal from the input; if the terminal does not match the terminal on the stack then an error is raised
  - ⋆ If the top symbol is a non-terminal, the parser looks at the next incoming token in the source program to decide what production rule to apply to replace that non-terminal

## The LL(1) parsing technique

Like recursive descent, the LL(1) technique is top-down, goal-oriented, and predictive

- Recursive descent uses the program stack for keeping track of recursive methods calls
- LL(1) makes this stack explicit

LL(1) stands for:

**L** indicates a left-to-right scan of the input token stream;

**L** indicates that it produces a left parse, which is a left-most derivation

**1** indicates we just look ahead at the next one symbol in the input to make a decision

An LL(1) parser works in a top-down fashion:

- At the start, the start symbol is pushed onto the stack as the initial goal,
- depending on the first token in the source program, the start symbol is replaced by a sequence of symbols from the right-hand-side of a rule defining the start symbol
- Parsing continues by parsing each symbol as it is removed from the top of the stack
  - ⋆ If the top symbol is a terminal, it scans a terminal from the input; if the terminal does not match the terminal on the stack then an error is raised
  - ⋆ If the top symbol is a non-terminal, the parser looks at the next incoming token in the source program to decide what production rule to apply to replace that non-terminal

## The LL(1) parsing technique

Like recursive descent, the LL(1) technique is top-down, goal-oriented, and predictive

- Recursive descent uses the program stack for keeping track of recursive methods calls
- LL(1) makes this stack explicit

LL(1) stands for:

**L** indicates a left-to-right scan of the input token stream;

**L** indicates that it produces a left parse, which is a left-most derivation

**1** indicates we just look ahead at the next one symbol in the input to make a decision

An LL(1) parser works in a top-down fashion:

- At the start, the start symbol is pushed onto the stack as the initial goal,
- depending on the first token in the source program, the start symbol is replaced by a sequence of symbols from the right-hand-side of a rule defining the start symbol
- Parsing continues by parsing each symbol as it is removed from the top of the stack
  - ⋆ If the top symbol is a terminal, it scans a terminal from the input; if the terminal does not match the terminal on the stack then an error is raised
  - ⋆ If the top symbol is a non-terminal, the parser looks at the next incoming token in the source program to decide what production rule to apply to replace that non-terminal

## The LL(1) parsing technique

Like recursive descent, the LL(1) technique is top-down, goal-oriented, and predictive

- Recursive descent uses the program stack for keeping track of recursive methods calls
- LL(1) makes this stack explicit

LL(1) stands for:

**L** indicates a left-to-right scan of the input token stream;

**L** indicates that it produces a left parse, which is a left-most derivation

**1** indicates we just look ahead at the next one symbol in the input to make a decision

An LL(1) parser works in a top-down fashion:

- At the start, the start symbol is pushed onto the stack as the initial goal,
- depending on the first token in the source program, the start symbol is replaced by a sequence of symbols from the right-hand-side of a rule defining the start symbol
- Parsing continues by parsing each symbol as it is removed from the top of the stack
  - ★ If the top symbol is a terminal, it scans a terminal from the input; if the terminal does not match the terminal on the stack then an error is raised
  - ★ If the top symbol is a non-terminal, the parser looks at the next incoming token in the source program to decide what production rule to apply to replace that non-terminal

LL(1) parsing technique is table-driven:

- a unique parse table is produced for each grammar
- the table has a row per non-terminal and a column per terminal, including a special terminator # to mark the end of the string
- given the non-terminal $E$ on top of the stack and the next incoming token, it determines the BNF rule to use to rewrite $E$

It is important that the grammar be such that one may always unambiguously decide what BNF rule to apply; equivalently, no table entry may contain more than one rule

For example, consider the following grammar

1. $E$ ::= $T\ E'$
2. $E'$ ::= + T $E'$
3. $E'$ ::= $\epsilon$
4. $T$ ::= $F\ T'$
5. $T'$ ::= * $F\ T'$
6. $T'$ ::= $\epsilon$
7. $F$ ::= (E)
8. $F$ ::= id

which describes the same language as the following grammar

$E$ ::= $T$ {+ $T$}
$T$ ::= $F$ {* $F$}
$F$ ::= (E) | id

LL(1) parsing technique is table-driven:

- a unique parse table is produced for each grammar
- the table has a row per non-terminal and a column per terminal, including a special terminator # to mark the end of the string
- given the non-terminal $E$ on top of the stack and the next incoming token, it determines the BNF rule to use to rewrite $E$

It is important that the grammar be such that one may always unambiguously decide what BNF rule to apply; equivalently, no table entry may contain more than one rule

For example, consider the following grammar

1. $E ::= T\ E'$
2. $E' ::= +\ \mathrm{T}\ E'$
3. $E' ::= \epsilon$
4. $T ::= F\ T'$
5. $T' ::= *\ F\ T'$
6. $T' ::= \epsilon$
7. $F ::= (E)$
8. $F ::= \mathtt{id}$

which describes the same language as the following grammar

$$E ::= T\ \{+\ T\}$$
$$T ::= F\ \{*\ F\}$$
$$F ::= (E) \mid \mathtt{id}$$

LL(1) parsing technique is table-driven:

- a unique parse table is produced for each grammar
- the table has a row per non-terminal and a column per terminal, including a special terminator # to mark the end of the string
- given the non-terminal $E$ on top of the stack and the next incoming token, it determines the BNF rule to use to rewrite $E$

It is important that the grammar be such that one may always unambiguously decide what BNF rule to apply; equivalently, no table entry may contain more than one rule

For example, consider the following grammar

1. $E ::= T\ E'$
2. $E' ::= +\ \mathrm{T}\ E'$
3. $E' ::= \epsilon$
4. $T ::= F\ T'$
5. $T' ::= *\ F\ T'$
6. $T' ::= \epsilon$
7. $F ::= (E)$
8. $F ::= \mathtt{id}$

which describes the same language as the following grammar

$$E ::= T\ \{+\ T\}$$
$$T ::= F\ \{*\ F\}$$
$$F ::= (E) \mid \mathtt{id}$$

LL(1) parsing technique is table-driven:

- a unique parse table is produced for each grammar
- the table has a row per non-terminal and a column per terminal, including a special terminator # to mark the end of the string
- given the non-terminal $E$ on top of the stack and the next incoming token, it determines the BNF rule to use to rewrite $E$

It is important that the grammar be such that one may always unambiguously decide what BNF rule to apply; equivalently, no table entry may contain more than one rule

For example, consider the following grammar

1. $E \ ::= T \ E'$
2. $E' ::= \texttt{+} \ \mathrm{T} \ E'$
3. $E' ::= \epsilon$
4. $T \ ::= F \ T'$
5. $T' ::= \texttt{*} \ F \ T'$
6. $T' ::= \epsilon$
7. $F \ ::= \texttt{(} E \texttt{)}$
8. $F \ ::= \texttt{id}$

which describes the same language as the following grammar

$E ::= T \ \{\texttt{+} \ T\}$
$T ::= F \ \{\texttt{*} \ F\}$
$F ::= \texttt{(} E \texttt{)} \mid \texttt{id}$

The parse table for the grammar is shown below; the numbers in the table's entries refer to the numbers assigned to BNF rules in the grammar

1. $E \ ::= T \ E'$
2. $E' ::= \texttt{+} \ \mathrm{T} \ E'$
3. $E' ::= \epsilon$
4. $T \ ::= F \ T'$
5. $T' ::= \texttt{*} \ F \ T'$
6. $T' ::= \epsilon$
7. $F \ ::= \texttt{(} E \texttt{)}$
8. $F \ ::= \texttt{id}$

| | + | * | ( | ) | id | # |
|---|---|---|---|---|---|---|
| E | | | 1 | | 1 | |
| E' | 2 | | | 3 | | 3 |
| T | | | 4 | | 4 | |
| T' | 6 | 5 | | 6 | | 6 |
| F | | | 7 | | 8 | |

The LL(1) parsing algorithm is parameterized by this table, and makes use of an explicit pushdown stack

**Algorithm** LL(1) Parsing Algorithm

**Require:** LL(1) parse table $table$, production rules $rules$, and a sentence $w$, where $w$ is a string of terminals followed by a terminator #

**Ensure:** a left-parse, which is a left-most derivation for $w$

The parse table for the grammar is shown below; the numbers in the table's entries refer to the numbers assigned to BNF rules in the grammar

1. $E \ ::= T \ E'$
2. $E' ::= \texttt{+} \ \mathrm{T} \ E'$
3. $E' ::= \epsilon$
4. $T \ ::= F \ T'$
5. $T' ::= \texttt{*} \ F \ T'$
6. $T' ::= \epsilon$
7. $F \ ::= \texttt{(} E \texttt{)}$
8. $F \ ::= \texttt{id}$

| | + | * | ( | ) | id | # |
|---|---|---|---|---|---|---|
| E | | | 1 | | 1 | |
| E' | 2 | | | 3 | | 3 |
| T | | | 4 | | 4 | |
| T' | 6 | 5 | | 6 | | 6 |
| F | | | 7 | | 8 | |

The LL(1) parsing algorithm is parameterized by this table, and makes use of an explicit pushdown stack

**Algorithm** LL(1) Parsing Algorithm

**Require:** LL(1) parse table $table$, production rules $rules$, and a sentence $w$, where $w$ is a string of terminals followed by a terminator #

**Ensure:** a left-parse, which is a left-most derivation for $w$

The parse table for the grammar is shown below; the numbers in the table's entries refer to the numbers assigned to BNF rules in the grammar

1. $E \ ::= T \ E'$
2. $E' ::= \texttt{+} \ \mathrm{T} \ E'$
3. $E' ::= \epsilon$
4. $T \ ::= F \ T'$
5. $T' ::= \texttt{*} \ F \ T'$
6. $T' ::= \epsilon$
7. $F \ ::= \texttt{(} E \texttt{)}$
8. $F \ ::= \texttt{id}$

| | + | * | ( | ) | id | # |
|---|---|---|---|---|---|---|
| E | | | 1 | | 1 | |
| E' | 2 | | | 3 | | 3 |
| T | | | 4 | | 4 | |
| T' | 6 | 5 | | 6 | | 6 |
| F | | | 7 | | 8 | |

The LL(1) parsing algorithm is parameterized by this table, and makes use of an explicit pushdown stack

**Algorithm** LL(1) Parsing Algorithm

**Require:** LL(1) parse table $table$, production rules $rules$, and a sentence $w$, where $w$ is a string of terminals followed by a terminator #

**Ensure:** a left-parse, which is a left-most derivation for $w$

**Algorithm** LL(1) Parsing Algorithm

**Require:** LL(1) parse table $table$, production rules $rules$, and a sentence $w$, where $w$ is a string of terminals followed by a terminator #
**Ensure:** a left-parse, which is a left-most derivation for $w$

```
Stack stk initially contains the terminator # and the start symbol S, with S on top
Symbol sym is the first symbol in the sentence w
while true do
    Symbol top ← stk.pop()
    if top = sym = # then
        Halt successfully
    else if top is a terminal symbol then
        if top = sym then
            Advance sym to the next symbol in w
        else
            Halt with an error: a sym found where a top was expected
        end if
    else if top is a non-terminal Y then
        index ← table[Y, sym]
        if index ≠ err then
            rule ← rules[index]
            Say rule is Y ::= X_1 X_2 ... X_n; push X_n, ..., X_2, X_1 onto the stack stk, with X_1 on top
        else
            Halt with an error
        end if
    end if
end while
```

**Algorithm** LL(1) Parsing Algorithm

**Require:** LL(1) parse table $table$, production rules $rules$, and a sentence $w$, where $w$ is a string of terminals followed by a terminator #

**Ensure:** a left-parse, which is a left-most derivation for $w$

```
Stack stk initially contains the terminator # and the start symbol S, with S on top
Symbol sym is the first symbol in the sentence w
while true do
    Symbol top ← stk.pop()
    if top = sym = # then
        Halt successfully
    else if top is a terminal symbol then
        if top = sym then
            Advance sym to the next symbol in w
        else
            Halt with an error: a sym found where a top was expected
        end if
    else if top is a non-terminal Y then
        index ← table[Y, sym]
        if index ≠ err then
            rule ← rules[index]
            Say rule is Y ::= X_1 X_2 ... X_n; push X_n, ..., X_2, X_1 onto the stack stk, with X_1 on top
        else
            Halt with an error
        end if
    end if
end while
```

# The steps in parsing `id + id * id` against the given LL(1) parse table

id+id*id#
Initial Configuration
(a)

| | + | * | ( | ) | id | # |
|---|---|---|---|---|---|---|
| E | | | 1 | | 1 | |
| E’ | 2 | | | 3 | | 3 |
| T | | | 4 | | 4 | |
| T’ | 6 | 5 | | 6 | | 6 |
| F | | | 7 | | 8 | |

# The steps in parsing `id + id * id` against the given LL(1) parse table

| | + | * | ( | ) | id | # |
|---|---|---|---|---|---|---|
| E | | | 1 | | 1 | |
| E' | 2 | | | 3 | | 3 |
| T | | | 4 | | 4 | |
| T' | 6 | 5 | | 6 | | 6 |
| F | | | 7 | | 8 | |

# The steps in parsing `id + id * id` against the given LL(1) parse table

| | + | * | ( | ) | id | # |
|---|---|---|---|---|---|---|
| E | | | 1 | | 1 | |
| E' | 2 | | | 3 | | 3 |
| T | | | 4 | | 4 | |
| T' | 6 | 5 | | 6 | | 6 |
| F | | | 7 | | 8 | |

# The steps in parsing `id + id * id` against the given LL(1) parse table

| | + | * | ( | ) | id | # |
|---|---|---|---|---|---|---|
| E | | | 1 | | 1 | |
| E’ | 2 | | | 3 | | 3 |
| T | | | 4 | | 4 | |
| T’ | 6 | 5 | | 6 | | 6 |
| F | | | 7 | | 8 | |

## An alternative way of illustrating the states that the parser goes through is as follows

| Stack | Input | Output |
|---|---|---|
| #E | id+id*id# | |
| #$E'T$ | id+id*id# | 1 |
| #$E'T'F$ | id+id*id# | 4 |
| #$E'T'$id | id+id*id# | 8 |
| #$E'T'$ | +id*id# | |
| #$E'$ | +id*id# | 6 |
| #$E'T$+ | +id*id# | 2 |
| #$E'T$ | id *id# | |
| #$E'T'F$ | id*id# | 4 |
| #$E'T'$id | id *id# | 8 |
| #$E'T'$ | *id# | |
| #$E'T'F$* | *id# | 5 |
| #$E'T'F$ | id# | |
| #$E'T'$id | id# | 8 |
| #$E'T'$ | # | |
| #$E'$ | # | 6 |
| # | # | 3 |

## How do we construct the LL(1) parse table?

### Case RHS starts with terminal

**table[$Y$, $a$] = $i$, where $i$ is the number of the rule $Y ::= X_1 X_2 \ldots X_n$**

if there is the goal $Y$ on the stack and the next un-scanned symbol is $a$, then we can rewrite $Y$ on the stack as the sequence of sub goals $X_1 X_2 \ldots X_n$

Consider the last two rules of our grammar

7. $F ::= \texttt{(} E \texttt{)}$
8. $F ::= \texttt{id}$

If we have the non-terminal $F$ on top of our parsing stack, the choice is simple:

- If the next un-scanned symbol is an open parenthesis `(`, we apply rule 7

  $\text{table}[F, \texttt{(}] = 7$

- If the next un-scanned symbol is an `id`, we apply rule 8

  $\text{table}[F, \texttt{id}] = 8$

## How do we construct the LL(1) parse table?

### Case RHS starts with terminal

**table[$Y$, $a$] = $i$, where $i$ is the number of the rule $Y ::= X_1 X_2 \ldots X_n$**

if there is the goal $Y$ on the stack and the next un-scanned symbol is $a$, then we can rewrite $Y$ on the stack as the sequence of sub goals $X_1 X_2 \ldots X_n$

Consider the last two rules of our grammar

7. $F ::=$ `(`$E$`)`
8. $F ::=$ `id`

If we have the non-terminal $F$ on top of our parsing stack, the choice is simple:

- If the next un-scanned symbol is an open parenthesis `(`, we apply rule 7

  $\text{table}[F, \texttt{(}] = 7$

- If the next un-scanned symbol is an `id`, we apply rule 8

  $\text{table}[F, \texttt{id}] = 8$

## How do we construct the LL(1) parse table?

### Case RHS starts with non-terminal or is $\epsilon$

The problem becomes slightly more complicated when the right-hand-side of the rule either starts with a non-terminal or is $\epsilon$

Let $\alpha$ and $\beta$ be (possibly empty) strings of terminals and non-terminals.
Then we have table$[Y, a] = i$, where rule $i$ is $Y ::= X_1 X_2 \ldots X_n$, if either

1. $X_1 X_2 \ldots X_n \overset{*}{\Rightarrow} a\alpha$, or
2. $X_1 X_2 \ldots X_n \overset{*}{\Rightarrow} \epsilon$, and S# $\overset{*}{\Rightarrow} \alpha Y a \beta$, (i.e., $a$ can follow $Y$ in a derivation).

For this we need two helper functions:

- first
- follow

## How do we construct the LL(1) parse table?

### Case RHS starts with non-terminal or is $\epsilon$

The problem becomes slightly more complicated when the right-hand-side of the rule either starts with a non-terminal or is $\epsilon$

Let $\alpha$ and $\beta$ be (possibly empty) strings of terminals and non-terminals.
Then we have table[$Y$, $a$] $= i$, where rule $i$ is $Y ::= X_1 X_2 \ldots X_n$, if either

1. $X_1 X_2 \ldots X_n \overset{*}{\Rightarrow} a\alpha$, or
2. $X_1 X_2 \ldots X_n \overset{*}{\Rightarrow} \epsilon$, and S# $\overset{*}{\Rightarrow} \alpha Y a \beta$, (i.e., $a$ can follow $Y$ in a derivation).

For this we need two helper functions:

- first
- follow

## How do we construct the LL(1) parse table?

### Case RHS starts with non-terminal or is $\epsilon$

The problem becomes slightly more complicated when the right-hand-side of the rule either starts with a non-terminal or is $\epsilon$

Let $\alpha$ and $\beta$ be (possibly empty) strings of terminals and non-terminals.
Then we have table[$Y$, $a$] $= i$, where rule $i$ is $Y ::= X_1 X_2 \ldots X_n$, if either

1. $X_1 X_2 \ldots X_n \overset{*}{\Rightarrow} a\alpha$, or
2. $X_1 X_2 \ldots X_n \overset{*}{\Rightarrow} \epsilon$, and S# $\overset{*}{\Rightarrow} \alpha Y a \beta$, (i.e., $a$ can follow $Y$ in a derivation).

For this we need two helper functions:

- first
- follow

## How do we construct the LL(1) parse table?

### Case RHS starts with non-terminal or is $\epsilon$

The problem becomes slightly more complicated when the right-hand-side of the rule either starts with a non-terminal or is $\epsilon$

Let $\alpha$ and $\beta$ be (possibly empty) strings of terminals and non-terminals.
Then we have table$[Y, a] = i$, where rule $i$ is $Y ::= X_1 X_2 \ldots X_n$, if either

1. $X_1 X_2 \ldots X_n \overset{*}{\Rightarrow} a\alpha$, or
2. $X_1 X_2 \ldots X_n \overset{*}{\Rightarrow} \epsilon$, and S# $\overset{*}{\Rightarrow} \alpha Y a \beta$, (i.e., $a$ can follow $Y$ in a derivation).

For this we need two helper functions:

- first
- follow

## How do we construct the LL(1) parse table?

### Case RHS starts with non-terminal or is $\epsilon$

The problem becomes slightly more complicated when the right-hand-side of the rule either starts with a non-terminal or is $\epsilon$

Let $\alpha$ and $\beta$ be (possibly empty) strings of terminals and non-terminals.
Then we have table$[Y, a] = i$, where rule $i$ is $Y ::= X_1 X_2 \ldots X_n$, if either

1. $X_1 X_2 \ldots X_n \overset{*}{\Rightarrow} a\alpha$, or
2. $X_1 X_2 \ldots X_n \overset{*}{\Rightarrow} \epsilon$, and S# $\overset{*}{\Rightarrow} \alpha Y a \beta$, (i.e., $a$ can follow $Y$ in a derivation).

For this we need two helper functions:

- first
- follow

## How do we construct the LL(1) parse table?

### Case RHS starts with non-terminal or is $\epsilon$: FIRST

We define, first$(X_1 X_2 \ldots X_n) = \{a | X_1 X_2 \ldots X_n \overset{*}{\Rightarrow} a\alpha, a \in T\}$:

- The set of all terminals that can start strings derivable from $X_1 X_2 \ldots X_n$;
- Also, if $X_1 X_2 \ldots X_n \overset{*}{\Rightarrow} \epsilon$, then first$(X_1 X_2 \ldots X_n)$ includes $\epsilon$

We define two algorithms for computing first:

- one for a single symbol, and one for a sequence of symbols
- the two algorithms are both mutually recursive and make use of memoization, ie, what was previously computed for each set is remembered upon each iteration

## How do we construct the LL(1) parse table?

### Case RHS starts with non-terminal or is $\epsilon$: FIRST

We define, first$(X_1 X_2 \ldots X_n) = \{a | X_1 X_2 \ldots X_n \overset{*}{\Rightarrow} a\alpha, a \in T\}$:

- The set of all terminals that can start strings derivable from $X_1 X_2 \ldots X_n$;
- Also, if $X_1 X_2 \ldots X_n \overset{*}{\Rightarrow} \epsilon$, then first$(X_1 X_2 \ldots X_n)$ includes $\epsilon$

We define two algorithms for computing first:

- one for a single symbol, and one for a sequence of symbols
- the two algorithms are both mutually recursive and make use of memoization, ie, what was previously computed for each set is remembered upon each iteration

## The algorithm for computing first for a single symbol

**Algorithm** Compute first($X$) for all symbols $X$ in a Grammar $G$

**Require:** a context-free grammar $G = (N, T, S, P)$
**Ensure:** first($X$) for all symbols $X$ in $G$

```
// Step 1: Initialization for terminals
for each terminal x do
    first(x) ← {x}
end for
// Step 2: Initialization for non-terminals
for each non-terminal X do
    first(X) ← {}
end for
// Step 3: Initialization for RHS with ϵ
if (X ::= ϵ) ∈ P then
    add ϵ to first(X)
end if
// Step 4: Actual computation, iterate many times over all rules
repeat
    for each (Y ::= X1 X2 ... Xn) ∈ P do
        add all symbols from first(X1 X2 ... Xn) to first(Y)
    end for
until no new symbols are added to any set
```

## The algorithm for computing first for a sequence of symbols

**Algorithm** Compute first$(X_1 X_2 \ldots X_n)$ for a Grammar $G$

**Require:** a context-free grammar $G = (N, T, S, P)$ and a sequence $X_1 X_2 \ldots X_n$
**Ensure:** first$(X_1 X_2 \ldots X_n)$
  Set $S \leftarrow$ first$(X_1)$
  $i \leftarrow 2$
  **while** $\epsilon \in S$ and $i \leq n$ **do**
    $S \leftarrow S - \epsilon$
    Add first$(X_i)$ to $S$
    $i \leftarrow i + 1$
  **end while**
  **return** $S$

In order to compute first$(X_1, \ldots, X_n)$:

- We start from first$(X_1)$
- If $X_1 \overset{*}{\Rightarrow} \epsilon$, then we must include also first$(X_2)$
- If $X_2 \overset{*}{\Rightarrow} \epsilon$, then we must include also first$(X_3)$
- ...
- If $X_{n-1} \overset{*}{\Rightarrow} \epsilon$, then we must include also first$(X_n)$

Consider our example grammar

1. $E \ ::= T \ E'$
2. $E' ::= \texttt{+} \ \mathrm{T} \ E'$
3. $E' ::= \epsilon$
4. $T \ ::= F \ T'$
5. $T' ::= \texttt{*} \ F \ T'$
6. $T' ::= \epsilon$
7. $F \ ::= \texttt{(} E \texttt{)}$
8. $F \ ::= \texttt{id}$

The computation of first for the terminals, by step 1 of the first algorithm is trivial

$$\begin{aligned}
\mathrm{first}(\texttt{+}) &= \{\texttt{+}\} \\
\mathrm{first}(\texttt{*}) &= \{\texttt{*}\} \\
\mathrm{first}(\texttt{(}) &= \{\texttt{(}\} \\
\mathrm{first}(\texttt{)}) &= \{\texttt{)}\} \\
\mathrm{first}(\texttt{id}) &= \{\texttt{id}\}
\end{aligned}$$

Step 2 of the first algorithm gives

$$\begin{aligned}
\mathrm{first}(E) &= \{\} \\
\mathrm{first}(E') &= \{\} \\
\mathrm{first}(T) &= \{\} \\
\mathrm{first}(T') &= \{\} \\
\mathrm{first}(F) &= \{\}
\end{aligned}$$

Consider our example grammar

1. $E \ ::= T \ E'$
2. $E' ::= \texttt{+} \ \mathrm{T} \ E'$
3. $E' ::= \epsilon$
4. $T \ ::= F \ T'$
5. $T' ::= \texttt{*} \ F \ T'$
6. $T' ::= \epsilon$
7. $F \ ::= \texttt{(} E \texttt{)}$
8. $F \ ::= \texttt{id}$

The computation of first for the terminals, by step 1 of the first algorithm is trivial

$$\begin{aligned}
\mathrm{first}(\texttt{+}) &= \{\texttt{+}\} \\
\mathrm{first}(\texttt{*}) &= \{\texttt{*}\} \\
\mathrm{first}(\texttt{(}) &= \{\texttt{(}\} \\
\mathrm{first}(\texttt{)}) &= \{\texttt{)}\} \\
\mathrm{first}(\texttt{id}) &= \{\texttt{id}\}
\end{aligned}$$

Step 2 of the first algorithm gives

$$\begin{aligned}
\mathrm{first}(E) &= \{\} \\
\mathrm{first}(E') &= \{\} \\
\mathrm{first}(T) &= \{\} \\
\mathrm{first}(T') &= \{\} \\
\mathrm{first}(F) &= \{\}
\end{aligned}$$

Consider our example grammar

1. $E \ ::= T \ E'$
2. $E' ::= \texttt{+} \ \mathrm{T} \ E'$
3. $E' ::= \epsilon$
4. $T \ ::= F \ T'$
5. $T' ::= \texttt{*} \ F \ T'$
6. $T' ::= \epsilon$
7. $F \ ::= \texttt{(} E \texttt{)}$
8. $F \ ::= \texttt{id}$

The computation of first for the terminals, by step 1 of the first algorithm is trivial

$$\begin{aligned}
\mathrm{first}(\texttt{+}) &= \{\texttt{+}\} \\
\mathrm{first}(\texttt{*}) &= \{\texttt{*}\} \\
\mathrm{first}(\texttt{(}) &= \{\texttt{(}\} \\
\mathrm{first}(\texttt{)}) &= \{\texttt{)}\} \\
\mathrm{first}(\texttt{id}) &= \{\texttt{id}\}
\end{aligned}$$

Step 2 of the first algorithm gives

$$\begin{aligned}
\mathrm{first}(E) &= \{\} \\
\mathrm{first}(E') &= \{\} \\
\mathrm{first}(T) &= \{\} \\
\mathrm{first}(T') &= \{\} \\
\mathrm{first}(F) &= \{\}
\end{aligned}$$

Step 3 of the first algorithm yields

$$\text{first}(E') = \{\epsilon\}$$
$$\text{first}(T') = \{\epsilon\}$$

Step 4 of the first algorithm is repeatedly executed until no further symbols are added; the first round yields

$$\text{first}(\text{E}) = \{\}$$
$$\text{first}(E') = \{\texttt{+}, \epsilon\}$$
$$\text{first}(T) = \{\}$$
$$\text{first}(T') = \{\texttt{*}, \epsilon\}$$
$$\text{first}(F) = \{\texttt{(}, \texttt{id}\}$$

Second round of step 4 yields

$$\text{first}(E) = \{\}$$
$$\text{first}(E') = \{\texttt{+}, \epsilon\}$$
$$\text{first}(T) = \{\texttt{(}, \texttt{id}\}$$
$$\text{first}(T') = \{\texttt{*}, \epsilon\}$$
$$\text{first}(F) = \{\texttt{(}, \texttt{id}\}$$

Step 3 of the first algorithm yields

$$\text{first}(E') = \{\epsilon\}$$
$$\text{first}(T') = \{\epsilon\}$$

Step 4 of the first algorithm is repeatedly executed until no further symbols are added; the first round yields

$$\text{first}(\text{E}) = \{\}$$
$$\text{first}(E') = \{\texttt{+},\ \epsilon\}$$
$$\text{first}(T) = \{\}$$
$$\text{first}(T') = \{\texttt{*},\ \epsilon\}$$
$$\text{first}(F) = \{\texttt{(},\ \texttt{id}\}$$

Second round of step 4 yields

$$\text{first}(E) = \{\}$$
$$\text{first}(E') = \{\texttt{+},\ \epsilon\}$$
$$\text{first}(T) = \{\texttt{(},\ \texttt{id}\}$$
$$\text{first}(T') = \{\texttt{*},\ \epsilon\}$$
$$\text{first}(F) = \{\texttt{(},\ \texttt{id}\}$$

Step 3 of the first algorithm yields

$$\begin{aligned}\text{first}(E') &= \{\epsilon\}\\ \text{first}(T') &= \{\epsilon\}\end{aligned}$$

Step 4 of the first algorithm is repeatedly executed until no further symbols are added; the first round yields

$$\begin{aligned}\text{first}(\text{E}) &= \{\}\\ \text{first}(E') &= \{\texttt{+},\ \epsilon\}\\ \text{first}(T) &= \{\}\\ \text{first}(T') &= \{\texttt{*},\ \epsilon\}\\ \text{first}(F) &= \{\texttt{(},\ \texttt{id}\}\end{aligned}$$

Second round of step 4 yields

$$\begin{aligned}\text{first}(E) &= \{\}\\ \text{first}(E') &= \{\texttt{+},\ \epsilon\}\\ \text{first}(T) &= \{\texttt{(},\ \texttt{id}\}\\ \text{first}(T') &= \{\texttt{*},\ \epsilon\}\\ \text{first}(F) &= \{\texttt{(},\ \texttt{id}\}\end{aligned}$$

Step 3 of the first algorithm yields

$$\begin{aligned}\text{first}(E') &= \{\epsilon\}\\ \text{first}(T') &= \{\epsilon\}\end{aligned}$$

Step 4 of the first algorithm is repeatedly executed until no further symbols are added; the first round yields

$$\begin{aligned}\text{first}(\text{E}) &= \{\}\\ \text{first}(E') &= \{\texttt{+},\ \epsilon\}\\ \text{first}(T) &= \{\}\\ \text{first}(T') &= \{\texttt{*},\ \epsilon\}\\ \text{first}(F) &= \{\texttt{(},\ \texttt{id}\}\end{aligned}$$

Second round of step 4 yields

$$\begin{aligned}\text{first}(E) &= \{\}\\ \text{first}(E') &= \{\texttt{+},\ \epsilon\}\\ \text{first}(T) &= \{\texttt{(},\ \texttt{id}\}\\ \text{first}(T') &= \{\texttt{*},\ \epsilon\}\\ \text{first}(F) &= \{\texttt{(},\ \texttt{id}\}\end{aligned}$$

Third round of step 4 yields

$$\begin{aligned}
\text{first}(E) &= \{\texttt{(}, \texttt{id}\} \\
\text{first}(E') &= \{\texttt{+}, \epsilon\} \\
\text{first}(T) &= \{\texttt{(}, \texttt{id}\} \\
\text{first}(T') &= \{\texttt{*}, \epsilon\} \\
\text{first}(F) &= \{\texttt{(}, \texttt{id}\}
\end{aligned}$$

The fourth round of step 4 adds no symbols to any set, so we are done

Third round of step 4 yields

$$\begin{aligned}
\text{first}(E) &= \{\texttt{(}, \texttt{id}\} \\
\text{first}(E') &= \{\texttt{+}, \epsilon\} \\
\text{first}(T) &= \{\texttt{(}, \texttt{id}\} \\
\text{first}(T') &= \{\texttt{*}, \epsilon\} \\
\text{first}(F) &= \{\texttt{(}, \texttt{id}\}
\end{aligned}$$

The fourth round of step 4 adds no symbols to any set, so we are done

## How do we construct the LL(1) parse table?

### Case RHS starts with non-terminal or is $\epsilon$: FOLLOW

When is a rule $X ::= \epsilon$ applicable? For this we need the notion of follow

We define, follow$(X) = \{a | S \overset{*}{\Rightarrow} wX\alpha$ and $\alpha \overset{*}{\Rightarrow} a \ldots\}$

- All terminals that start terminal strings derivable from what can follow $X$ in a derivation

Another definition is:

1. follow$(S)$ contains #, that is, the terminator follows the start symbol
2. If there is a rule $Y ::= \alpha X \beta$ in $P$, follow$(X)$ contains first$(\beta) - \{\epsilon\}$
3. If there is a rule $Y ::= \alpha X \beta$ in $P$ and either $\beta = \epsilon$ or first$(\beta)$ contains $\epsilon$, follow$(X)$ contains follow$(Y)$

## How do we construct the LL(1) parse table?

### Case RHS starts with non-terminal or is $\epsilon$: FOLLOW

When is a rule $X ::= \epsilon$ applicable? For this we need the notion of follow

We define, follow$(X) = \{a | S \overset{*}{\Rightarrow} wX\alpha$ and $\alpha \overset{*}{\Rightarrow} a \ldots\}$

- All terminals that start terminal strings derivable from what can follow $X$ in a derivation

Another definition is:

1. follow$(S)$ contains #, that is, the terminator follows the start symbol
2. If there is a rule $Y ::= \alpha X \beta$ in $P$, follow$(X)$ contains first$(\beta) - \{\epsilon\}$
3. If there is a rule $Y ::= \alpha X \beta$ in $P$ and either $\beta = \epsilon$ or first$(\beta)$ contains $\epsilon$, follow$(X)$ contains follow$(Y)$

## How do we construct the LL(1) parse table?

### Case RHS starts with non-terminal or is $\epsilon$: FOLLOW

When is a rule $X ::= \epsilon$ applicable? For this we need the notion of follow

We define, follow$(X) = \{a | S \overset{*}{\Rightarrow} wX\alpha$ and $\alpha \overset{*}{\Rightarrow} a \ldots\}$

- All terminals that start terminal strings derivable from what can follow $X$ in a derivation

Another definition is:

1. follow$(S)$ contains #, that is, the terminator follows the start symbol
2. If there is a rule $Y ::= \alpha X \beta$ in $P$, follow$(X)$ contains first$(\beta) - \{\epsilon\}$
3. If there is a rule $Y ::= \alpha X \beta$ in $P$ and either $\beta = \epsilon$ or first$(\beta)$ contains $\epsilon$, follow$(X)$ contains follow$(Y)$

## The algorithm for computing follow

The definition of follow from the previous slide

1. follow($S$) contains #, that is, the terminator follows the start symbol
2. If there is a rule $Y ::= \alpha X \beta$ in $P$, follow($X$) contains first($\beta$) $- \{\epsilon\}$
3. If there is a rule $Y ::= \alpha X \beta$ in $P$ and $\beta = \epsilon$ or $\epsilon \in$ first($\beta$), follow($X$) contains follow($Y$)

**Algorithm** Compute follow($X$) for all non-terminals $X$ in a Grammar $G$

**Require:** a context-free grammar $G = (N, T, S, P)$
**Ensure:** follow($X$) for all non-terminals $X$ in $G$

```
// Step 1: Initialize all sets
for each non-terminal X ∈ N do
    follow(X) ← {}
end for
// Step 2: Initialize the set of the start symbol
follow(S) ← {#}
// Step 3: Actual computation
repeat
    for each rule (Y ::= X1 X2 ... Xn) ∈ P do
        for each non-terminal Xi in the above X1 X2 ... Xn do
            if i < n then
                Add first(Xi+1 Xi+2 ... Xn) − {ε} to follow(Xi),
            end if
            if i == n or (ε ∈ first(Xi+1 Xi+2 ... Xn)) then
                Add follow(Y) to follow(Xi)
            end if
        end for
    end for
until no new symbols are added to any set
```

## The algorithm for computing follow

The definition of follow from the previous slide

1. follow($S$) contains #, that is, the terminator follows the start symbol
2. If there is a rule $Y ::= \alpha X \beta$ in $P$, follow($X$) contains first($\beta$) $- \{\epsilon\}$
3. If there is a rule $Y ::= \alpha X \beta$ in $P$ and $\beta = \epsilon$ or $\epsilon \in$ first($\beta$), follow($X$) contains follow($Y$)

**Algorithm** Compute follow($X$) for all non-terminals $X$ in a Grammar $G$

**Require:** a context-free grammar $G = (N, T, S, P)$
**Ensure:** follow($X$) for all non-terminals $X$ in $G$

```
// Step 1: Initialize all sets
for each non-terminal X ∈ N do
    follow(X) ← {}
end for
// Step 2: Initialize the set of the start symbol
follow(S) ← {#}
// Step 3: Actual computation
repeat
    for each rule (Y ::= X1 X2 ... Xn) ∈ P do
        for each non-terminal Xi in the above X1 X2 ... Xn do
            if i < n then
                Add first(Xi+1 Xi+2 ... Xn) − {ε} to follow(Xi),
            end if
            if i == n or (ε ∈ first(Xi+1 Xi+2 ... Xn)) then
                Add follow(Y) to follow(Xi)
            end if
        end for
    end for
until no new symbols are added to any set
```

## Again, consider our example grammar

| | | |
|---|---|---|
| 1. $E ::= T\ E'$ | $\text{first}(E) = \{\texttt{(}, \texttt{id}\}$ | $\text{follow}(E) = \{\texttt{\#}\}$ |
| 2. $E' ::= \texttt{+}\ \text{T}\ E'$ | $\text{first}(E') = \{\texttt{+}, \epsilon\}$ | $\text{follow}(E') = \{\}$ |
| 3. $E' ::= \epsilon$ | $\text{first}(T) = \{\texttt{(}, \texttt{id}\}$ | $\text{follow}(T) = \{\}$ |
| 4. $T ::= F\ T'$ | $\text{first}(T') = \{\texttt{*}, \epsilon\}$ | $\text{follow}(T') = \{\}$ |
| 5. $T' ::= \texttt{*}\ F\ T'$ | $\text{first}(F) = \{\texttt{(}, \texttt{id}\}$ | $\text{follow}(F) = \{\}$ |
| 6. $T' ::= \epsilon$ | | |
| 7. $F ::= \texttt{(}E\texttt{)}$ | | |
| 8. $F ::= \texttt{id}$ | | |

From steps 1 and 2 we get the above initial 'follow' sets

From rule 1, follow($T$) contains first($E'$) - $\{\epsilon\} = \{+\}$, and because first($E'$) contains $\epsilon$, it also contains follow($E$); also, follow($E'$) contains follow($E$); this yields:

$$\text{follow}(E') = \{\texttt{\#}\}$$
$$\text{follow}(T) = \{\texttt{+}, \texttt{\#}\}$$

## Again, consider our example grammar

| | | |
|---|---|---|
| 1. $E ::= T\ E'$ | $\text{first}(E) = \{\texttt{(}, \texttt{id}\}$ | $\text{follow}(E) = \{\texttt{\#}\}$ |
| 2. $E' ::= \texttt{+}\ \text{T}\ E'$ | $\text{first}(E') = \{\texttt{+}, \epsilon\}$ | $\text{follow}(E') = \{\}$ |
| 3. $E' ::= \epsilon$ | $\text{first}(T) = \{\texttt{(}, \texttt{id}\}$ | $\text{follow}(T) = \{\}$ |
| 4. $T ::= F\ T'$ | $\text{first}(T') = \{\texttt{*}, \epsilon\}$ | $\text{follow}(T') = \{\}$ |
| 5. $T' ::= \texttt{*}\ F\ T'$ | $\text{first}(F) = \{\texttt{(}, \texttt{id}\}$ | $\text{follow}(F) = \{\}$ |
| 6. $T' ::= \epsilon$ | | |
| 7. $F ::= \texttt{(}E\texttt{)}$ | | |
| 8. $F ::= \texttt{id}$ | | |

From steps 1 and 2 we get the above initial 'follow' sets

From rule 1, follow($T$) contains first($E'$) - $\{\epsilon\} = \{+\}$, and because first($E'$) contains $\epsilon$, it also contains follow($E$); also, follow($E'$) contains follow($E$); this yields:

$$\text{follow}(E') = \{\texttt{\#}\}$$
$$\text{follow}(T) = \{\texttt{+}, \texttt{\#}\}$$

## Again, consider our example grammar

| | | |
|---|---|---|
| 1. $E ::= T\ E'$ | $\text{first}(E) = \{\texttt{(}, \texttt{id}\}$ | $\text{follow}(E) = \{\texttt{\#}\}$ |
| 2. $E' ::= \texttt{+}\ \text{T}\ E'$ | $\text{first}(E') = \{\texttt{+}, \epsilon\}$ | $\text{follow}(E') = \{\texttt{\#}\}$ |
| 3. $E' ::= \epsilon$ | $\text{first}(T) = \{\texttt{(}, \texttt{id}\}$ | $\text{follow}(T) = \{\texttt{+}, \texttt{\#}\}$ |
| 4. $T ::= F\ T'$ | $\text{first}(T') = \{\texttt{*}, \epsilon\}$ | $\text{follow}(T') = \{\}$ |
| 5. $T' ::= \texttt{*}\ F\ T'$ | $\text{first}(F) = \{\texttt{(}, \texttt{id}\}$ | $\text{follow}(F) = \{\}$ |
| 6. $T' ::= \epsilon$ | | |
| 7. $F ::= \texttt{(}E\texttt{)}$ | | |
| 8. $F ::= \texttt{id}$ | | |

From steps 1 and 2 we get the above initial 'follow' sets

From rule 1, follow($T$) contains first($E'$) - $\{\epsilon\} = \{+\}$, and because first($E'$) contains $\epsilon$, it also contains follow($E$); also, follow($E'$) contains follow($E$); this yields:

$$\text{follow}(E') = \{\texttt{\#}\}$$
$$\text{follow}(T) = \{\texttt{+}, \texttt{\#}\}$$

## Again, consider our example grammar

| | | |
|---|---|---|
| 1. $E$ ::= $T\ E'$ | first($E$) = {(, id} | follow($E$) = {#} |
| 2. $E'$ ::= + T $E'$ | first($E'$) = {+, $\epsilon$} | follow($E'$) = {#} |
| 3. $E'$ ::= $\epsilon$ | first($T$) = {(, id} | follow($T$) = {+, #} |
| 4. $T$ ::= $F\ T'$ | first($T'$) = {*, $\epsilon$} | follow($T'$) = {} |
| 5. $T'$ ::= * $F\ T'$ | first($F$) = {(, id} | follow($F$) = {} |
| 6. $T'$ ::= $\epsilon$ | | |
| 7. $F$ ::= (E) | | |
| 8. $F$ ::= id | | |

From steps 1 and 2 we get the above initial 'follow' sets

From rule 1, follow($T$) contains first($E'$) - $\{\epsilon\} = \{+\}$, and because first($E'$) contains $\epsilon$, it also contains follow($E$); also, follow($E'$) contains follow($E$); this yields:

$$\text{follow}(E') = \{\#\}$$
$$\text{follow}(T) = \{+, \#\}$$

We get nothing additional from rule 2: follow($T$) contains first($E'$) - $\{\epsilon\}$, but we saw that in rule 1; also, follow($E'$) contains follow($E'$), but that says nothing.

We get nothing additional from rule 3

## Again, consider our example grammar

| | | |
|---|---|---|
| 1. $E$ ::= $T\ E'$ | first($E$) = {(, id} | follow($E$) = {#} |
| 2. $E'$ ::= + T $E'$ | first($E'$) = {+, $\epsilon$} | follow($E'$) = {#} |
| 3. $E'$ ::= $\epsilon$ | first($T$) = {(, id} | follow($T$) = {+, #} |
| 4. $T$ ::= $F\ T'$ | first($T'$) = {*, $\epsilon$} | follow($T'$) = {} |
| 5. $T'$ ::= * $F\ T'$ | first($F$) = {(, id} | follow($F$) = {} |
| 6. $T'$ ::= $\epsilon$ | | |
| 7. $F$ ::= (E) | | |
| 8. $F$ ::= id | | |

From steps 1 and 2 we get the above initial 'follow' sets

From rule 1, follow($T$) contains first($E'$) - $\{\epsilon\} = \{+\}$, and because first($E'$) contains $\epsilon$, it also contains follow($E$); also, follow($E'$) contains follow($E$); this yields:

$$\text{follow}(E') = \{\#\}$$
$$\text{follow}(T) = \{+, \#\}$$

We get nothing additional from rule 2: follow($T$) contains first($E'$) - $\{\epsilon\}$, but we saw that in rule 1; also, follow($E'$) contains follow($E'$), but that says nothing.

We get nothing additional from rule 3

## Again, consider our example grammar

| | | |
|---|---|---|
| 1. $E ::= T\ E'$ | $\text{first}(E) = \{\texttt{(}, \texttt{id}\}$ | $\text{follow}(E) = \{\texttt{\#}\}$ |
| 2. $E' ::= \texttt{+}\ \text{T}\ E'$ | $\text{first}(E') = \{\texttt{+}, \epsilon\}$ | $\text{follow}(E') = \{\texttt{\#}\}$ |
| 3. $E' ::= \epsilon$ | $\text{first}(T) = \{\texttt{(}, \texttt{id}\}$ | $\text{follow}(T) = \{\texttt{+}, \texttt{\#}\}$ |
| 4. $T ::= F\ T'$ | $\text{first}(T') = \{\texttt{*}, \epsilon\}$ | $\text{follow}(T') = \{\}$ |
| 5. $T' ::= \texttt{*}\ F\ T'$ | $\text{first}(F) = \{\texttt{(}, \texttt{id}\}$ | $\text{follow}(F) = \{\}$ |
| 6. $T' ::= \epsilon$ | | |
| 7. $F ::= \texttt{(}E\texttt{)}$ | | |
| 8. $F ::= \texttt{id}$ | | |

From rule 4, follow($F$) contains first($T'$) - $\{\epsilon\} = \{\texttt{*}\}$, and because first($T'$) contains $\epsilon$, it also contains follow($T$); also, follow($T'$) contains follow($T$); so, we have

$$\text{follow}(T') = \{\texttt{+}, \texttt{\#}\}$$
$$\text{follow}(F) = \{\texttt{+}, \texttt{*}, \texttt{\#}\}$$

## Again, consider our example grammar

| | | |
|---|---|---|
| 1. $E ::= T\ E'$ | $\text{first}(E) = \{\texttt{(}, \texttt{id}\}$ | $\text{follow}(E) = \{\texttt{\#}\}$ |
| 2. $E' ::= \texttt{+}\ \text{T}\ E'$ | $\text{first}(E') = \{\texttt{+}, \epsilon\}$ | $\text{follow}(E') = \{\texttt{\#}\}$ |
| 3. $E' ::= \epsilon$ | $\text{first}(T) = \{\texttt{(}, \texttt{id}\}$ | $\text{follow}(T) = \{\texttt{+}, \texttt{\#}\}$ |
| 4. $T ::= F\ T'$ | $\text{first}(T') = \{\texttt{*}, \epsilon\}$ | $\text{follow}(T') = \{\texttt{+}, \texttt{\#}\}$ |
| 5. $T' ::= \texttt{*}\ F\ T'$ | $\text{first}(F) = \{\texttt{(}, \texttt{id}\}$ | $\text{follow}(F) = \{\texttt{+}, \texttt{*}, \texttt{\#}\}$ |
| 6. $T' ::= \epsilon$ | | |
| 7. $F ::= \texttt{(}E\texttt{)}$ | | |
| 8. $F ::= \texttt{id}$ | | |

From rule 4, follow($F$) contains first($T'$) - $\{\epsilon\} = \{\texttt{*}\}$, and because first($T'$) contains $\epsilon$, it also contains follow($T$); also, follow($T'$) contains follow($T$); so, we have

$$\text{follow}(T') = \{\texttt{+}, \texttt{\#}\}$$
$$\text{follow}(F) = \{\texttt{+}, \texttt{*}, \texttt{\#}\}$$

## Again, consider our example grammar

| | | |
|---|---|---|
| 1. $E ::= T\ E'$ | $\text{first}(E) = \{\texttt{(}, \texttt{id}\}$ | $\text{follow}(E) = \{\texttt{\#}\}$ |
| 2. $E' ::= \texttt{+}\ \text{T}\ E'$ | $\text{first}(E') = \{\texttt{+}, \epsilon\}$ | $\text{follow}(E') = \{\texttt{\#}\}$ |
| 3. $E' ::= \epsilon$ | $\text{first}(T) = \{\texttt{(}, \texttt{id}\}$ | $\text{follow}(T) = \{\texttt{+}, \texttt{\#}\}$ |
| 4. $T ::= F\ T'$ | $\text{first}(T') = \{\texttt{*}, \epsilon\}$ | $\text{follow}(T') = \{\texttt{+}, \texttt{\#}\}$ |
| 5. $T' ::= \texttt{*}\ F\ T'$ | $\text{first}(F) = \{\texttt{(}, \texttt{id}\}$ | $\text{follow}(F) = \{\texttt{+}, \texttt{*}, \texttt{\#}\}$ |
| 6. $T' ::= \epsilon$ | | |
| 7. $F ::= \texttt{(}E\texttt{)}$ | | |
| 8. $F ::= \texttt{id}$ | | |

From rule 4, follow($F$) contains first($T'$) - $\{\epsilon\} = \{\texttt{*}\}$, and because first($T'$) contains $\epsilon$, it also contains follow($T$); also, follow($T'$) contains follow($T$); so, we have

$$\text{follow}(T') = \{\texttt{+}, \texttt{\#}\}$$
$$\text{follow}(F) = \{\texttt{+}, \texttt{*}, \texttt{\#}\}$$

Rules 5 and 6 give us nothing new (for the same reasons rules 2 and 3 did not)

## Again, consider our example grammar

| | | |
|---|---|---|
| 1. $E ::= T\ E'$ | $\text{first}(E) = \{\texttt{(}, \texttt{id}\}$ | $\text{follow}(E) = \{\texttt{\#}\}$ |
| 2. $E' ::= \texttt{+}\ \text{T}\ E'$ | $\text{first}(E') = \{\texttt{+}, \epsilon\}$ | $\text{follow}(E') = \{\texttt{\#}\}$ |
| 3. $E' ::= \epsilon$ | $\text{first}(T) = \{\texttt{(}, \texttt{id}\}$ | $\text{follow}(T) = \{\texttt{+}, \texttt{\#}\}$ |
| 4. $T ::= F\ T'$ | $\text{first}(T') = \{\texttt{*}, \epsilon\}$ | $\text{follow}(T') = \{\texttt{+}, \texttt{\#}\}$ |
| 5. $T' ::= \texttt{*}\ F\ T'$ | $\text{first}(F) = \{\texttt{(}, \texttt{id}\}$ | $\text{follow}(F) = \{\texttt{+}, \texttt{*}, \texttt{\#}\}$ |
| 6. $T' ::= \epsilon$ | | |
| 7. $F ::= \texttt{(}E\texttt{)}$ | | |
| 8. $F ::= \texttt{id}$ | | |

From rule 4, follow($F$) contains first($T'$) - $\{\epsilon\} = \{\texttt{*}\}$, and because first($T'$) contains $\epsilon$, it also contains follow($T$); also, follow($T'$) contains follow($T$); so, we have

$$\text{follow}(T') = \{\texttt{+}, \texttt{\#}\}$$
$$\text{follow}(F) = \{\texttt{+}, \texttt{*}, \texttt{\#}\}$$

Rules 5 and 6 give us nothing new (for the same reasons rules 2 and 3 did not)

Rule 7, $F ::= \texttt{(}E\texttt{)}$, adds ) to follow($E$), so follow($E$) $= \{\texttt{)}, \texttt{\#}\}$

## Again, consider our example grammar

| | | |
|---|---|---|
| 1. $E ::= T\ E'$ | $\text{first}(E) = \{\texttt{(}, \texttt{id}\}$ | $\text{follow}(E) = \{\texttt{)}, \texttt{\#}\}$ |
| 2. $E' ::= \texttt{+}\ \text{T}\ E'$ | $\text{first}(E') = \{\texttt{+}, \epsilon\}$ | $\text{follow}(E') = \{\texttt{\#}\}$ |
| 3. $E' ::= \epsilon$ | $\text{first}(T) = \{\texttt{(}, \texttt{id}\}$ | $\text{follow}(T) = \{\texttt{+}, \texttt{\#}\}$ |
| 4. $T ::= F\ T'$ | $\text{first}(T') = \{\texttt{*}, \epsilon\}$ | $\text{follow}(T') = \{\texttt{+}, \texttt{\#}\}$ |
| 5. $T' ::= \texttt{*}\ F\ T'$ | $\text{first}(F) = \{\texttt{(}, \texttt{id}\}$ | $\text{follow}(F) = \{\texttt{+}, \texttt{*}, \texttt{\#}\}$ |
| 6. $T' ::= \epsilon$ | | |
| 7. $F ::= \texttt{(}E\texttt{)}$ | | |
| 8. $F ::= \texttt{id}$ | | |

From rule 4, follow($F$) contains first($T'$) - $\{\epsilon\} = \{\texttt{*}\}$, and because first($T'$) contains $\epsilon$, it also contains follow($T$); also, follow($T'$) contains follow($T$); so, we have

$$\text{follow}(T') = \{\texttt{+}, \texttt{\#}\}$$
$$\text{follow}(F) = \{\texttt{+}, \texttt{*}, \texttt{\#}\}$$

Rules 5 and 6 give us nothing new (for the same reasons rules 2 and 3 did not)

Rule 7, $F ::= \texttt{(}E\texttt{)}$, adds ) to follow($E$), so follow($E$) $= \{\texttt{)}, \texttt{\#}\}$

## Again, consider our example grammar

| | | |
|---|---|---|
| 1. $E ::= T\ E'$ | $\text{first}(E) = \{\texttt{(}, \texttt{id}\}$ | $\text{follow}(E) = \{\texttt{)}, \texttt{\#}\}$ |
| 2. $E' ::= \texttt{+}\ \text{T}\ E'$ | $\text{first}(E') = \{\texttt{+}, \epsilon\}$ | $\text{follow}(E') = \{\texttt{\#}\}$ |
| 3. $E' ::= \epsilon$ | $\text{first}(T) = \{\texttt{(}, \texttt{id}\}$ | $\text{follow}(T) = \{\texttt{+}, \texttt{\#}\}$ |
| 4. $T ::= F\ T'$ | $\text{first}(T') = \{\texttt{*}, \epsilon\}$ | $\text{follow}(T') = \{\texttt{+}, \texttt{\#}\}$ |
| 5. $T' ::= \texttt{*}\ F\ T'$ | $\text{first}(F) = \{\texttt{(}, \texttt{id}\}$ | $\text{follow}(F) = \{\texttt{+}, \texttt{*}, \texttt{\#}\}$ |
| 6. $T' ::= \epsilon$ | | |
| 7. $F ::= \texttt{(}E\texttt{)}$ | | |
| 8. $F ::= \texttt{id}$ | | |

From rule 4, follow($F$) contains first($T'$) - $\{\epsilon\} = \{\texttt{*}\}$, and because first($T'$) contains $\epsilon$, it also contains follow($T$); also, follow($T'$) contains follow($T$); so, we have

$$\text{follow}(T') = \{\texttt{+}, \texttt{\#}\}$$
$$\text{follow}(F) = \{\texttt{+}, \texttt{*}, \texttt{\#}\}$$

Rules 5 and 6 give us nothing new (for the same reasons rules 2 and 3 did not)

Rule 7, $F ::= \texttt{(}E\texttt{)}$, adds ) to follow($E$), so follow($E$) $= \{\texttt{)}, \texttt{\#}\}$

Rule 8 gives us nothing

## Again, consider our example grammar

| | | |
|---|---|---|
| 1. $E ::= T\ E'$ | $\text{first}(E) = \{\texttt{(}, \texttt{id}\}$ | $\text{follow}(E) = \{\texttt{)}, \texttt{\#}\}$ |
| 2. $E' ::= \texttt{+}\ \text{T}\ E'$ | $\text{first}(E') = \{\texttt{+}, \epsilon\}$ | $\text{follow}(E') = \{\texttt{\#}\}$ |
| 3. $E' ::= \epsilon$ | $\text{first}(T) = \{\texttt{(}, \texttt{id}\}$ | $\text{follow}(T) = \{\texttt{+}, \texttt{\#}\}$ |
| 4. $T ::= F\ T'$ | $\text{first}(T') = \{\texttt{*}, \epsilon\}$ | $\text{follow}(T') = \{\texttt{+}, \texttt{\#}\}$ |
| 5. $T' ::= \texttt{*}\ F\ T'$ | $\text{first}(F) = \{\texttt{(}, \texttt{id}\}$ | $\text{follow}(F) = \{\texttt{+}, \texttt{*}, \texttt{\#}\}$ |
| 6. $T' ::= \epsilon$ | | |
| 7. $F ::= \texttt{(}E\texttt{)}$ | | |
| 8. $F ::= \texttt{id}$ | | |

From rule 4, follow($F$) contains first($T'$) - $\{\epsilon\} = \{\texttt{*}\}$, and because first($T'$) contains $\epsilon$, it also contains follow($T$); also, follow($T'$) contains follow($T$); so, we have

$$\text{follow}(T') = \{\texttt{+}, \texttt{\#}\}$$
$$\text{follow}(F) = \{\texttt{+}, \texttt{*}, \texttt{\#}\}$$

Rules 5 and 6 give us nothing new (for the same reasons rules 2 and 3 did not)

Rule 7, $F ::= \texttt{(}E\texttt{)}$, adds ) to follow($E$), so follow($E$) $= \{\texttt{)}, \texttt{\#}\}$

Rule 8 gives us nothing

We terminated round 1 of step 3.

## Again, consider our example grammar

| | | |
|---|---|---|
| 1. $E ::= T\ E'$ | $\text{first}(E) = \{\texttt{(}, \texttt{id}\}$ | $\text{follow}(E) = \{\texttt{)}, \texttt{\#}\}$ |
| 2. $E' ::= \texttt{+}\ \text{T}\ E'$ | $\text{first}(E') = \{\texttt{+}, \epsilon\}$ | $\text{follow}(E') = \{\texttt{\#}\}$ |
| 3. $E' ::= \epsilon$ | $\text{first}(T) = \{\texttt{(}, \texttt{id}\}$ | $\text{follow}(T) = \{\texttt{+}, \texttt{\#}\}$ |
| 4. $T ::= F\ T'$ | $\text{first}(T') = \{\texttt{*}, \epsilon\}$ | $\text{follow}(T') = \{\texttt{+}, \texttt{\#}\}$ |
| 5. $T' ::= \texttt{*}\ F\ T'$ | $\text{first}(F) = \{\texttt{(}, \texttt{id}\}$ | $\text{follow}(F) = \{\texttt{+}, \texttt{*}, \texttt{\#}\}$ |
| 6. $T' ::= \epsilon$ | | |
| 7. $F ::= \texttt{(}E\texttt{)}$ | | |
| 8. $F ::= \texttt{id}$ | | |

In round 2 of step 3, the ) that was added to follow($E$) trickles down into other follow sets

From rule 1, because first($E'$) contains $\epsilon$, follow($T$) contains follow($E$); also, follow($E'$) contains follow($E$); so, we have

follow($E'$) = {), #}
follow($T$) = {+, ), #}

## Again, consider our example grammar

| | | |
|---|---|---|
| 1. $E ::= T\ E'$ | $\text{first}(E) = \{\texttt{(}, \texttt{id}\}$ | $\text{follow}(E) = \{\texttt{)}, \texttt{\#}\}$ |
| 2. $E' ::= \texttt{+}\ \text{T}\ E'$ | $\text{first}(E') = \{\texttt{+}, \epsilon\}$ | $\text{follow}(E') = \{\texttt{\#}\}$ |
| 3. $E' ::= \epsilon$ | $\text{first}(T) = \{\texttt{(}, \texttt{id}\}$ | $\text{follow}(T) = \{\texttt{+}, \texttt{\#}\}$ |
| 4. $T ::= F\ T'$ | $\text{first}(T') = \{\texttt{*}, \epsilon\}$ | $\text{follow}(T') = \{\texttt{+}, \texttt{\#}\}$ |
| 5. $T' ::= \texttt{*}\ F\ T'$ | $\text{first}(F) = \{\texttt{(}, \texttt{id}\}$ | $\text{follow}(F) = \{\texttt{+}, \texttt{*}, \texttt{\#}\}$ |
| 6. $T' ::= \epsilon$ | | |
| 7. $F ::= \texttt{(}E\texttt{)}$ | | |
| 8. $F ::= \texttt{id}$ | | |

In round 2 of step 3, the ) that was added to follow($E$) trickles down into other follow sets

From rule 1, because first($E'$) contains $\epsilon$, follow($T$) contains follow($E$); also, follow($E'$) contains follow($E$); so, we have

follow($E'$) = {), #}
follow($T$) = {+, ), #}

## Again, consider our example grammar

| | | |
|---|---|---|
| 1. $E ::= T\ E'$ | $\text{first}(E) = \{\texttt{(}, \texttt{id}\}$ | $\text{follow}(E) = \{\texttt{)}, \texttt{\#}\}$ |
| 2. $E' ::= \texttt{+}\ \text{T}\ E'$ | $\text{first}(E') = \{\texttt{+}, \epsilon\}$ | $\text{follow}(E') = \{\texttt{)}, \texttt{\#}\}$ |
| 3. $E' ::= \epsilon$ | $\text{first}(T) = \{\texttt{(}, \texttt{id}\}$ | $\text{follow}(T) = \{\texttt{+}, \texttt{)}, \texttt{\#}\}$ |
| 4. $T ::= F\ T'$ | $\text{first}(T') = \{\texttt{*}, \epsilon\}$ | $\text{follow}(T') = \{\texttt{+}, \texttt{\#}\}$ |
| 5. $T' ::= \texttt{*}\ F\ T'$ | $\text{first}(F) = \{\texttt{(}, \texttt{id}\}$ | $\text{follow}(F) = \{\texttt{+}, \texttt{*}, \texttt{\#}\}$ |
| 6. $T' ::= \epsilon$ | | |
| 7. $F ::= \texttt{(}E\texttt{)}$ | | |
| 8. $F ::= \texttt{id}$ | | |

In round 2 of step 3, the ) that was added to follow($E$) trickles down into other follow sets

From rule 1, because first($E'$) contains $\epsilon$, follow($T$) contains follow($E$); also, follow($E'$) contains follow($E$); so, we have

follow($E'$) $= \{\texttt{)}, \texttt{\#}\}$
follow($T$) $= \{\texttt{+}, \texttt{)}, \texttt{\#}\}$

## Again, consider our example grammar

| | | |
|---|---|---|
| 1. $E ::= T\ E'$ | first$(E) = \{$(, id$\}$ | follow$(E) = \{$), #$\}$ |
| 2. $E' ::=$ + T $E'$ | first$(E') = \{$+, $\epsilon\}$ | follow$(E') = \{$), #$\}$ |
| 3. $E' ::= \epsilon$ | first$(T) = \{$(, id$\}$ | follow$(T) = \{$+, ), #$\}$ |
| 4. $T ::= F\ T'$ | first$(T') = \{$*, $\epsilon\}$ | follow$(T') = \{$+, #$\}$ |
| 5. $T' ::=$ * $F\ T'$ | first$(F) = \{$(, id$\}$ | follow$(F) = \{$+, *, #$\}$ |
| 6. $T' ::= \epsilon$ | | |
| 7. $F ::=$ ($E$) | | |
| 8. $F ::=$ id | | |

In round 2 of step 3, the ) that was added to follow($E$) trickles down into other follow sets

From rule 1, because first($E'$) contains $\epsilon$, follow($T$) contains follow($E$); also, follow($E'$) contains follow($E$); so, we have

follow($E'$) = {), #}
follow($T$) = {+, ), #}

From rule 4, because first($T'$) contains $\epsilon$, follow($F$) contains follow($T$); also, follow($T'$) contains follow($T$); so, we have

follow($T'$) = {+, ), #}
follow($F$) = {+, *, ), #}

## Again, consider our example grammar

| | | |
|---|---|---|
| 1. $E ::= T\ E'$ | first$(E) = \{$(, id$\}$ | follow$(E) = \{$), #$\}$ |
| 2. $E' ::=$ + T $E'$ | first$(E') = \{$+, $\epsilon\}$ | follow$(E') = \{$), #$\}$ |
| 3. $E' ::= \epsilon$ | first$(T) = \{$(, id$\}$ | follow$(T) = \{$+, ), #$\}$ |
| 4. $T ::= F\ T'$ | first$(T') = \{$*, $\epsilon\}$ | follow$(T') = \{$+, ), #$\}$ |
| 5. $T' ::=$ * $F\ T'$ | first$(F) = \{$(, id$\}$ | follow$(F) = \{$+, *, ), #$\}$ |
| 6. $T' ::= \epsilon$ | | |
| 7. $F ::=$ ($E$) | | |
| 8. $F ::=$ id | | |

In round 2 of step 3, the ) that was added to follow($E$) trickles down into other follow sets

From rule 1, because first($E'$) contains $\epsilon$, follow($T$) contains follow($E$); also, follow($E'$) contains follow($E$); so, we have

follow($E'$) = $\{$), #$\}$
follow($T$) = $\{$+, ), #$\}$

From rule 4, because first($T'$) contains $\epsilon$, follow($F$) contains follow($T$); also, follow($T'$) contains follow($T$); so, we have

follow($T'$) = $\{$+, ), #$\}$
follow($F$) = $\{$+, *, ), #$\}$

## Again, consider our example grammar

| | | |
|---|---|---|
| 1. $E ::= T\ E'$ | first$(E) = \{$(, id$\}$ | follow$(E) = \{$), #$\}$ |
| 2. $E' ::=$ + T $E'$ | first$(E') = \{$+, $\epsilon\}$ | follow$(E') = \{$), #$\}$ |
| 3. $E' ::= \epsilon$ | first$(T) = \{$(, id$\}$ | follow$(T) = \{$+, ), #$\}$ |
| 4. $T ::= F\ T'$ | first$(T') = \{$*, $\epsilon\}$ | follow$(T') = \{$+, ), #$\}$ |
| 5. $T' ::=$ * $F\ T'$ | first$(F) = \{$(, id$\}$ | follow$(F) = \{$+, *, ), #$\}$ |
| 6. $T' ::= \epsilon$ | | |
| 7. $F ::=$ ($E$) | | |
| 8. $F ::=$ id | | |

In round 2 of step 3, the ) that was added to follow($E$) trickles down into other follow sets

From rule 1, because first($E'$) contains $\epsilon$, follow($T$) contains follow($E$); also, follow($E'$) contains follow($E$); so, we have

follow($E'$) = $\{$), #$\}$
follow($T$) = $\{$+, ), #$\}$

From rule 4, because first($T'$) contains $\epsilon$, follow($F$) contains follow($T$); also, follow($T'$) contains follow($T$); so, we have

follow($T'$) = $\{$+, ), #$\}$
follow($F$) = $\{$+, *, ), #$\}$

We terminated round 2 of step 3.

## Again, consider our example grammar

| | | |
|---|---|---|
| 1. $E ::= T\ E'$ | first$(E) = \{$(, id$\}$ | follow$(E) = \{$), #$\}$ |
| 2. $E' ::=$ + T $E'$ | first$(E') = \{$+, $\epsilon\}$ | follow$(E') = \{$), #$\}$ |
| 3. $E' ::= \epsilon$ | first$(T) = \{$(, id$\}$ | follow$(T) = \{$+, ), #$\}$ |
| 4. $T ::= F\ T'$ | first$(T') = \{$*, $\epsilon\}$ | follow$(T') = \{$+, ), #$\}$ |
| 5. $T' ::=$ * $F\ T'$ | first$(F) = \{$(, id$\}$ | follow$(F) = \{$+, *, ), #$\}$ |
| 6. $T' ::= \epsilon$ | | |
| 7. $F ::=$ ($E$) | | |
| 8. $F ::=$ id | | |

In round 2 of step 3, the ) that was added to follow($E$) trickles down into other follow sets

From rule 1, because first($E'$) contains $\epsilon$, follow($T$) contains follow($E$); also, follow($E'$) contains follow($E$); so, we have

follow($E'$) = $\{$), #$\}$
follow($T$) = $\{$+, ), #$\}$

From rule 4, because first($T'$) contains $\epsilon$, follow($F$) contains follow($T$); also, follow($T'$) contains follow($T$); so, we have

follow($T'$) = $\{$+, ), #$\}$
follow($F$) = $\{$+, *, ), #$\}$

We terminated round 2 of step 3.

Round 3 of step 3 does not introduce any change, hence we are done.

## Let's put all together: Construct an LL(1) Parse Table

| | + | * | ( | ) | id | # |
|---|---|---|---|---|---|---|
| E | | | 1 | | 1 | |
| E' | 2 | | | 3 | | 3 |
| T | | | 4 | | 4 | |
| T' | 6 | 5 | | 6 | | 6 |
| F | | | 7 | | 8 | |

## Let's put all together: Construct an LL(1) Parse Table

Recall, assuming both $\alpha$ and $\beta$ are (possibly empty) strings of terminals and non-terminals, table$[Y, a] = i$, where rule $i$ is $Y ::= X_1 X_2 \ldots X_n$, if either

1. $X_1 X_2 \ldots X_n \overset{*}{\Rightarrow} a\alpha$, or
2. $X_1 X_2 \ldots X_n \overset{*}{\Rightarrow} \epsilon$, and $S\# \overset{*}{\Rightarrow} \alpha Y a \beta$, (i.e., $a$ can follow $Y$ in a derivation).

| | + | * | ( | ) | id | # |
|---|---|---|---|---|---|---|
| E | | | 1 | | 1 | |
| E' | 2 | | | 3 | | 3 |
| T | | | 4 | | 4 | |
| T' | 6 | 5 | | 6 | | 6 |
| F | | | 7 | | 8 | |

## Let's put all together: Construct an LL(1) Parse Table

Recall, assuming both $\alpha$ and $\beta$ are (possibly empty) strings of terminals and non-terminals, table$[Y, a] = i$, where rule $i$ is $Y ::= X_1 X_2 \ldots X_n$, if either

1. $X_1 X_2 \ldots X_n \overset{*}{\Rightarrow} a\alpha$, or
2. $X_1 X_2 \ldots X_n \overset{*}{\Rightarrow} \epsilon$, and $S\# \overset{*}{\Rightarrow} \alpha Y a \beta$, (i.e., $a$ can follow $Y$ in a derivation).

| | + | * | ( | ) | id | # |
|---|---|---|---|---|---|---|
| E | | | | | | |
| E' | | | | | | |
| T | | | | | | |
| T' | | | | | | |
| F | | | | | | |

## Let's put all together: Construct an LL(1) Parse Table

Recall, assuming both $\alpha$ and $\beta$ are (possibly empty) strings of terminals and non-terminals, table$[Y, a] = i$, where rule $i$ is $Y ::= X_1 X_2 \ldots X_n$, if either

1. $X_1 X_2 \ldots X_n \overset{*}{\Rightarrow} a\alpha$, or
2. $X_1 X_2 \ldots X_n \overset{*}{\Rightarrow} \epsilon$, and $S\# \overset{*}{\Rightarrow} \alpha Y a \beta$, (i.e., $a$ can follow $Y$ in a derivation).

This definition, together with the functions first and follow, suggests the following algorithm for constructing the LL(1) parse table

**Algorithm** Construct an LL(1) Parse Table for a Grammar $G = (N, T, S, P)$

**Require:** a context-free grammar $G = (N, T, S, P)$
**Ensure:** LL(1) Parse Table for $G$

```
for each non-terminal Y ∈ N do
    for each rule (Y ::= X1 X2 ... Xn) ∈ P with number i do
        for each terminal a ∈ first(X1 X2 ... Xn) − {ε} do
            table[Y, a] ← i
            if first(X1 X2 ... Xn) contains ε then
                for each terminal a (or #) in follow(Y) do
                    table[Y, a] ← i
                end for
            end if
        end for
    end for
end for
```

## Let us construct a parse table for our example grammar

| | | |
|---|---|---|
| 1. $E ::= T\ E'$ | $\text{first}(E) = \{\texttt{(}, \texttt{id}\}$ | $\text{follow}(E) = \{\texttt{)}, \texttt{\#}\}$ |
| 2. $E' ::= \texttt{+}\ \text{T}\ E'$ | $\text{first}(E') = \{\texttt{+}, \epsilon\}$ | $\text{follow}(E') = \{\texttt{)}, \texttt{\#}\}$ |
| 3. $E' ::= \epsilon$ | $\text{first}(T) = \{\texttt{(}, \texttt{id}\}$ | $\text{follow}(T) = \{\texttt{+}, \texttt{)}, \texttt{\#}\}$ |
| 4. $T ::= F\ T'$ | $\text{first}(T') = \{\texttt{*}, \epsilon\}$ | $\text{follow}(T') = \{\texttt{+}, \texttt{)}, \texttt{\#}\}$ |
| 5. $T' ::= \texttt{*}\ F\ T'$ | $\text{first}(F) = \{\texttt{(}, \texttt{id}\}$ | $\text{follow}(F) = \{\texttt{+}, \texttt{*}, \texttt{)}, \texttt{\#}\}$ |
| 6. $T' ::= \epsilon$ | | |
| 7. $F ::= \texttt{(}E\texttt{)}$ | | |
| 8. $F ::= \texttt{id}$ | | |

For the non-terminal $E$, we consider rule 1, where first$(T\ E') =$ first$(T) = \{\texttt{(},\texttt{id}\}$. We get

$\text{table}[E, \texttt{(}] = 1$
$\text{table}[E, \texttt{id}] = 1$

Because first$(T\ E')$ does not contain $\epsilon$, we do not need to consider follow$(E)$

For the non-terminal $E'$, we first consider rule 2, where first$(\texttt{+}TE') =$ first$(\texttt{+}) = \{\texttt{+}\}$. We get:

$\text{table}[E', \texttt{+}] = 2$

Then we consider rule 3: $E' ::= \epsilon$. It is applicable for symbols in follow$(E') = \{\texttt{)}, \texttt{\#}\}$:

$\text{table}[E', \texttt{)}] = 3$
$\text{table}[E', \texttt{\#}] = 3$

## Let us construct a parse table for our example grammar

| | | |
|---|---|---|
| 1. $E ::= T\ E'$ | $\text{first}(E) = \{\texttt{(}, \texttt{id}\}$ | $\text{follow}(E) = \{\texttt{)}, \texttt{\#}\}$ |
| 2. $E' ::= \texttt{+}\ \text{T}\ E'$ | $\text{first}(E') = \{\texttt{+}, \epsilon\}$ | $\text{follow}(E') = \{\texttt{)}, \texttt{\#}\}$ |
| 3. $E' ::= \epsilon$ | $\text{first}(T) = \{\texttt{(}, \texttt{id}\}$ | $\text{follow}(T) = \{\texttt{+}, \texttt{)}, \texttt{\#}\}$ |
| 4. $T ::= F\ T'$ | $\text{first}(T') = \{\texttt{*}, \epsilon\}$ | $\text{follow}(T') = \{\texttt{+}, \texttt{)}, \texttt{\#}\}$ |
| 5. $T' ::= \texttt{*}\ F\ T'$ | $\text{first}(F) = \{\texttt{(}, \texttt{id}\}$ | $\text{follow}(F) = \{\texttt{+}, \texttt{*}, \texttt{)}, \texttt{\#}\}$ |
| 6. $T' ::= \epsilon$ | | |
| 7. $F ::= \texttt{(}E\texttt{)}$ | | |
| 8. $F ::= \texttt{id}$ | | |

For the non-terminal $E$, we consider rule 1, where first($T\ E'$) = first($T$) = $\{\texttt{(},\texttt{id}\}$. We get

$\text{table}[E, \texttt{(}] = 1$
$\text{table}[E, \texttt{id}] = 1$

Because first($T\ E'$) does not contain $\epsilon$, we do not need to consider follow($E$)

For the non-terminal $E'$, we first consider rule 2, where first($\texttt{+}TE'$) = first($\texttt{+}$) = $\{\texttt{+}\}$. We get:

$\text{table}[E', \texttt{+}] = 2$

Then we consider rule 3: $E' ::= \epsilon$. It is applicable for symbols in follow($E'$) = $\{\texttt{)}, \texttt{\#}\}$:

$\text{table}[E', \texttt{)}] = 3$
$\text{table}[E', \texttt{\#}] = 3$

## Let us construct a parse table for our example grammar

| | | |
|---|---|---|
| 1. $E ::= T\ E'$ | $\text{first}(E) = \{\texttt{(}, \texttt{id}\}$ | $\text{follow}(E) = \{\texttt{)}, \texttt{\#}\}$ |
| 2. $E' ::= \texttt{+}\ \text{T}\ E'$ | $\text{first}(E') = \{\texttt{+}, \epsilon\}$ | $\text{follow}(E') = \{\texttt{)}, \texttt{\#}\}$ |
| 3. $E' ::= \epsilon$ | $\text{first}(T) = \{\texttt{(}, \texttt{id}\}$ | $\text{follow}(T) = \{\texttt{+}, \texttt{)}, \texttt{\#}\}$ |
| 4. $T ::= F\ T'$ | $\text{first}(T') = \{\texttt{*}, \epsilon\}$ | $\text{follow}(T') = \{\texttt{+}, \texttt{)}, \texttt{\#}\}$ |
| 5. $T' ::= \texttt{*}\ F\ T'$ | $\text{first}(F) = \{\texttt{(}, \texttt{id}\}$ | $\text{follow}(F) = \{\texttt{+}, \texttt{*}, \texttt{)}, \texttt{\#}\}$ |
| 6. $T' ::= \epsilon$ | | |
| 7. $F ::= \texttt{(}E\texttt{)}$ | | |
| 8. $F ::= \texttt{id}$ | | |

For the non-terminal $E$, we consider rule 1, where first($T\ E'$) = first($T$) = $\{\texttt{(},\texttt{id}\}$. We get

$\text{table}[E, \texttt{(}] = 1$
$\text{table}[E, \texttt{id}] = 1$

Because first($T\ E'$) does not contain $\epsilon$, we do not need to consider follow($E$)

For the non-terminal $E'$, we first consider rule 2, where first($\texttt{+}TE'$) = first($\texttt{+}$) = $\{\texttt{+}\}$. We get:

$\text{table}[E', \texttt{+}] = 2$

Then we consider rule 3: $E' ::= \epsilon$. It is applicable for symbols in follow($E'$) = $\{\texttt{)}, \texttt{\#}\}$:

$\text{table}[E', \texttt{)}] = 3$
$\text{table}[E', \texttt{\#}] = 3$

## Let us construct a parse table for our example grammar

| | | |
|---|---|---|
| 1. $E ::= T\ E'$ | $\text{first}(E) = \{\texttt{(}, \texttt{id}\}$ | $\text{follow}(E) = \{\texttt{)}, \texttt{\#}\}$ |
| 2. $E' ::= \texttt{+}\ \text{T}\ E'$ | $\text{first}(E') = \{\texttt{+}, \epsilon\}$ | $\text{follow}(E') = \{\texttt{)}, \texttt{\#}\}$ |
| 3. $E' ::= \epsilon$ | $\text{first}(T) = \{\texttt{(}, \texttt{id}\}$ | $\text{follow}(T) = \{\texttt{+}, \texttt{)}, \texttt{\#}\}$ |
| 4. $T ::= F\ T'$ | $\text{first}(T') = \{\texttt{*}, \epsilon\}$ | $\text{follow}(T') = \{\texttt{+}, \texttt{)}, \texttt{\#}\}$ |
| 5. $T' ::= \texttt{*}\ F\ T'$ | $\text{first}(F) = \{\texttt{(}, \texttt{id}\}$ | $\text{follow}(F) = \{\texttt{+}, \texttt{*}, \texttt{)}, \texttt{\#}\}$ |
| 6. $T' ::= \epsilon$ | | |
| 7. $F ::= \texttt{(}E\texttt{)}$ | | |
| 8. $F ::= \texttt{id}$ | | |

For the non-terminal $E$, we consider rule 1, where first($T\ E'$) = first($T$) = $\{\texttt{(},\texttt{id}\}$. We get

$$\text{table}[E, \texttt{(}] = 1$$
$$\text{table}[E, \texttt{id}] = 1$$

Because first($T\ E'$) does not contain $\epsilon$, we do not need to consider follow($E$)

For the non-terminal $E'$, we first consider rule 2, where first($\texttt{+}TE'$) = first($\texttt{+}$) = $\{\texttt{+}\}$. We get:

$$\text{table}[E', \texttt{+}] = 2$$

Then we consider rule 3: $E' ::= \epsilon$. It is applicable for symbols in follow($E'$) = $\{\texttt{)}, \texttt{\#}\}$:

$$\text{table}[E', \texttt{)}] = 3$$
$$\text{table}[E', \texttt{\#}] = 3$$

## Let us construct a parse table for our example grammar

| | | |
|---|---|---|
| 1. $E ::= T\ E'$ | $\text{first}(E) = \{\texttt{(}, \texttt{id}\}$ | $\text{follow}(E) = \{\texttt{)}, \texttt{\#}\}$ |
| 2. $E' ::= \texttt{+}\ \text{T}\ E'$ | $\text{first}(E') = \{\texttt{+}, \epsilon\}$ | $\text{follow}(E') = \{\texttt{)}, \texttt{\#}\}$ |
| 3. $E' ::= \epsilon$ | $\text{first}(T) = \{\texttt{(}, \texttt{id}\}$ | $\text{follow}(T) = \{\texttt{+}, \texttt{)}, \texttt{\#}\}$ |
| 4. $T ::= F\ T'$ | $\text{first}(T') = \{\texttt{*}, \epsilon\}$ | $\text{follow}(T') = \{\texttt{+}, \texttt{)}, \texttt{\#}\}$ |
| 5. $T' ::= \texttt{*}\ F\ T'$ | $\text{first}(F) = \{\texttt{(}, \texttt{id}\}$ | $\text{follow}(F) = \{\texttt{+}, \texttt{*}, \texttt{)}, \texttt{\#}\}$ |
| 6. $T' ::= \epsilon$ | | |
| 7. $F ::= \texttt{(}E\texttt{)}$ | | |
| 8. $F ::= \texttt{id}$ | | |

For the non-terminal $E$, we consider rule 1, where first($T\ E'$) = first($T$) = $\{\texttt{(},\texttt{id}\}$. We get

$$\text{table}[E, \texttt{(}] = 1$$
$$\text{table}[E, \texttt{id}] = 1$$

Because first($T\ E'$) does not contain $\epsilon$, we do not need to consider follow($E$)

For the non-terminal $E'$, we first consider rule 2, where first($\texttt{+}TE'$) = first($\texttt{+}$) = $\{\texttt{+}\}$. We get:

$$\text{table}[E', \texttt{+}] = 2$$

Then we consider rule 3: $E' ::= \epsilon$. It is applicable for symbols in follow($E'$) = $\{\texttt{)}, \texttt{\#}\}$:

$$\text{table}[E', \texttt{)}] = 3$$
$$\text{table}[E', \texttt{\#}] = 3$$

## Let us construct a parse table for our example grammar

| | | |
|---|---|---|
| 1. $E ::= T\ E'$ | $\text{first}(E) = \{(, \texttt{id}\}$ | $\text{follow}(E) = \{), \#\}$ |
| 2. $E' ::= + \text{T}\ E'$ | $\text{first}(E') = \{+, \epsilon\}$ | $\text{follow}(E') = \{), \#\}$ |
| 3. $E' ::= \epsilon$ | $\text{first}(T) = \{(, \texttt{id}\}$ | $\text{follow}(T) = \{+, ), \#\}$ |
| 4. $T ::= F\ T'$ | $\text{first}(T') = \{*, \epsilon\}$ | $\text{follow}(T') = \{+, ), \#\}$ |
| 5. $T' ::= * F\ T'$ | $\text{first}(F) = \{(, \texttt{id}\}$ | $\text{follow}(F) = \{+, *, ), \#\}$ |
| 6. $T' ::= \epsilon$ | | |
| 7. $F ::= (E)$ | | |
| 8. $F ::= \texttt{id}$ | | |

For the non-terminal $T$, we consider rule 4, where first$(F\ T')$ = first$(F) = \{(, \texttt{id}\}$. We get

$\text{table}[T, (] = 4$
$\text{table}[T, \texttt{id}] = 4$

Because first$(F\ T')$ does not contain $\epsilon$, we do not need to consider follow$(T)$

For non-terminal $T'$, we first consider rule 5, where first$(*\ F\ T')$ = first$(*) = \{*\}$. We get

$\text{table}[T', *] = 5$

Then we consider rule 6: $T' ::= \epsilon$. It is applicable for symbols in follow$(T') = \{+, ), \#\}$:

$\text{table}[T', +] = 6$
$\text{table}[T', )] = 6$
$\text{table}[T', \#] = 6$

## Let us construct a parse table for our example grammar

| | | |
|---|---|---|
| 1. $E ::= T\ E'$ | $\text{first}(E) = \{\texttt{(}, \texttt{id}\}$ | $\text{follow}(E) = \{\texttt{)}, \texttt{\#}\}$ |
| 2. $E' ::= \texttt{+}\ \text{T}\ E'$ | $\text{first}(E') = \{\texttt{+}, \epsilon\}$ | $\text{follow}(E') = \{\texttt{)}, \texttt{\#}\}$ |
| 3. $E' ::= \epsilon$ | $\text{first}(T) = \{\texttt{(}, \texttt{id}\}$ | $\text{follow}(T) = \{\texttt{+}, \texttt{)}, \texttt{\#}\}$ |
| 4. $T ::= F\ T'$ | $\text{first}(T') = \{\texttt{*}, \epsilon\}$ | $\text{follow}(T') = \{\texttt{+}, \texttt{)}, \texttt{\#}\}$ |
| 5. $T' ::= \texttt{*}\ F\ T'$ | $\text{first}(F) = \{\texttt{(}, \texttt{id}\}$ | $\text{follow}(F) = \{\texttt{+}, \texttt{*}, \texttt{)}, \texttt{\#}\}$ |
| 6. $T' ::= \epsilon$ | | |
| 7. $F ::= \texttt{(}E\texttt{)}$ | | |
| 8. $F ::= \texttt{id}$ | | |

For the non-terminal $T$, we consider rule 4, where first($F\ T'$) = first($F$) = $\{\texttt{(}, \texttt{id}\}$. We get

$\text{table}[T, \texttt{(}] = 4$
$\text{table}[T, \texttt{id}] = 4$

Because first($F\ T'$) does not contain $\epsilon$, we do not need to consider follow($T$)

For non-terminal $T'$, we first consider rule 5, where first($\texttt{*}\ F\ T'$) = first($\texttt{*}$) = $\{\texttt{*}\}$. We get

$\text{table}[T', \texttt{*}] = 5$

Then we consider rule 6: $T' ::= \epsilon$. It is applicable for symbols in follow($T'$) = $\{\texttt{+}, \texttt{)}, \texttt{\#}\}$:

$\text{table}[T', \texttt{+}] = 6$
$\text{table}[T', \texttt{)}] = 6$
$\text{table}[T', \texttt{\#}] = 6$

## Let us construct a parse table for our example grammar

| | | |
|---|---|---|
| 1. $E ::= T\ E'$ | $\text{first}(E) = \{\texttt{(}, \texttt{id}\}$ | $\text{follow}(E) = \{\texttt{)}, \texttt{\#}\}$ |
| 2. $E' ::= \texttt{+}\ \mathrm{T}\ E'$ | $\text{first}(E') = \{\texttt{+}, \epsilon\}$ | $\text{follow}(E') = \{\texttt{)}, \texttt{\#}\}$ |
| 3. $E' ::= \epsilon$ | $\text{first}(T) = \{\texttt{(}, \texttt{id}\}$ | $\text{follow}(T) = \{\texttt{+}, \texttt{)}, \texttt{\#}\}$ |
| 4. $T ::= F\ T'$ | $\text{first}(T') = \{\texttt{*}, \epsilon\}$ | $\text{follow}(T') = \{\texttt{+}, \texttt{)}, \texttt{\#}\}$ |
| 5. $T' ::= \texttt{*}\ F\ T'$ | $\text{first}(F) = \{\texttt{(}, \texttt{id}\}$ | $\text{follow}(F) = \{\texttt{+}, \texttt{*}, \texttt{)}, \texttt{\#}\}$ |
| 6. $T' ::= \epsilon$ | | |
| 7. $F ::= \texttt{(}E\texttt{)}$ | | |
| 8. $F ::= \texttt{id}$ | | |

For the non-terminal $T$, we consider rule 4, where first$(F\ T') =$ first$(F) = \{\texttt{(}, \texttt{id}\}$. We get

$$\text{table}[T, \texttt{(}] = 4$$
$$\text{table}[T, \texttt{id}] = 4$$

Because first$(F\ T')$ does not contain $\epsilon$, we do not need to consider follow$(T)$

For non-terminal $T'$, we first consider rule 5, where first$(\texttt{*}\ F\ T') =$ first$(\texttt{*}) = \{\texttt{*}\}$. We get

$$\text{table}[T', \texttt{*}] = 5$$

Then we consider rule 6: $T' ::= \epsilon$. It is applicable for symbols in follow$(T') = \{\texttt{+}, \texttt{)}, \texttt{\#}\}$:

$$\text{table}[T', \texttt{+}] = 6$$
$$\text{table}[T', \texttt{)}] = 6$$
$$\text{table}[T', \texttt{\#}] = 6$$

## Let us construct a parse table for our example grammar

| | | |
|---|---|---|
| 1. $E ::= T\ E'$ | $\text{first}(E) = \{\texttt{(}, \texttt{id}\}$ | $\text{follow}(E) = \{\texttt{)}, \texttt{\#}\}$ |
| 2. $E' ::= \texttt{+}\ \mathrm{T}\ E'$ | $\text{first}(E') = \{\texttt{+}, \epsilon\}$ | $\text{follow}(E') = \{\texttt{)}, \texttt{\#}\}$ |
| 3. $E' ::= \epsilon$ | $\text{first}(T) = \{\texttt{(}, \texttt{id}\}$ | $\text{follow}(T) = \{\texttt{+}, \texttt{)}, \texttt{\#}\}$ |
| 4. $T ::= F\ T'$ | $\text{first}(T') = \{\texttt{*}, \epsilon\}$ | $\text{follow}(T') = \{\texttt{+}, \texttt{)}, \texttt{\#}\}$ |
| 5. $T' ::= \texttt{*}\ F\ T'$ | $\text{first}(F) = \{\texttt{(}, \texttt{id}\}$ | $\text{follow}(F) = \{\texttt{+}, \texttt{*}, \texttt{)}, \texttt{\#}\}$ |
| 6. $T' ::= \epsilon$ | | |
| 7. $F ::= \texttt{(}E\texttt{)}$ | | |
| 8. $F ::= \texttt{id}$ | | |

For the non-terminal $T$, we consider rule 4, where first($F\ T'$) = first($F$) = $\{\texttt{(}, \texttt{id}\}$. We get

$$\text{table}[T, \texttt{(}] = 4$$
$$\text{table}[T, \texttt{id}] = 4$$

Because first($F\ T'$) does not contain $\epsilon$, we do not need to consider follow($T$)

For non-terminal $T'$, we first consider rule 5, where first($\texttt{*}\ F\ T'$) = first($\texttt{*}$) = $\{\texttt{*}\}$. We get

$$\text{table}[T', \texttt{*}] = 5$$

Then we consider rule 6: $T' ::= \epsilon$. It is applicable for symbols in follow($T'$) = $\{\texttt{+}, \texttt{)}, \texttt{\#}\}$:

$$\text{table}[T', \texttt{+}] = 6$$
$$\text{table}[T', \texttt{)}] = 6$$
$$\text{table}[T', \texttt{\#}] = 6$$

## Let us construct a parse table for our example grammar

| | | |
|---|---|---|
| 1. $E ::= T\ E'$ | $\text{first}(E) = \{\texttt{(}, \texttt{id}\}$ | $\text{follow}(E) = \{\texttt{)}, \texttt{\#}\}$ |
| 2. $E' ::= \texttt{+}\ \text{T}\ E'$ | $\text{first}(E') = \{\texttt{+}, \epsilon\}$ | $\text{follow}(E') = \{\texttt{)}, \texttt{\#}\}$ |
| 3. $E' ::= \epsilon$ | $\text{first}(T) = \{\texttt{(}, \texttt{id}\}$ | $\text{follow}(T) = \{\texttt{+}, \texttt{)}, \texttt{\#}\}$ |
| 4. $T ::= F\ T'$ | $\text{first}(T') = \{\texttt{*}, \epsilon\}$ | $\text{follow}(T') = \{\texttt{+}, \texttt{)}, \texttt{\#}\}$ |
| 5. $T' ::= \texttt{*}\ F\ T'$ | $\text{first}(F) = \{\texttt{(}, \texttt{id}\}$ | $\text{follow}(F) = \{\texttt{+}, \texttt{*}, \texttt{)}, \texttt{\#}\}$ |
| 6. $T' ::= \epsilon$ | | |
| 7. $F ::= \texttt{(}E\texttt{)}$ | | |
| 8. $F ::= \texttt{id}$ | | |

For non-terminal $F$ we first consider rule 7. Since first($\texttt{(}E\texttt{)}$) = first($\texttt{(}$) = $\{\texttt{(}\}$, we have

$$\text{table}[F, \texttt{(}] = 7$$

Then we consider rule 8. Since first($\texttt{id}$) = $\{\texttt{id}\}$, we have table$[F, \texttt{id}] = 8$

## Let us construct a parse table for our example grammar

| | | |
|---|---|---|
| 1. $E ::= T\ E'$ | $\text{first}(E) = \{\texttt{(}, \texttt{id}\}$ | $\text{follow}(E) = \{\texttt{)}, \texttt{\#}\}$ |
| 2. $E' ::= \texttt{+}\ \text{T}\ E'$ | $\text{first}(E') = \{\texttt{+}, \epsilon\}$ | $\text{follow}(E') = \{\texttt{)}, \texttt{\#}\}$ |
| 3. $E' ::= \epsilon$ | $\text{first}(T) = \{\texttt{(}, \texttt{id}\}$ | $\text{follow}(T) = \{\texttt{+}, \texttt{)}, \texttt{\#}\}$ |
| 4. $T ::= F\ T'$ | $\text{first}(T') = \{\texttt{*}, \epsilon\}$ | $\text{follow}(T') = \{\texttt{+}, \texttt{)}, \texttt{\#}\}$ |
| 5. $T' ::= \texttt{*}\ F\ T'$ | $\text{first}(F) = \{\texttt{(}, \texttt{id}\}$ | $\text{follow}(F) = \{\texttt{+}, \texttt{*}, \texttt{)}, \texttt{\#}\}$ |
| 6. $T' ::= \epsilon$ | | |
| 7. $F ::= \texttt{(}E\texttt{)}$ | | |
| 8. $F ::= \texttt{id}$ | | |

For non-terminal $F$ we first consider rule 7. Since first($\texttt{(}E\texttt{)}$) = first($\texttt{(}$) = $\{\texttt{(}\}$, we have

$$\text{table}[F, \texttt{(}] = 7$$

Then we consider rule 8. Since first($\texttt{id}$) = $\{\texttt{id}\}$, we have table$[F, \texttt{id}] = 8$

## Let us construct a parse table for our example grammar

1. $E$ ::= $T\ E'$
2. $E'$ ::= + T $E'$
3. $E'$ ::= $\epsilon$
4. $T$ ::= $F\ T'$
5. $T'$ ::= * $F\ T'$
6. $T'$ ::= $\epsilon$
7. $F$ ::= (E)
8. $F$ ::= id

first($E$) = {(, id}
first($E'$) = {+, $\epsilon$}
first($T$) = {(, id}
first($T'$) = {*, $\epsilon$}
first($F$) = {(, id}

follow($E$) = {), #}
follow($E'$) = {), #}
follow($T$) = {+, ), #}
follow($T'$) = {+, ), #}
follow($F$) = {+, *, ), #}

For non-terminal $F$ we first consider rule 7. Since first((E)) = first(() = {(}, we have

table[$F$, (] = 7

Then we consider rule 8. Since first(id) = {id}, we have table[$F$, id] = 8

| | + | * | ( | ) | id | # |
|---|---|---|---|---|---|---|
| E | | | 1 | | 1 | |
| E' | 2 | | | 3 | | 3 |
| T | | | 4 | | 4 | |
| T' | 6 | 5 | | 6 | | 6 |
| F | | | 7 | | 8 | |

## Beyond LL(1) grammars: LL(k) grammars

A grammar is LL(1) if the parse table has no conflicts: no entries having more than one rule

If a grammar is shown to be LL(1) then it is unambiguous

It is possible for a grammar not to be LL(1) but LL($k$) for some $k > 1$; in principle, this would mean a table having columns for each combination of $k$ symbols

- TOO BIG EVEN FOR SIMPLE GRAMMARS!

LL(1) parser generators based on the table construction algorithm (like JavaCC)

- might allow one to specify a $k$-symbol look ahead for specific non-terminals or rules;
- these special cases can be handled specially by the parser and so need not lead to overly large (and, most likely sparse) tables

## Beyond LL(1) grammars: LL(k) grammars

A grammar is LL(1) if the parse table has no conflicts: no entries having more than one rule

If a grammar is shown to be LL(1) then it is unambiguous

It is possible for a grammar not to be LL(1) but LL($k$) for some $k > 1$; in principle, this would mean a table having columns for each combination of $k$ symbols

- TOO BIG EVEN FOR SIMPLE GRAMMARS!

LL(1) parser generators based on the table construction algorithm (like JavaCC)

- might allow one to specify a $k$-symbol look ahead for specific non-terminals or rules;
- these special cases can be handled specially by the parser and so need not lead to overly large (and, most likely sparse) tables

## Beyond LL(1) grammars: LL(k) grammars

A grammar is LL(1) if the parse table has no conflicts: no entries having more than one rule

If a grammar is shown to be LL(1) then it is unambiguous

It is possible for a grammar not to be LL(1) but LL($k$) for some $k > 1$; in principle, this would mean a table having columns for each combination of $k$ symbols

- TOO BIG EVEN FOR SIMPLE GRAMMARS!

LL(1) parser generators based on the table construction algorithm (like JavaCC)

- might allow one to specify a $k$-symbol look ahead for specific non-terminals or rules;
- these special cases can be handled specially by the parser and so need not lead to overly large (and, most likely sparse) tables

## Beyond LL(1) grammars: LL(k) grammars

A grammar is LL(1) if the parse table has no conflicts: no entries having more than one rule

If a grammar is shown to be LL(1) then it is unambiguous

It is possible for a grammar not to be LL(1) but LL($k$) for some $k > 1$; in principle, this would mean a table having columns for each combination of $k$ symbols

- TOO BIG EVEN FOR SIMPLE GRAMMARS!

LL(1) parser generators based on the table construction algorithm (like JavaCC)

- might allow one to specify a $k$-symbol look ahead for specific non-terminals or rules;
- these special cases can be handled specially by the parser and so need not lead to overly large (and, most likely sparse) tables

## Outline

1. Introduction
2. Context-free Grammars and Languages
   - Backus-Naur Form (BNF)
   - Left- and right-most derivation, & Parse tree
   - Ambiguity
3. Top-down Deterministic Parsing
   - Example: Top-down deterministic parsing of a *j*-- program
   - Parsing by Recursive Descent
   - Parsing by the LL(1) technique
4. **Transform a non-LL(1) grammar into an equivalent LL(1) one**
5. Parser Generation Using JavaCC

## Transform a non-LL(1) grammar into an equivalent LL(1) one

### Left Recursion

Not all context-free grammars are LL(1), but for many that are not, one may define equivalent grammars that are LL(1)

One class of non-LL(1) grammars are those having rules with left recursion; for example

$$\mathrm{Y} ::= \mathrm{Y}\ \alpha$$
$$\mathrm{Y} ::= \beta$$

Clearly, a grammar having these two rules is not LL(1), because, by definition, first($Y\alpha$) must include first($\beta$) making it impossible to discern which rule to apply for expanding $Y$

Introducing an extra non-terminal, an extra rule, and replacing the left recursion with right recursion easily removes the direct left recursion

$$\mathrm{Y} \ ::= \beta\ Y'$$
$$Y' ::= \alpha\ Y'$$
$$Y' ::= \epsilon$$

## Transform a non-LL(1) grammar into an equivalent LL(1) one

### Left Recursion

Not all context-free grammars are LL(1), but for many that are not, one may define equivalent grammars that are LL(1)

One class of non-LL(1) grammars are those having rules with left recursion; for example

$$\mathrm{Y} ::= \mathrm{Y}\ \alpha$$
$$\mathrm{Y} ::= \beta$$

Clearly, a grammar having these two rules is not LL(1), because, by definition, first($Y\alpha$) must include first($\beta$) making it impossible to discern which rule to apply for expanding $Y$

Introducing an extra non-terminal, an extra rule, and replacing the left recursion with right recursion easily removes the direct left recursion

$$\mathrm{Y} ::= \beta\ Y'$$
$$Y' ::= \alpha\ Y'$$
$$Y' ::= \epsilon$$

## Transform a non-LL(1) grammar into an equivalent LL(1) one

### Left Recursion

Not all context-free grammars are LL(1), but for many that are not, one may define equivalent grammars that are LL(1)

One class of non-LL(1) grammars are those having rules with left recursion; for example

$$\mathrm{Y} ::= \mathrm{Y}\ \alpha$$
$$\mathrm{Y} ::= \beta$$

Clearly, a grammar having these two rules is not LL(1), because, by definition, first($Y\alpha$) must include first($\beta$) making it impossible to discern which rule to apply for expanding $Y$

Introducing an extra non-terminal, an extra rule, and replacing the left recursion with right recursion easily removes the direct left recursion

$$\mathrm{Y} ::= \beta\ Y'$$
$$Y' ::= \alpha\ Y'$$
$$Y' ::= \epsilon$$

## Transform a non-LL(1) grammar into an equivalent LL(1) one

### Left Recursion

Not all context-free grammars are LL(1), but for many that are not, one may define equivalent grammars that are LL(1)

One class of non-LL(1) grammars are those having rules with left recursion; for example

$$\mathrm{Y} ::= \mathrm{Y}\ \alpha$$
$$\mathrm{Y} ::= \beta$$

Clearly, a grammar having these two rules is not LL(1), because, by definition, first($Y\alpha$) must include first($\beta$) making it impossible to discern which rule to apply for expanding $Y$

Introducing an extra non-terminal, an extra rule, and replacing the left recursion with right recursion easily removes the direct left recursion

$$\mathrm{Y} \ ::= \beta\ Y'$$
$$Y' ::= \alpha\ Y'$$
$$Y' ::= \epsilon$$

For example, consider the following context-free grammar

$$
\begin{aligned}
E &::= E \texttt{ + } T \\
E &::= T \\
T &::= T \texttt{ * } \mathrm{F} \\
T &::= F \\
F &::= \texttt{(} E \texttt{)} \\
F &::= \texttt{id}
\end{aligned}
$$

Since the grammar has left-recursive rules, it is not LL(1); we may apply the left-recursion removal rule to this grammar

Applying the rule to $E$ to produces

$$
\begin{aligned}
E &::= T\ E' \\
E' &::= \texttt{+}\ T\ E' \\
E' &::= \epsilon
\end{aligned}
$$

For example, consider the following context-free grammar

$$
\begin{aligned}
E &::= E \texttt{ + } T \\
E &::= T \\
T &::= T \texttt{ * } \mathrm{F} \\
T &::= F \\
F &::= \texttt{(} E \texttt{)} \\
F &::= \texttt{id}
\end{aligned}
$$

Since the grammar has left-recursive rules, it is not LL(1); we may apply the left-recursion removal rule to this grammar

Applying the rule to $E$ to produces

$$
\begin{aligned}
E &::= T\ E' \\
E' &::= \texttt{+}\ T\ E' \\
E' &::= \epsilon
\end{aligned}
$$

For example, consider the following context-free grammar

$$
\begin{aligned}
E &::= E \texttt{ + } T \\
E &::= T \\
T &::= T \texttt{ * } \mathrm{F} \\
T &::= F \\
F &::= \texttt{(} E \texttt{)} \\
F &::= \texttt{id}
\end{aligned}
$$

Since the grammar has left-recursive rules, it is not LL(1); we may apply the left-recursion removal rule to this grammar

Applying the rule to $E$ to produces

$$
\begin{aligned}
E &::= T\ E' \\
E' &::= \texttt{+}\ T\ E' \\
E' &::= \epsilon
\end{aligned}
$$

For example, consider the following context-free grammar

$$
\begin{aligned}
E &::= E \texttt{ + } T \\
E &::= T \\
T &::= T \texttt{ * } \mathrm{F} \\
T &::= F \\
F &::= \texttt{(} E \texttt{)} \\
F &::= \texttt{id}
\end{aligned}
$$

Since the grammar has left-recursive rules, it is not LL(1); we may apply the left-recursion removal rule to this grammar

Applying the rule to $E$ to produces

$$
\begin{aligned}
E &::= T\ E' \\
E' &::= \texttt{+}\ T\ E' \\
E' &::= \epsilon
\end{aligned}
$$

For example, consider the following context-free grammar

$$
\begin{aligned}
E &::= E \texttt{ + } T \\
E &::= T \\
T &::= T \texttt{ * } \mathrm{F} \\
T &::= F \\
F &::= \texttt{(} E \texttt{)} \\
F &::= \texttt{id}
\end{aligned}
$$

Since the grammar has left-recursive rules, it is not LL(1); we may apply the left-recursion removal rule to this grammar

Applying the rule to $E$ to produces

$$
\begin{aligned}
E &::= T\ E' \\
E' &::= \texttt{+}\ T\ E' \\
E' &::= \epsilon
\end{aligned}
$$

Do we have more examples of left recursion?

For example, consider the following context-free grammar

$$
\begin{aligned}
E &::= E \texttt{ + } T \\
E &::= T \\
T &::= T \texttt{ * } \mathrm{F} \\
T &::= F \\
F &::= \texttt{(} E \texttt{)} \\
F &::= \texttt{id}
\end{aligned}
$$

Since the grammar has left-recursive rules, it is not LL(1); we may apply the left-recursion removal rule to this grammar

Applying the rule to $E$ to produces

$$
\begin{aligned}
E &::= T\ E' \\
E' &::= \texttt{+}\ T\ E' \\
E' &::= \epsilon
\end{aligned}
$$

Do we have more examples of left recursion?

Applying the rule to $T$ produces ...

Applying the rule to $T$ yields

$$
\begin{aligned}
T &::= F\ T' \\
T' &::= *\ F\ T' \\
T' &::= \epsilon
\end{aligned}
$$

Giving us the LL(1) grammar

$$
\begin{aligned}
E &::= T\ E' \\
E' &::= +\ T\ E' \\
E' &::= \epsilon \\
T &::= F\ T' \\
T' &::= *\ F\ T' \\
T' &::= \epsilon \\
F &::= (E) \\
F &::= \texttt{id}
\end{aligned}
$$

Applying the rule to $T$ yields

$$
\begin{aligned}
T &::= F\ T' \\
T' &::= *\ F\ T' \\
T' &::= \epsilon
\end{aligned}
$$

Giving us the LL(1) grammar

$$
\begin{aligned}
E &::= T\ E' \\
E' &::= +\ T\ E' \\
E' &::= \epsilon \\
T &::= F\ T' \\
T' &::= *\ F\ T' \\
T' &::= \epsilon \\
F &::= (E) \\
F &::= \texttt{id}
\end{aligned}
$$

Applying the rule to $T$ yields

$$
\begin{aligned}
T &::= F\ T' \\
T' &::= \texttt{*}\ F\ T' \\
T' &::= \epsilon
\end{aligned}
$$

Giving us the LL(1) grammar

$$
\begin{aligned}
E &::= T\ E' \\
E' &::= \texttt{+}\ T\ E' \\
E' &::= \epsilon \\
T &::= F\ T' \\
T' &::= \texttt{*}\ F\ T' \\
T' &::= \epsilon \\
F &::= \texttt{(}E\texttt{)} \\
F &::= \texttt{id}
\end{aligned}
$$

Applying the rule to $T$ yields

$$
\begin{aligned}
T &::= F\ T' \\
T' &::= \texttt{*}\ F\ T' \\
T' &::= \epsilon
\end{aligned}
$$

Giving us the LL(1) grammar

$$
\begin{aligned}
E &::= T\ E' \\
E' &::= \texttt{+}\ T\ E' \\
E' &::= \epsilon \\
T &::= F\ T' \\
T' &::= \texttt{*}\ F\ T' \\
T' &::= \epsilon \\
F &::= \texttt{(}E\texttt{)} \\
F &::= \texttt{id}
\end{aligned}
$$

Have we seen this grammar already?

# Transform a non-LL(1) grammar into an equivalent LL(1) one

## Indirect Left Recursion

Much less common is the case of indirect left recursion:

$$S ::= A\texttt{a} \mid \texttt{b}$$
$$A ::= S\texttt{c} \mid \texttt{b}$$

# Transform a non-LL(1) grammar into an equivalent LL(1) one

## An algorithm for eliminating Left Recursion

**Algorithm** Left Recursion Removal for a Grammar $G = (N, T, S, P)$

**Require:** a context-free grammar $G = (N, T, S, P)$
**Ensure:** $G$ with left recursion eliminated
  Arbitrarily enumerate the non-terminals of $G : X_1, X_2, \ldots, X_n$
  **for** $i := 1$ to $n$ **do**
    **for** $j := 1$ to $i - 1$ **do**
      Replace each rule in $P$ of the form $X_i ::= X_j\alpha$ by the rules $X_i ::= \beta_1\alpha|\beta_2\alpha|\ldots|\beta_k\alpha$ where $X_j ::= \beta_1|\beta_2|\ldots|\beta_k$ are the current rules defining $X_j$
      Eliminate any direct left recursion applying the rule from the previous slides
    **end for**
  **end for**

## Transform a non-LL(1) grammar into an equivalent LL(1) one

### Left factoring

Often grammars violate the LL(1) property due to more rules with same LHS and prefix:

$$Y ::= \alpha\beta$$
$$Y ::= \alpha\gamma$$

If first($\beta$) and first($\gamma$) are disjoint, we solve the problem by adding a new non-terminal:

$$Y ::= \alpha Y'$$
$$Y' ::= \beta$$
$$Y' ::= \gamma$$

What about if?

$$\begin{aligned} S ::= & \texttt{ if } E \texttt{ do } S \\ & \mid \texttt{ if } E \texttt{ then } S \texttt{ else } S \\ & \mid \texttt{ s} \\ E ::= & \texttt{ e} \end{aligned}$$

## Transform a non-LL(1) grammar into an equivalent LL(1) one

### Left factoring

Often grammars violate the LL(1) property due to more rules with same LHS and prefix:

$$Y ::= \alpha\beta$$
$$Y ::= \alpha\gamma$$

If first($\beta$) and first($\gamma$) are disjoint, we solve the problem by adding a new non-terminal:

$$Y ::= \alpha Y'$$
$$Y' ::= \beta$$
$$Y' ::= \gamma$$

What about if?

$$\begin{aligned} S ::= & \texttt{ if } E \texttt{ do } S \\ & \mid \texttt{ if } E \texttt{ then } S \texttt{ else } S \\ & \mid \texttt{ s} \\ E ::= & \texttt{ e} \end{aligned}$$

## Transform a non-LL(1) grammar into an equivalent LL(1) one

### Left factoring

Often grammars violate the LL(1) property due to more rules with same LHS and prefix:

$$Y ::= \alpha\beta$$
$$Y ::= \alpha\gamma$$

If first($\beta$) and first($\gamma$) are disjoint, we solve the problem by adding a new non-terminal:

$$Y ::= \alpha Y'$$
$$Y' ::= \beta$$
$$Y' ::= \gamma$$

What about if?

$$\begin{aligned} S ::= & \texttt{ if } E \texttt{ do } S \\ & \mid \texttt{ if } E \texttt{ then } S \texttt{ else } S \\ & \mid \texttt{ s} \\ E ::= & \texttt{ e} \end{aligned}$$

# Transform a non-LL(1) grammar into an equivalent LL(1) one

## Left factoring

Often grammars violate the LL(1) property due to more rules with same LHS and prefix:

$$Y ::= \alpha\beta$$
$$Y ::= \alpha\gamma$$

If first($\beta$) and first($\gamma$) are disjoint, we solve the problem by adding a new non-terminal:

$$Y ::= \alpha Y'$$
$$Y' ::= \beta$$
$$Y' ::= \gamma$$

What about if?

$$\begin{aligned} S ::= & \texttt{ if } E \texttt{ do } S \\ & \mid \texttt{ if } E \texttt{ then } S \texttt{ else } S \\ & \mid \texttt{ s} \\ E ::= & \texttt{ e} \end{aligned}$$

$$\begin{aligned} S ::= & \texttt{ if } E \texttt{ do } S' \\ & \mid \texttt{ s} \\ S' ::= & \texttt{ do } S \\ & \mid \texttt{ then } S \texttt{ else } S \\ E ::= & \texttt{ e} \end{aligned}$$

## Outline

1. Introduction
2. Context-free Grammars and Languages
   - Backus-Naur Form (BNF)
   - Left- and right-most derivation, & Parse tree
   - Ambiguity
3. Top-down Deterministic Parsing
   - Example: Top-down deterministic parsing of a *j*-- program
   - Parsing by Recursive Descent
   - Parsing by the LL(1) technique
4. Transform a non-LL(1) grammar into an equivalent LL(1) one
5. Parser Generation Using JavaCC

## The structure of file j--.jj

Besides containing the regular expressions for the lexical structure for *j*--, the `j--.jj` file also contains the syntactic rules for the language

The Java code between the `PARSER_BEGIN(JavaCCParser)` and `PARSER_END(JavaCCParser)` block is copied verbatim to the generated `JavaCCParser.java` file in the `jminusminus` package

The Java code defines helper functions, which are available for use within the generated parser; some of the helpers include: `reportParserError()` for reporting errors, and `recoverFromError()` for recovering from errors

Following the block is the specification for the scanner for *j*--, and following that is the specification for the parser for *j*--

The general layout for the syntactic specification is this: we define a start symbol, which is a high level non-terminal (`compilationUnit` in case of *j*--) that references lower level non-terminals, which in turn reference the tokens defined in the lexical specification

## The structure of file j--.jj

Besides containing the regular expressions for the lexical structure for *j*--, the `j--.jj` file also contains the syntactic rules for the language

The Java code between the `PARSER_BEGIN(JavaCCParser)` and `PARSER_END(JavaCCParser)` block is copied verbatim to the generated `JavaCCParser.java` file in the `jminusminus` package

The Java code defines helper functions, which are available for use within the generated parser; some of the helpers include: `reportParserError()` for reporting errors, and `recoverFromError()` for recovering from errors

Following the block is the specification for the scanner for *j*--, and following that is the specification for the parser for *j*--

The general layout for the syntactic specification is this: we define a start symbol, which is a high level non-terminal (`compilationUnit` in case of *j*--) that references lower level non-terminals, which in turn reference the tokens defined in the lexical specification

## The structure of file j--.jj

Besides containing the regular expressions for the lexical structure for *j*--, the `j--.jj` file also contains the syntactic rules for the language

The Java code between the `PARSER_BEGIN(JavaCCParser)` and `PARSER_END(JavaCCParser)` block is copied verbatim to the generated `JavaCCParser.java` file in the `jminusminus` package

The Java code defines helper functions, which are available for use within the generated parser; some of the helpers include: `reportParserError()` for reporting errors, and `recoverFromError()` for recovering from errors

Following the block is the specification for the scanner for *j*--, and following that is the specification for the parser for *j*--

The general layout for the syntactic specification is this: we define a start symbol, which is a high level non-terminal (`compilationUnit` in case of *j*--) that references lower level non-terminals, which in turn reference the tokens defined in the lexical specification

## The structure of file j--.jj

Besides containing the regular expressions for the lexical structure for *j*--, the `j--.jj` file also contains the syntactic rules for the language

The Java code between the `PARSER_BEGIN(JavaCCParser)` and `PARSER_END(JavaCCParser)` block is copied verbatim to the generated `JavaCCParser.java` file in the `jminusminus` package

The Java code defines helper functions, which are available for use within the generated parser; some of the helpers include: `reportParserError()` for reporting errors, and `recoverFromError()` for recovering from errors

Following the block is the specification for the scanner for *j*--, and following that is the specification for the parser for *j*--

The general layout for the syntactic specification is this: we define a start symbol, which is a high level non-terminal (`compilationUnit` in case of *j*--) that references lower level non-terminals, which in turn reference the tokens defined in the lexical specification

## The structure of file j--.jj

Besides containing the regular expressions for the lexical structure for *j*--, the `j--.jj` file also contains the syntactic rules for the language

The Java code between the `PARSER_BEGIN(JavaCCParser)` and `PARSER_END(JavaCCParser)` block is copied verbatim to the generated `JavaCCParser.java` file in the `jminusminus` package

The Java code defines helper functions, which are available for use within the generated parser; some of the helpers include: `reportParserError()` for reporting errors, and `recoverFromError()` for recovering from errors

Following the block is the specification for the scanner for *j*--, and following that is the specification for the parser for *j*--

The general layout for the syntactic specification is this: we define a start symbol, which is a high level non-terminal (`compilationUnit` in case of *j*--) that references lower level non-terminals, which in turn reference the tokens defined in the lexical specification

We are allowed to use the following BNF syntax in the syntactic specification

- $[a]$ for "zero or one", or an "optional" occurrence of $a$
- $(a)*$ for "zero or more" occurrences of $a$
- $a|b$ for alternation, ie, either $a$ or $b$
- $()$ for grouping

The syntax for a non-terminal declaration (or production rule) in the input file almost resembles that of a Java method declaration; it has a return type (could be `void`), a name, can accept arguments, and has a body, which specifies the BNF rules along with any actions that we want performed as the production rule is parsed

The method declaration also has a block preceding the body, which declares any local variables used within the body

Syntactic actions, such as creating/returning an AST node, are Java code embedded within blocks

JavaCC turns the specification for each non-terminal into a Java method within the generated parser

We are allowed to use the following BNF syntax in the syntactic specification

- $[a]$ for "zero or one", or an "optional" occurrence of $a$
- $(a)*$ for "zero or more" occurrences of $a$
- $a|b$ for alternation, ie, either $a$ or $b$
- $()$ for grouping

The syntax for a non-terminal declaration (or production rule) in the input file almost resembles that of a Java method declaration; it has a return type (could be `void`), a name, can accept arguments, and has a body, which specifies the BNF rules along with any actions that we want performed as the production rule is parsed

The method declaration also has a block preceding the body, which declares any local variables used within the body

Syntactic actions, such as creating/returning an AST node, are Java code embedded within blocks

JavaCC turns the specification for each non-terminal into a Java method within the generated parser

We are allowed to use the following BNF syntax in the syntactic specification

- $[a]$ for "zero or one", or an "optional" occurrence of $a$
- $(a)*$ for "zero or more" occurrences of $a$
- $a|b$ for alternation, ie, either $a$ or $b$
- $()$ for grouping

The syntax for a non-terminal declaration (or production rule) in the input file almost resembles that of a Java method declaration; it has a return type (could be `void`), a name, can accept arguments, and has a body, which specifies the BNF rules along with any actions that we want performed as the production rule is parsed

The method declaration also has a block preceding the body, which declares any local variables used within the body

Syntactic actions, such as creating/returning an AST node, are Java code embedded within blocks

JavaCC turns the specification for each non-terminal into a Java method within the generated parser

We are allowed to use the following BNF syntax in the syntactic specification

- $[a]$ for "zero or one", or an "optional" occurrence of $a$
- $(a)*$ for "zero or more" occurrences of $a$
- $a|b$ for alternation, ie, either $a$ or $b$
- $()$ for grouping

The syntax for a non-terminal declaration (or production rule) in the input file almost resembles that of a Java method declaration; it has a return type (could be `void`), a name, can accept arguments, and has a body, which specifies the BNF rules along with any actions that we want performed as the production rule is parsed

The method declaration also has a block preceding the body, which declares any local variables used within the body

Syntactic actions, such as creating/returning an AST node, are Java code embedded within blocks

JavaCC turns the specification for each non-terminal into a Java method within the generated parser

We are allowed to use the following BNF syntax in the syntactic specification

- $[a]$ for "zero or one", or an "optional" occurrence of $a$
- $(a)*$ for "zero or more" occurrences of $a$
- $a|b$ for alternation, ie, either $a$ or $b$
- $()$ for grouping

The syntax for a non-terminal declaration (or production rule) in the input file almost resembles that of a Java method declaration; it has a return type (could be `void`), a name, can accept arguments, and has a body, which specifies the BNF rules along with any actions that we want performed as the production rule is parsed

The method declaration also has a block preceding the body, which declares any local variables used within the body

Syntactic actions, such as creating/returning an AST node, are Java code embedded within blocks

JavaCC turns the specification for each non-terminal into a Java method within the generated parser

The specification for the following rule for parsing a qualified identifier in *j--*

qualifiedIdentifier ::= <identifier> {. <identifier>}

is shown below

```
private TypeName qualifiedIdentifier(): {
    int line = 0;
    String qualifiedIdentifier = "";
}
{
    try {
        <IDENTIFIER>
        {
            line = token.beginLine;
            qualifiedIdentifier = token.image;
        }
        (
            <DOT> <IDENTIFIER>
            { qualifiedIdentifier += "." + token.image; }
        )*
    }
    catch (ParseException e) {
        recoverFromError(new int[] { SEMI, EOF }, e);
    }
    { return new TypeName(line, qualifiedIdentifier); }
}
```

The specification for the following rule for parsing a statement in *j--*

statement ::= block
| `<identifier>` : statement
| `if` parExpression statement [`else` statement]
| `while` parExpression statement
| `return` [expression] ;
| ;
| statementExpression ;

is shown below

```
private JStatement statement(): {
    int line = 0;
    JStatement statement = null;
    JExpression test = null;
    JStatement consequent = null;
    JStatement alternate = null;
    JStatement body = null;
    JExpression expr = null;
}
{
    try {
        statement = block() |
```

```
        <IF> { line = token.beginLine; }
        test = parExpression()
        consequent = statement()
        [
            LOOKAHEAD(<ELSE>)
            <ELSE> alternate = statement()
        ]
        { statement =
            new JIfStatement(line, test, consequent, alternate); } |
        <WHILE> { line = token.beginLine; }
        test = parExpression()
        body = statement()
        { statement = new JWhileStatement(line, test, body); } |
        <RETURN> { line = token.beginLine; }
        [
            expr = expression()
        ]
        <SEMI>
        { statement = new JReturnStatement(line, expr); } |
        <SEMI>
        { statement = new JEmptyStatement(line); } |
        statement = statementExpression()
        <SEMI>
    }
    catch (ParseException e) {
        recoverFromError(new int[] { SEMI, EOF }, e);
    }
    { return statement; }
}
```

## What to do if we need to lookahead

As in the case of a recursive descent parser, we cannot always decide which production rule to use in parsing a non-terminal just by looking at the current token; we have to look ahead at the next few symbols to decide

E.g., the specification for the following rule for parsing a simple unary expression in *j--*

simpleUnaryExpression ::= ! unaryExpression
| ( basicType ) unaryExpression //cast
| ( referenceType ) simpleUnaryExpression // cast
| postfixExpression

involves lookahead and is shown below

```
private JExpression simpleUnaryExpression(): {
    int line = 0;
    Type type = null;
    JExpression expr = null, unaryExpr = null, simpleUnaryExpr = null;
}
{
    try {
        <LNOT> { line = token.beginLine; }
        unaryExpr = unaryExpression()
        { expr = new JLogicalNotOp(line, unaryExpr); } |
```

## What to do if we need to lookahead

As in the case of a recursive descent parser, we cannot always decide which production rule to use in parsing a non-terminal just by looking at the current token; we have to look ahead at the next few symbols to decide

E.g., the specification for the following rule for parsing a simple unary expression in *j--*

$$
\begin{aligned}
\text{simpleUnaryExpression} ::= \; & \texttt{!}\ \text{unaryExpression} \\
| \; & \texttt{(}\ \text{basicType}\ \texttt{)}\ \ \text{unaryExpression}\ \text{//cast} \\
| \; & \texttt{(}\ \text{referenceType}\ \texttt{)}\ \text{simpleUnaryExpression}\ \text{// cast} \\
| \; & \text{postfixExpression}
\end{aligned}
$$

involves lookahead and is shown below

```
private JExpression simpleUnaryExpression(): {
    int line = 0;
    Type type = null;
    JExpression expr = null, unaryExpr = null, simpleUnaryExpr = null;
}
{
    try {
        <LNOT> { line = token.beginLine; }
        unaryExpr = unaryExpression()
        { expr = new JLogicalNotOp(line, unaryExpr); } |
```

## What to do if we need to lookahead

```
        LOOKAHEAD(<LPAREN> basicType() <RPAREN>)
        <LPAREN> { line = token.beginLine; }
        type = basicType()
        <RPAREN>
        unaryExpr = unaryExpression()
        { expr = new JCastOp(line, type, unaryExpr); } |
        LOOKAHEAD(<LPAREN> referenceType() <RPAREN>)
        <LPAREN> { line = token.beginLine; }
        type = referenceType()
        <RPAREN>
        simpleUnaryExpr = simpleUnaryExpression()
        { expr = new JCastOp(line, type, simpleUnaryExpr); } |
        expr = postfixExpression()
    }
    catch (ParseException e) {
        recoverFromError(new int[] { SEMI, EOF }, e);
    }
    { return expr; }
}
```

## Error recovery

The error recovery mechanism we use in our JavaCC parser for *j*-- involves catching within the body of a non-terminal, the `ParseException` that is raised in the event of a parsing error

The exception instance `e` along with skip-to tokens is passed to our `recoverFromError()` error recovery function

The exception instance has information about the erroneous token that was found and the token that was expected, and `skipTo` is an array of tokens that we would like to skip to in order to recover from the error

In the current error recovery scheme, all non-terminals specify `SEMI` and `EOF` as `skipTo` tokens; this scheme could be made more sophisticated by specifying the follow set of the non-terminal as `skipTo` tokens

When `ParseException` is raised, control is transferred to the calling non-terminal, and thus when an error occurs within higher non-terminals, the lower non-terminals go unparsed

## Error recovery

The error recovery mechanism we use in our JavaCC parser for *j*-- involves catching within the body of a non-terminal, the `ParseException` that is raised in the event of a parsing error

The exception instance `e` along with skip-to tokens is passed to our `recoverFromError()` error recovery function

The exception instance has information about the erroneous token that was found and the token that was expected, and `skipTo` is an array of tokens that we would like to skip to in order to recover from the error

In the current error recovery scheme, all non-terminals specify `SEMI` and `EOF` as `skipTo` tokens; this scheme could be made more sophisticated by specifying the follow set of the non-terminal as `skipTo` tokens

When `ParseException` is raised, control is transferred to the calling non-terminal, and thus when an error occurs within higher non-terminals, the lower non-terminals go unparsed

## Error recovery

The error recovery mechanism we use in our JavaCC parser for *j*-- involves catching within the body of a non-terminal, the `ParseException` that is raised in the event of a parsing error

The exception instance `e` along with skip-to tokens is passed to our `recoverFromError()` error recovery function

The exception instance has information about the erroneous token that was found and the token that was expected, and `skipTo` is an array of tokens that we would like to skip to in order to recover from the error

In the current error recovery scheme, all non-terminals specify `SEMI` and `EOF` as `skipTo` tokens; this scheme could be made more sophisticated by specifying the follow set of the non-terminal as `skipTo` tokens

When `ParseException` is raised, control is transferred to the calling non-terminal, and thus when an error occurs within higher non-terminals, the lower non-terminals go unparsed

## Error recovery

The error recovery mechanism we use in our JavaCC parser for *j*-- involves catching within the body of a non-terminal, the `ParseException` that is raised in the event of a parsing error

The exception instance `e` along with skip-to tokens is passed to our `recoverFromError()` error recovery function

The exception instance has information about the erroneous token that was found and the token that was expected, and `skipTo` is an array of tokens that we would like to skip to in order to recover from the error

In the current error recovery scheme, all non-terminals specify `SEMI` and `EOF` as `skipTo` tokens; this scheme could be made more sophisticated by specifying the follow set of the non-terminal as `skipTo` tokens

When `ParseException` is raised, control is transferred to the calling non-terminal, and thus when an error occurs within higher non-terminals, the lower non-terminals go unparsed

## Error recovery

The error recovery mechanism we use in our JavaCC parser for *j*-- involves catching within the body of a non-terminal, the `ParseException` that is raised in the event of a parsing error

The exception instance `e` along with skip-to tokens is passed to our `recoverFromError()` error recovery function

The exception instance has information about the erroneous token that was found and the token that was expected, and `skipTo` is an array of tokens that we would like to skip to in order to recover from the error

In the current error recovery scheme, all non-terminals specify `SEMI` and `EOF` as `skipTo` tokens; this scheme could be made more sophisticated by specifying the follow set of the non-terminal as `skipTo` tokens

When `ParseException` is raised, control is transferred to the calling non-terminal, and thus when an error occurs within higher non-terminals, the lower non-terminals go unparsed

## Error recovery

```
private void recoverFromError(int[] skipTo, ParseException e) {
    // Get the possible expected tokens
    StringBuffer expected = new StringBuffer();
    for (int i = 0; i < e.expectedTokenSequences.length; i++) {
        for (int j = 0; j < e.expectedTokenSequences[i].length; j++) {
            expected.append("\n");
            expected.append("    ");
            expected.append(tokenImage[e.expectedTokenSequences[i][j]]);
            expected.append("...");
        }
    }

    // Print error message
    if (e.expectedTokenSequences.length == 1) {
        reportParserError("\"%s\" found where %s sought",
            getToken(1), expected);
    }
    else {
        reportParserError("\"%s\" found where one of %s sought",
            getToken(1), expected);
    }

    // Recover
    boolean loop = true;
    do {
        token = getNextToken();
        for (int i = 0; i < skipTo.length; i++) {
            if (token.kind == skipTo[i]) {
                loop = false;
                break;
            }
        }
    } while(loop);
}
```