Parsing

Outline

Introduction

Once we have identified the tokens in our program, we then want to determine its syntactic structure and the process of doing so 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

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)

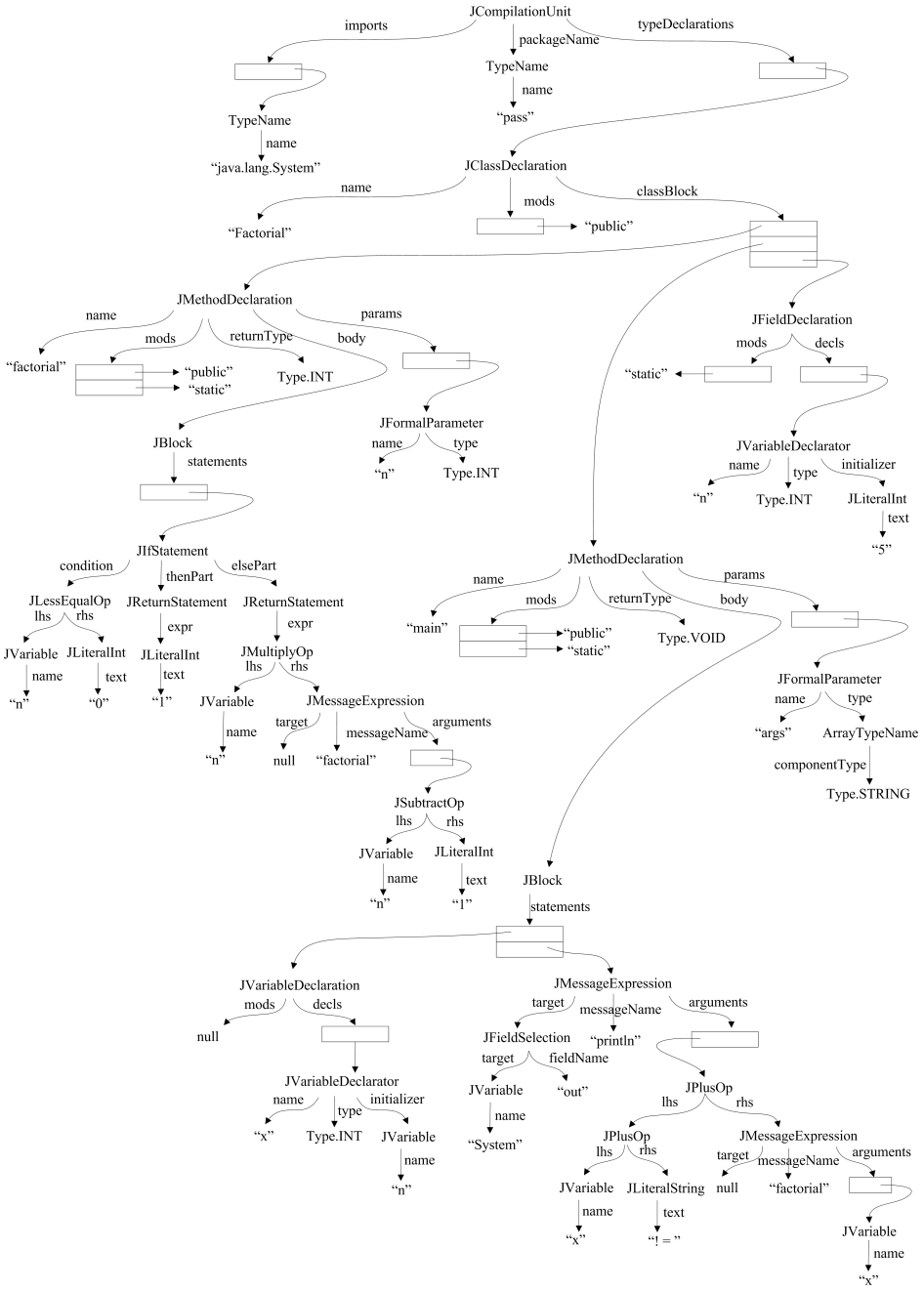
For example, given the following *j--* program

```
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

Introduction

AST for the `Factorial` program



Introduction

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 AST makes the syntax implicit in the program text, explicit, which is essentially the purpose of parsing

Context-free Grammars and Languages

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

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

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 and Languages

Square brackets indicate that a phrase is optional; for example, the two rules from above can be written as

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

Curly braces denote the Kleene closure, indicating that 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

$$T\ \texttt{+}\ T\ \texttt{+}\ T\ \ldots$$

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

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

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

$$T\ \texttt{+}\ T\ \texttt{-}\ T\ \texttt{+}\ T$$

Context-free Grammars and Languages

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, $G = (N, T, S, P)$ where

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

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

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

Context-free Grammars and Languages

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

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

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

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

Context-free Grammars and Languages

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`

We say the language $L(G)$ that is described by a grammar G consists of all the strings (sentences) comprised of only terminal symbols, that can be derived from the start symbol, ie, $L(G) = \{w | S \overset{*}{\Rightarrow} w \text{ and } w \in T*\}$

For example, in the grammar above

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

so, $L(G)$ includes each of

```
id + id * id
id
(id + id) * id
```

and infinitely more finite sentences

We are interested in languages consisting of strings of terminals that can be derived from a grammar's start symbol

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

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

Context-free Grammars and Languages

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

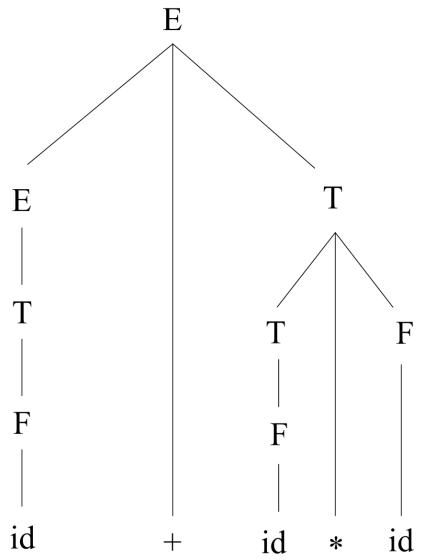
For example, in the previous derivation

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

Context-free Grammars and Languages

An alternative representation of a derivation is the parse tree, a tree that illustrates the derivation and the structure of an input string (at the leaves) from a start symbol (at the root)

For example, the following figure shows the parse tree for `id + id * id`.



Given a grammar G, if there exists a sentence s in $L(G)$ for which there are more than one left-most derivations in G (or equivalently, either more than one right-most derivations, or more than one parse tree for s in G), we say that the sentence s is ambiguous

If a grammar G describes at least one ambiguous sentence, the grammar G is an ambiguous grammar; if there is no such sentence, we say the grammar is unambiguous

Context-free Grammars and Languages

Consider the grammar

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

and, 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

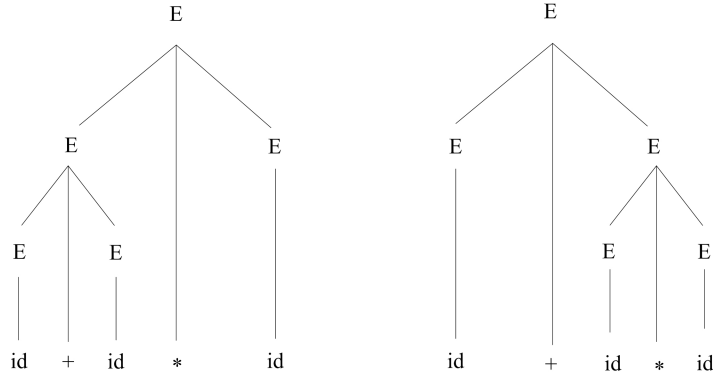
The above sentence also has two right-most derivations in the grammar

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

and

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

with the following two parse trees



As another example, consider the grammar describing conditional statements

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

and consider the sentence

```
if (e) if (e) s else s
```

Context-free Grammars and Languages

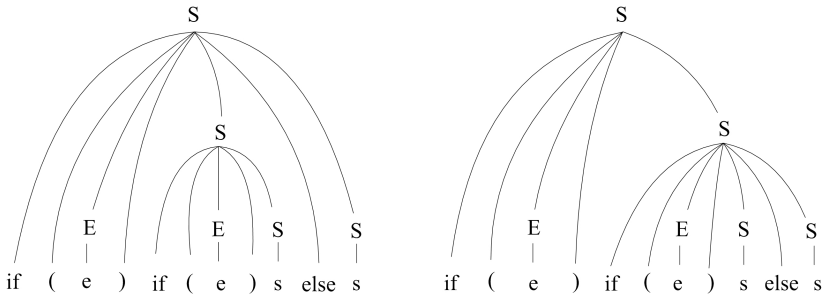
There exist two left-most derivations for the above sentence in the grammar

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

and

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

with the following parse trees



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

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

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`

Context-free Grammars and Languages

The *j--* grammar (and the 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 expression, then that would be evident in the syntax `x.y.z.w()`

But, what about `x.y.z`? There are several possibilities depending on the types of the names `x`, `y` and `z`

- If `x` is the name of 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
- Alternatively, `x.y` might be the name of a package, in which the class `z` is defined; in this case, we would parse the expression by first parsing `x.y.z` as a fully qualified class and then parse `w` as a static field selection of that class
- 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, so represents `x.y.z` 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

Top-down Deterministic Parsing

There are two popular top-down deterministic parsing strategies: parsing by recursive descent and LL(1) parsing; both 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

Top-down Deterministic Parsing

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

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

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>`

That we start at the start symbol, and continually rewrite non-terminals using BNF rules until we eventually reach leaves (the tokens are the leaves) makes this a top-down parsing technique

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

Top-down Deterministic Parsing

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

Top-down Deterministic Parsing

In the above example, the decision could be made by looking at the next single un-scanned input token; when this is the case, we say that our grammar is LL(1)

In some cases, one must lookahead several tokens in the input to decide which alternative to take

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 available to us

1. Parsing by recursive descent
2. LL(1) or LL(k) parsing

Top-down Deterministic Parsing

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 decides which rule to apply and then scans any terminals (tokens) on the right hand side by calling upon the `Scanner`, and 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`

As another example, consider the rule defining qualifiedIdentifier

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

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 incoming token and if it is an identifier, we scan it; otherwise, we raise an error

Top-down Deterministic Parsing

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

Top-down Deterministic Parsing

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);
```

Top-down Deterministic Parsing

```
        } 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;
        }
    }
```

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

Top-down Deterministic Parsing

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 in deciding what to do, ie, we must look ahead 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 that is a parenthesized expression; for example `(x)`

Top-down Deterministic Parsing

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;
    }
}
```

Top-down Deterministic Parsing

```
    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;
    }
```

The method `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 (ie, for 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

Top-down Deterministic Parsing

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

Top-down Deterministic Parsing

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

Top-down Deterministic Parsing

```
    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

Top-down Deterministic Parsing

The recursive invocation of methods in the recursive descent parsing technique depends on the underlying program stack for keeping track of the recursive calls

The LL(1) parsing technique makes this stack explicit

The first L in its name indicates a left-to-right scan of the input token stream; the second L signifies that it produces a left parse, which is a left-most derivation; the (1) indicates we just look ahead at the next one symbol in the input to make a decision

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

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, and 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

Top-down Deterministic Parsing

LL(1) parsing technique is table-driven, with a unique parse table is produced for each grammar; this table has a row for each non-terminal and a column for each terminal, including a special terminator `#` to mark the end of the string

The parser consults this table, given the non-terminal on top of the stack and the next incoming token to determine which BNF rule to use in rewriting the non-terminal

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

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

Top-down Deterministic Parsing

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

	+	*	(	)	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

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

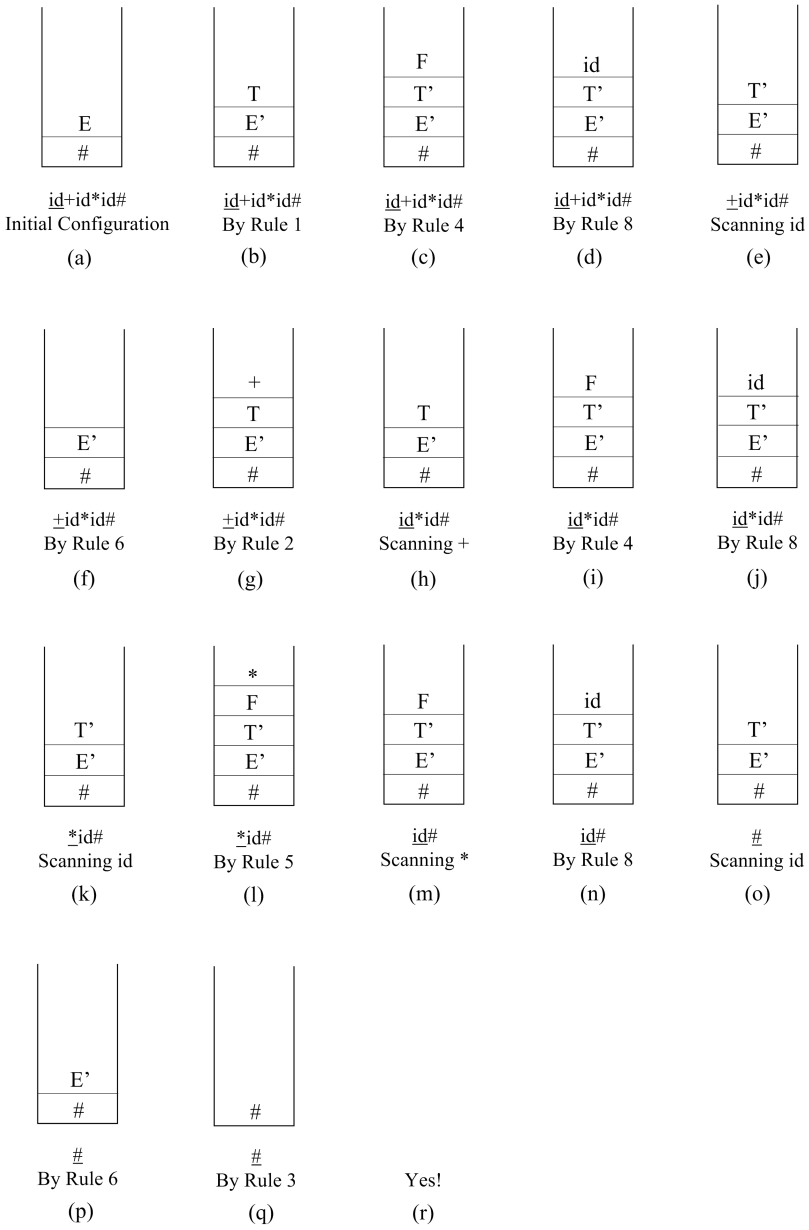
Output: a left-parse, which is a left-most derivation for w

Algorithm LL(1) Parsing Algorithm (contd.)

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 $top \leftarrow stk$.pop()
  if $top = sym =$ # **then**
    Halt successfully
  else if top is a terminal symbol **then**
    if $top = sym$ **then**
      Advance sym to be 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**
    $index \leftarrow table[Y, sym]$
    if $index \neq err$ **then**
      $rule \leftarrow rules[index]$
      Say rule is $Y ::= X_1 X_2 \ldots X_n$; push $X_n, \ldots, X_2, X_1$ onto the stack stk, with X_1 on top
    else
      Halt with an error
    end if
  end if
end while

Top-down Deterministic Parsing

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



Top-down Deterministic Parsing

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?

table[Y, a] $= i$, where i is the number of the rule $Y ::= X_1 X_2 \ldots X_n$

says that 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 $X_1 X_2 \ldots X_n$

If we consider the last two rules of our grammar

7. $F ::=$ `(`E`)`
8. $F ::=$ `id`

and 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, and if it is an `id`, we apply rule 8, ie,

$\text{table}[F,$ `(`$] = 7$
$\text{table}[F,$ `id`$] = 8$

Top-down Deterministic Parsing

The problem becomes slightly more complicated when the right-hand-side of the rule either starts with a non-terminal or is simply ϵ

In general, assuming both α and β are (possibly empty) strings of terminals and non-terminals, table[Y, a] $= i$, where i is the number of the rule $Y ::= X_1 X_2 \ldots X_n$, if either

1. $X_1 X_2 \ldots X_n \stackrel{*}{\Rightarrow} a\alpha$, or
2. $X_1 X_2 \ldots X_n \stackrel{*}{\Rightarrow} \epsilon$, and there is a derivation S# $\stackrel{*}{\Rightarrow} \alpha Y a \beta$, that is, a can follow Y in a derivation.

For this we need two helper functions, first and follow

We define, first$(X_1 X_2 \ldots X_n) = \{a | X_1 X_2 \ldots X_n \stackrel{*}{\Rightarrow} a\alpha, a \in T\}$, ie, the set of all terminals that can start strings derivable from $X_1 X_2 \ldots X_n$; also, if $X_1 X_2 \ldots X_n \stackrel{*}{\Rightarrow} \epsilon$, then we say that first$(X_1 X_2 \ldots X_n)$ includes ϵ

We define two algorithms for computing first: one for a single symbol and the other 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

Algorithm Compute first(X) for all symbols X in a Grammar G

Input: a context-free grammar $G = (N, T, S, P)$
Output: first(X) for all symbols X in G

```
for each terminal x do
    first(x) ← {x}
end for
for each non-terminal X do
    first(X) ← {}
end for
if X ::= ε ∈ P then
    add ε to first(X)
end if
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
```

Algorithm Compute first$(X_1 X_2 \ldots X_n)$ for a Grammar G

Input: a context-free grammar $G = (N, T, S, P)$ and a sequence $X_1 X_2 \ldots X_n$
Output: first$(X_1 X_2 \ldots X_n)$

```
Set S ← first(X_1)
i ← 2
while ε ∈ S and i ≤ n do
    S ← S − ε
    Add first(X_i) to S
    i ← i + 1
end while
return S
```

Top-down Deterministic Parsing

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

$$\begin{aligned}\text{first}(E') &= \{\epsilon\}\\ \text{first}(T') &= \{\epsilon\}\end{aligned}$$

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

$$\begin{aligned}\text{first}(E) &= \{\}\\ \text{first}(E') &= \{\texttt{+}, \epsilon\}\\ \text{first}(T) &= \{\texttt{(}, \texttt{id}\}\\ \text{first}(T') &= \{\texttt{*}, \epsilon\}\\ \text{first}(F) &= \{\texttt{(}, \texttt{id}\}\end{aligned}$$

Top-down Deterministic Parsing

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

When is a rule $X ::= \epsilon$ applicable? For this we need the notion of follow

We define, follow$(X) = \{a | S \stackrel{*}{\Rightarrow} wX\alpha$ and $\alpha \stackrel{*}{\Rightarrow} a \dots\}$, ie, all terminal symbols that start terminal strings derivable from what can follow X in a derivation

Another definition is as follows

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(β) contains ϵ, follow(X) contains follow(Y)

Algorithm Compute follow(X) for all non-terminals X in a Grammar G

Input: a context-free grammar $G = (N, T, S, P)$
Output: follow(X) for all non-terminals X in G

follow(S) $\leftarrow \{\#\}$
for each non-terminal $X \in S$ **do**
  follow(X) $\leftarrow \{\}$
end for
repeat
  for each rule $Y ::= X_1 X_2 \dots X_n \in P$ **do**
    for each non-terminal X_i **do**
      Add first($X_{i+1} X_{i+2} \dots X_n$) $- \{\epsilon\}$ to follow(X_i), and if X_i is the right-most symbol or first($X_{i+1} X_{i+2} \dots X_n$) contains ϵ, add follow(Y) to follow(X_i)
    end for
  end for
until no new symbols are added to any set

Again, 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. $F \ ::= \texttt{id}$

From rule 1, $E ::= TE'$, follow(T) contains first(E') - $\{\epsilon\} = \{+\}$, and because first(E') contains ϵ, it also contains follow(E); also, follow(E') contains follow(E); so, round 1 of step 3 yields,

$$\begin{aligned}\text{follow}(E') &= \{\texttt{\#}\} \\ \text{follow}(T) &= \{\texttt{+}, \texttt{\#}\}\end{aligned}$$

We get nothing additional from rule 2, $E' ::= \texttt{+} \ TE'$; 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, $E' ::= \epsilon$

Top-down Deterministic Parsing

From rule 4, $T ::= FT'$, follow(F) contains first(T') - $\{\epsilon\} = \{*\}$, and because first(T') contains ϵ, it also contains follow(T); also, follow(T') contains follow(T); so, we have

$$\begin{aligned}\text{follow}(T') &= \{+, \#\}\\ \text{follow}(F) &= \{+, *, \#\}\end{aligned}$$

Rules 5 and 6 give us nothing new (for the same reasons rules 2 and 3 did not)

Rule 7, $F ::= (E)$, adds) to follow(E), so

$$\text{follow}(E) = \{), \#\}$$

Rule 8 gives us nothing

So round 1 of step 3 gives

$$\begin{aligned}\text{follow}(E) &= \{), \#\}\\ \text{follow}(E') &= \{\#\}\\ \text{follow}(T) &= \{+, \#\}\\ \text{follow}(T') &= \{+, \#\}\\ \text{follow}(F) &= \{+, *, \#\}\end{aligned}$$

Top-down Deterministic Parsing

Now, in round 2 of step 3, the) that was added to follow(E) trickles down into the other follow sets

From rule 1, $E ::= TE'$, because first(E') contains ϵ, follow(T) contains follow(E); also, follow(E') contains follow(E); so, we have

$$\begin{aligned}\text{follow}(E') &= \{\texttt{)}, \texttt{\#}\}\\ \text{follow}(T) &= \{\texttt{+}, \texttt{)}, \texttt{\#}\}\end{aligned}$$

From rule 4, $T ::= FT'$, because first(T') contains ϵ, follow(F) contains follow(T); also, follow(T') contains follow(T); so, we have

$$\begin{aligned}\text{follow}(T') &= \{\texttt{+}, \texttt{)}, \texttt{\#}\}\\ \text{follow}(F) &= \{\texttt{+}, \texttt{*}, \texttt{)}, \texttt{\#}\}\end{aligned}$$

So round 2 produces

$$\begin{aligned}\text{follow}(E) &= \{\texttt{)}, \texttt{\#}\}\\ \text{follow}(E') &= \{\texttt{)}, \texttt{\#}\}\\ \text{follow}(T) &= \{\texttt{+}, \texttt{)}, \texttt{\#}\}\\ \text{follow}(T') &= \{\texttt{+}, \texttt{)}, \texttt{\#}\}\\ \text{follow}(F) &= \{\texttt{+}, \texttt{*}, \texttt{)}, \texttt{\#}\}\end{aligned}$$

Round 3 of step 3 adds no new symbols to any set, so we are done

Top-down Deterministic Parsing

Recall, assuming both α and β are (possibly empty) strings of terminals and non-terminals, table$[Y, a] = i$, where i is the number of the rule $Y ::= X_1 X_2 \ldots X_n$ if either

1. $X_1 X_2 \ldots X_n \stackrel{*}{\Rightarrow} a\alpha$, or
2. $X_1 X_2 \ldots X_n \stackrel{*}{\Rightarrow} \epsilon$, and there is a derivation $S\# \stackrel{*}{\Rightarrow} \alpha Y a \beta$, that is, 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 $G = (N, T, S, P)$

Input: a context-free grammar $G = (N, T, S, P)$
Output: LL(1) Parse Table for G

```
for each non-terminal Y ∈ G 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
```

Top-down Deterministic Parsing

Let us construct a parse table for 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. $F \ ::= \texttt{id}$

For the non-terminal E, we consider rule 1: $E ::= TE'$; first$(TE') = \{\texttt{(},\texttt{id}\}$; so, we have

$$\mathrm{table}[E, \texttt{(}] = 1$$
$$\mathrm{table}[E, \texttt{id}] = 1$$

Because first(TE') does not contain ϵ, we need not consider follow(E)

For the non-terminal E', we first consider rule 2: $E' ::= \texttt{+} \ TE'$; first$(\texttt{+} \ T \ E') = \{\texttt{+}\}$, so we have

$$\mathrm{table}[E', \texttt{+}] = 2$$

Top-down Deterministic Parsing

Rule 3: $E' ::= \epsilon$ is applicable for symbols in follow(E') = $\{$), #$\}$, so we have

$\text{table}[E', \texttt{)}] = 3$
$\text{table}[E', \texttt{\#}] = 3$

For the non-terminal T, we consider rule 4: $T ::= FT'$; first($F\ T'$) = $\{$(, id$\}$, so we have

$\text{table}[T, \texttt{(}] = 4$
$\text{table}[T, \texttt{id}] = 4$

Because first($F\ T'$) does not contain ϵ, we need not consider follow(T)

For non-terminal T', we first consider rule: 5: $T' ::= * FT'$; first($*\ F\ T'$) = $\{*, \epsilon\}$, so we have

$\text{table}[T', *] = 5$

Rule 6: $T' ::= \epsilon$ is applicable for symbols in follow(T') = $\{+, \texttt{)}, \texttt{\#}\}$, so we have

$\text{table}[T', +] = 6$
$\text{table}[T', \texttt{)}] = 6$
$\text{table}[T', \texttt{\#}] = 6$

Top-down Deterministic Parsing

Rule 7: $F ::= \texttt{(}E\texttt{)}$, and since first($\texttt{(}E\texttt{)}$) = $\{\texttt{(}\}$, we have

$$\text{table}[F, \texttt{(}] = 7$$

Rule 8: $F ::= \texttt{id}$, and since first($\texttt{id}$) = $\{\texttt{id}\}$, we have

$$\text{table}[F, \texttt{id}] = 8$$

We say a grammar is LL(1) if the parse table has no conflicts, ie, 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

On the other hand, an LL(1) parser generator (JavaCC for example) based on the table construction algorithm, 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

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 grammar that is not LL(1) is a grammar having a rule with left recursion; for example

$$\begin{aligned} \mathrm{Y} &::= \mathrm{Y}\ \alpha \\ \mathrm{Y} &::= \beta \end{aligned}$$

Clearly, a grammar having these two rules is not LL(1), because, by definition, first($Y\alpha$) must include first(β) 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

$$\begin{aligned} \mathrm{Y} &::= \beta\ Y' \\ Y' &::= \alpha\ Y' \\ Y' &::= \epsilon \end{aligned}$$

Top-down Deterministic Parsing

For example, consider the following context-free grammar

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

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 produce

$$
\begin{aligned}
E &::= T\ E' \\
E' &::= \texttt{ + } T\ E' \\
E' &::= \epsilon
\end{aligned}
$$

Top-down Deterministic Parsing

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

$$
\begin{aligned}
E &::= T\ E' \\
E' &::= \texttt{+}\ T\ E' \\
E' &::= \epsilon \\
T &::= F\ T' \\
T' &::= \texttt{*}\ F\ T' \\
T' &::= \epsilon \\
F &::= \texttt{(}E\texttt{)} \\
F &::= \texttt{id}
\end{aligned}
$$

Algorithm Left Recursion Removal for a Grammar $G = (N, T, S, P)$

Input: a context-free grammar $G = (N, T, S, P)$
Output: 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_i
    Eliminate any direct left recursion
  end for
end for

Bottom-up Deterministic Parsing

Consider the grammar

1. $E ::= E$ `+` T
2. $E ::= T$
3. $T ::= T$ `*` F
4. $T ::= F$
5. $F ::=$ `(`E`)`
6. $F ::=$ `id`

and suppose we want to parse the input string `id + id * id`

We would start off with the initial configuration

Stack	Input	Action
	`id+id*id#`	

The first action is to shift the first un-scanned input symbol onto the stack

Stack	Input	Action
	`id+id*id#`	shift
`id`	`+id*id#`	

Bottom-up Deterministic Parsing

From this configuration, the next action is to reduce the `id` on top of the stack to an F using rule 6

Stack	Input	Action
	`id+id*id#`	shift
`id`	`+id*id#`	reduce 6, output a 6
F	`+id*id#`	

From this configuration, the next two actions involve reducing the F to a T (by rule 4), and then to an E (by rule 2)

Stack	Input	Action
	`id+id*id#`	shift
`id`	`+id*id#`	reduce 6, output a 6
F	`+id*id#`	reduce 5, output a 4
T	`+id*id#`	reduce 2, output a 2
E	`+id*id#`	

Bottom-up Deterministic Parsing

The parser continues in this fashion, by a sequence of shifts and reductions, until we reach a configuration where E is on the stack (E on top) and the sole un-scanned symbol in the input is the terminator symbol `#`

Stack	Input	Action
	id+id*id#	shift
id	+id*id#	reduce 6, output a 6
F	+id*id#	reduce 4, output a 4
T	+id*id#	reduce 2, output a 2
E	+id*id#	shift
E+	id*id#	shift
E+id	*id#	reduce 6, output a 6
E+F	*id#	reduce 4, output a 4
E+T	*id#	shift
E+T*	id#	shift
E+T*id	#	reduce 6, output a 6
E+T*F	#	reduce 3, output a 3
E+T	#	reduce 1, output a 1
E	#	accept

At this point, we have reduced the entire input string to the grammar's start symbol E, so we can say the input is accepted

Bottom-up Deterministic Parsing

The sequence of reductions 6, 4, 2, 6, 4, 6, 3, 1 above represents the right-most derivation of the input string, but in reverse

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

The following questions arise

- How does the parser know when to shift and when to reduce?
- When reducing, how many symbols on top of the stack play a role in the reduction?
- Also, when reducing, by which rule does it make its reduction?

For example, in the derivation above, when the stack contains E+T and the next incoming token is a $*$, how do we know that we are to shift (the $*$ onto the stack) rather than reduce either the E+T to an E or the T to an E?

Bottom-up Deterministic Parsing

A couple of things to notice

1. The stack configuration combined with the un-scanned input stream represents a sentential form in a right-most derivation of the input
2. The part of the sentential form that is reduced to a non-terminal is always on top of the stack, so all actions take place at the top of the stack — we either shift a token onto the stack, or we reduce what is already there

We call the sequence of terminals on top of the stack that are reduced to a single non-terminal at each reduction step the handle

More formally, in a right-most derivation, $S \stackrel{*}{\Rightarrow} \alpha Y w \Rightarrow \alpha\beta w \stackrel{*}{\Rightarrow} uw$, where uw is the sentence, the handle is the rule $Y ::= \beta$ and a position in the right sentential form $\alpha\beta w$ where β may be replaced by Y to produce the previous right sentential form $\alpha Y w$ in a right-most derivation from the start symbol S

So, when a handle appears on top of the stack

Stack	Input
$\alpha\beta$	w

we reduce that handle (β to Y in this case)

Bottom-up Deterministic Parsing

Now if β is the sequence $X_1, X_2, \ldots, X_n$, then we call any subsequence, $X_1, X_2, \ldots, X_i$, for $i \leq n$ a viable prefix; only viable prefixes may appear on the top of the parse stack

If there is not a handle on top of the stack and shifting the first un-scanned input token from the input to the stack results in a viable prefix, a shift is called for

The LR(1) parsing algorithm is common to all LR(1) grammars and is driven by two tables (derived from a DFA that recognizes viable prefixes and handles): an Action table and a Goto table

The algorithm is a state machine with a pushdown stack, driven by two tables: Action and Goto

A configuration of the parser is a pair, consisting of the state of the stack and the state of the input

Stack	Input
$s_0 X_1 s_1 X_2 s_2 \ldots X_m s_m$	$a_k a_{k+1} \ldots a_n$

where the s_i are states, the X_i are (terminal or non-terminal) symbols, and $a_k a_{k+1} \ldots a_n$ are the un-scanned input symbols

This configuration represents a right sentential form in a right-most derivation of the input sentence $X_1 X_2 \ldots X_m a_k a_{k+1} \ldots a_n$

Algorithm The LR(1) Parsing Algorithm

Input: Action and Goto tables, and the input sentence w to be parsed, followed by the terminator #

Output: a right-most derivation in reverse

Initially, the parser has the configuration,

Stack	Input
s_0	$a_1 a_2 \ldots a_n$#

where $a_1 a_2 \ldots a_n$ is the input sentence

Algorithm The LR(1) Parsing Algorithm (contd.)

repeat

If Action$[s_m, a_k] = ss_i$, the parser executes a shift (the s stands for "shift") and goes into state s_i, going into the configuration

Stack	Input
$s_0 X_1 s_1 X_2 s_2 \ldots X_m s_m a_k s_i$	$a_{k+1} \ldots a_n \#$

Otherwise, if Action$[s_m, a_k] = ri$ (the r stands for "reduce"), where i is the number of the production rule $Y ::= X_j X_{j+1} \ldots X_m$, then replace the symbols and states $X_j s_j X_{j+1} s_{j+1} \ldots X_m s_m$ by $Y s$, where $s = \text{Goto}[s_{j-1}, Y]$; the parser then outputs production number i, and goes into the configuration

Stack	Input
$s_0 X_1 s_1 X_2 s_2 \ldots X_{j-1} s_{j-1} Y s$	$a_{k+1} \ldots a_n \#$

Otherwise, if Action$[s_m, a_k] =$ accept, then the parser halts and the input has been successfully parsed

Otherwise, if Action$[s_m, a_k] =$ error, then the parser raises an error; the input is not in the language

until either the sentence is parsed or an error is raised

Bottom-up Deterministic Parsing

The Action and Goto tables for our grammar for simple expressions are shown below

	Action						Goto		
	+	*	(	)	id	#	E	T	F
0			s4		s5		1	2	3
1	s6					accept			
2	r2	s7				r2			
3	r4	r4				r4			
4			s11		s12		8	9	10
5	r6	r6				r6			
6			s4		s5			13	3
7			s4		s5				14
8	s16			s15					
9	r2	s17		r2					
10	r4	r4		r4					
11			s11		s12		18	9	10
12	r6	r6		r6					
13	r1	s7				r1			
14	r3	r3				r3			
15	r5	r5				r5			
16			s11		s12			19	10
17			s11		s12				20
18	s16			s21					
19	r1	s17		r1					
20	r3	r3		r3					
21	r5	r5		r5					

Bottom-up Deterministic Parsing

Consider parsing the input string `id + id * id`; initially the parser is in state 0, so a 0 is pushed onto the stack

Stack	Input	Action
0	id+id*id#	

The next incoming symbol is an `id`, so we consult the Action table, Action[0, `id`] to determine what to do in state 0 with an incoming token, `id`; the entry is $s5$, so we shift the `id` onto the stack and go into state 5 (pushing the new state onto the stack above the `id`)

Stack	Input	Action
0	id+id*id#	shift 5
0id5	+id*id#	

Now, the 5 on top of the stack indicates we are in state 5 and the incoming token is `+`, so we consult Action[5, `+`]; the $r6$ indicates a reduction using rule 6: $F ::=$ `id`; to make the reduction, we pop $2k$ items off the stack where k is the number of symbols in the rule's right hand side; in our example $k = 1$ so we pop both the 5 and the `id`

Stack	Input	Action
0	id+id*id#	shift 5
0id5	+id*id#	reduce 6, output a 6
0	+id*id#	

Bottom-up Deterministic Parsing

Because we are reducing the right hand side to an F in this example, we push the F onto the stack.

Stack	Input	Action
0	id+id*id#	shift 5
0id5	+id*id#	reduce 6, output a 6
0F	+id*id#	

Finally, we consult Goto[0, F] to determine which state the parser, initially in state 0, should go into after parsing an F; since Goto[0, F] = 3, this is state 3; we push the 3 onto the stack to indicate the parser's new state

Stack	Input	Action
0	id+id*id#	shift 5
0id5	+id*id#	reduce 6, output a 6
0F3	+id*id#	

From state 3 and looking at the incoming token +, Action[3, +] tells us to reduce using rule 4: $T ::= F$

Stack	Input	Action
0	id+id*id#	shift 5
0id5	+id*id#	reduce 6, output a 6
0F3	+id*id#	reduce 4, output a 4
0T2	+id*id#	

Bottom-up Deterministic Parsing

From state 2 and looking at the incoming token +, Action[2, +] tells us to reduce using rule 2: $E ::= T$

Stack	Input	Action
0	id+id*id#	shift 5
0id5	+id*id#	reduce 6, output a 6
0F3	+id*id#	reduce 4, output a 4
0T2	+id*id#	reduce 2, output a 2
0E1	+id*id#	

From state 1 and looking the incoming token +, Action[3, +] $= s6$ tells us to shift (the + onto the stack and go into state 6

Stack	Input	Action
0	id+id*id#	shift 5
0id5	+id*id#	reduce 6, output a 6
0F3	+id*id#	reduce 4, output a 4
0T2	+id*id#	reduce 2, output a 2
0E1	+id*id#	shift 6
0E1+6	id*id#	

Bottom-up Deterministic Parsing

Continuing in this fashion, the parser goes through the following sequence of configurations and actions

Stack	Input	Action
0	id+id*id#	shift 5
0id5	+id*id#	reduce 6, output a 6
0F3	+id*id#	reduce 4, output a 4
0T2	+id*id#	reduce 2, output a 2
0E1	+id*id#	shift 6
0E1+6	id*id#	shift 5
0E1+6id5	*id#	reduce 6, output a 6
0E1+6F3	*id#	reduce 4, output a 4
0E1+6T13	*id#	shift 7
0E1+6T13*7	id#	shift 5
0E1+6T13*7id5	#	reduce a 6, output 6
0E1+6T13*7F14	#	reduce 3, output a 3
0E1+6T13	#	reduce 1, output a 1
0E1	#	accept

Bottom-up Deterministic Parsing

The parser has output 6, 4, 2, 6, 4, 6, 3, 1, which is a right-most derivation of the input string in reverse: 1, 3, 6, 4, 6, 2, 4, 6, ie

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

Bottom-up Deterministic Parsing

The LR(1) parsing tables, Action and Goto, for a grammar G are derived from a DFA for recognizing the possible handles for a parse in G

This DFA is constructed from what is called an LR(1) canonical collection, a collection of sets of items (representing potential handles) of the form

$$[\mathrm{Y} ::= \alpha \ \cdot \ \beta, \, \mathtt{a}]$$

where $Y ::= \alpha\beta$ is a production rule in the set of productions P, α and β are (possibly empty) strings of symbols, and `a` is a lookahead

The $\cdot$ is a position marker that marks the top of the stack, indicating that we have parsed the α and still have the β ahead of us in satisfying the Y

The lookahead symbol `a` is a token that can follow Y (and so, $\alpha\beta$) in a legal right-most derivation of some sentence

Bottom-up Deterministic Parsing

If the position marker comes at the start of the right hand side in an item

$$[\mathrm{Y} ::= \cdot\ \alpha\ \beta,\ \mathtt{a}]$$

the item is called a possibility

One way of parsing the Y is to first parse the α and then parse the β, after which point the next incoming token will be an $\mathtt{a}$; the parser might be in the following configuration

Stack	Input
γ	$u\mathtt{a}\ldots$

where $\alpha\beta \stackrel{*}{\Rightarrow} u$, where u is a string of terminals

Bottom-up Deterministic Parsing

If the position marker comes after a string of symbols α but before a string of symbols β in the right hand side in an item

$$[\mathrm{Y} ::= \alpha \ \cdot \ \beta, \, \mathtt{a}]$$

the item indicates that α has been parsed (and so is on the stack) but that there is still β to parse from the input

Stack	Input
$\gamma\alpha$	$v\mathtt{a}\ldots$

where $\beta \stackrel{*}{\Rightarrow} v$, where v is a string of terminals

Bottom-up Deterministic Parsing

If the position marker comes at the end of the right hand side in an item

$[\mathrm{Y} ::= \alpha\ \beta\ \cdot,\ \mathtt{a}]$

the item indicates that the parser has successfully parsed $\alpha\beta$ in a context where $Y\,\mathtt{a}$ would be valid, the $\alpha\beta$ can be reduced to a Y, and so $\alpha\beta$ is a handle, ie, the parser is in the configuration

Stack	Input
$\gamma\alpha\beta$	$\mathtt{a}\ldots$

and the reduction of $\alpha\beta$ would cause the parser to go into the configuration

Stack	Input
γY	$\mathtt{a}\ldots$

Bottom-up Deterministic Parsing

The states in the DFA for recognizing viable prefixes and handles are constructed from items, and we call the set of states the canonical collection

We first augment our grammar G with an additional start symbol S' and an additional rule so as to yield an equivalent grammar G', which does not have its start symbol on the right hand side of any rule

$$S' ::= S$$

For example, augmenting our grammar for simple expressions gives us the following grammar

0. $E' ::= E$
1. $E ::= E$ `+` T
2. $E ::= T$
3. $T ::= T$ `*` F
4. $T ::= F$
5. $F ::=$ `(`E`)`
6. $F ::=$ `id`

The first set, representing the initial state in our DFA, will contain the LR(1) item

$$\{[E' ::= \cdot\ E,\ \texttt{\#}]\}$$

which says that parsing an E' means parsing an E from the input, after which point the next (and last) remaining un-scanned token should be the terminator `#`

Bottom-up Deterministic Parsing

We must parse an E next, which means that we might be parsing either an E + T (by rule 1) or a T (by rule 2); so the initial set would also contain

$[E \ ::= \cdot \ E \ \texttt{+} \ T, \ \texttt{\#}]$
$[E \ ::= \cdot \ T, \ \texttt{\#}]$

The initial set, called kernel, may imply additional items

From the kernel, we compute a closure of all items implied by the kernel

---

Algorithm Computing the Closure of a Set of Items

Input: a set of items, s
Output: closure(s)
  add s to closure(s)
  repeat
    if closure(s) contains an item of the form,
    $[Y ::= \alpha \cdot X \ \beta, \ \texttt{a}]$
    add the item
    $[X ::= \cdot \ \gamma, \ \texttt{b}]$
    for every rule $X ::= \gamma$ in P and for every token b in first($\beta\texttt{a}$).
  until no new items may be added

Bottom-up Deterministic Parsing

For example, closure($\{[E' ::= \cdot E, \#]\}$) by step 1 is initially

$$\{[E' ::= \cdot\ E, \#]\}$$

We then invoke step 2; because the $\cdot$ comes before the E, and because we have the rule $E ::= E + T$ and $E ::= T$, we add $[E ::= \cdot\ E + T, \#]$ and $[E ::= \cdot\ T, \#]$ to get

$$\begin{array}{l}\{[E' ::= \cdot\ E, \#],\\ \ \ [E\ \ ::= \cdot\ E + T, \#],\\ \ \ [E\ \ ::= \cdot\ T, \#]\}\end{array}$$

The item $[E ::= \cdot\ E + T, \#]$ implies

$$\begin{array}{l}[E\ \ ::= \cdot\ E + T, +]\\ [E\ \ ::= \cdot\ T, +]\end{array}$$

because first($+T\#$) $= \{+\}$

Given that these items differ from previous items only in the lookaheads, we can use the more compact notation to get

$$\begin{array}{l}\{[E' ::= \cdot\ E, \#],\\ \ \ [E\ \ ::= \cdot\ E + T, +/\#],\\ \ \ [E\ \ ::= \cdot\ T, +/\#]\}\end{array}$$

Bottom-up Deterministic Parsing

The items $[E ::= \cdot\ T,\ \texttt{+/\#}]$ imply additional items (by similar logic), leading to

$$\begin{aligned}\{&[E' ::= \cdot\ E,\ \texttt{\#}],\\ &[E\ ::= \cdot\ E\ \texttt{+}\ T,\ \texttt{+/\#}],\\ &[E\ ::= \cdot\ T,\ \texttt{+/\#}],\\ &[T\ ::= \cdot\ T\ \ \texttt{*}\ F,\ \texttt{+/*/\#}],\\ &[T\ ::= \cdot\ F,\ \texttt{+/*/\#}]\}\end{aligned}$$

And finally the items $[T ::= \cdot\ F,\ \texttt{+/*/\#}]$ imply additional items (by similar logic), leading to

$$\begin{aligned}s_0 = \{&[E' ::= \cdot\ E,\ \texttt{\#}],\\ &[E\ ::= \cdot\ E\ \texttt{+}\ T,\ \texttt{+/\#}],\\ &[E\ ::= \cdot\ T,\ \texttt{+/\#}],\\ &[T\ ::= \cdot\ T\ \texttt{*}\ F,\ \texttt{+/*/\#}],\\ &[T\ ::= \cdot\ F,\ \texttt{+/*/\#}],\\ &[F\ ::= \cdot\texttt{(}E\texttt{)},\ \texttt{+/*/\#}],\\ &[F\ ::= \cdot\ \texttt{id},\ \texttt{+/*/\#}]\}\end{aligned}$$

which represents the initial state s_0 in our canonical LR(1) collection

Bottom-up Deterministic Parsing

We now need to compute all of the states and transitions of the DFA that recognizes viable prefixes and handles

For any item set s, and any symbol $X \in (T \cup N)$

$$\text{goto}(s, X) = \text{closure}(r),$$

where $r = \{[Y ::= \alpha X \cdot \beta, a] | [Y ::= \alpha \cdot X \beta, a]\}$, ie, to compute goto$(s, X)$, we take all items from s with a $\cdot$ before the X and move it after the X; we then take the closure of that

---

Algorithm Computing goto

Input: a state s, and a symbol $X \in T \cup N$
Output: the state, goto(s, X)

```
r ← {}
for each item [Y ::= α · Xβ, a] in s do
    add [Y ::= αX · β, a] to r
end for
return closure(r)
```

Bottom-up Deterministic Parsing

Consider the computation of goto(s_0, E) for our running example; the relevant items in s_0 are $[E' ::= \cdot E,$ `#`$]$ and $[E ::= \cdot E$ `+` $T,$ `+/#`$]$; moving the $\cdot$ to the right of the E in the items gives us $\{[E' ::= E\cdot,$ `#`$]$, $[E ::= E\cdot$ `+` $T,$ `+/#`$]\}$; the closure of this set is the set itself; let us call this state s_1

$$\begin{aligned}\text{goto}(s_0,\, E) &= s_1 \\ &= \{[E' ::= E \cdot,\, \texttt{\#}], \\ &\quad\ [E \ ::= E \cdot \texttt{+}\ T,\, \texttt{+/\#}]\}\end{aligned}$$

We can similarly compute

$$\begin{aligned}\text{goto}(s_0,\, T) &= s_2 \\ &= \{[E \ ::= T \cdot,\, \texttt{+/\#}], \\ &\quad\ [T \ ::= T \cdot \texttt{*}\ F,\, \texttt{+/*/\#}]\}\end{aligned}$$

$$\begin{aligned}\text{goto}(s_0,\, F) &= s_3 \\ &= \{[T \ ::= F \cdot,\, \texttt{+/*/\#}]\}\end{aligned}$$

goto(s_0, `(`) involves a closure since moving the $\cdot$ across the `(` puts it in front of the E

$$\begin{aligned}
\text{goto}(s_0, \texttt{(}) &= s_4 \\
&= \{[F \ ::= \texttt{(}\cdot\ E\texttt{)},\ \texttt{+/*/\#}], \\
&\quad\ [E \ ::= \cdot\ E\ \texttt{+}\ T,\ \texttt{+/)}], \\
&\quad\ [E \ ::= \cdot\ T,\ \texttt{+/)}], \\
&\quad\ [T \ ::= \cdot\ T\ \texttt{*}\ F,\ \texttt{+/*/)}], \\
&\quad\ [T \ ::= \cdot\ F,\ \texttt{+/*/)}], \\
&\quad\ [F \ ::= \cdot\ \texttt{(}E\texttt{)},\ \texttt{+/*/)}], \\
&\quad\ [F \ ::= \cdot\ \texttt{id},\ \texttt{+/*/)}]\}
\end{aligned}$$

$$\begin{aligned}
\text{goto}(s_0, \texttt{id}) &= s_5 \\
&= \{[F \ ::= \texttt{id}\ \cdot,\ \texttt{+/*/\#}]\}
\end{aligned}$$

We continue in this manner, computing goto for the states we have, and then for any new states repeatedly until we have defined no more new states; this gives us the canonical LR(1) collection

Algorithm Computing the LR(1) Collection

Input: a context-free grammar $G = (N, T, S, P)$
Output: the canonical LR(1) collection of states $c = \{s_0, s_1, \ldots, s_n\}$

Define an augmented grammar G' which is G with the added non-terminal S' and added production rule $S' ::= S$, where S is G's start symbol
$c \leftarrow \{s_0\}$ where $s_0 =$ closure($\{[S' ::= \cdot S,$ # $]\}$)
repeat
  for each s in c, and for each symbol $X \in T \cup N$ **do**
    if goto$(s, X) \neq \emptyset$ and goto$(s, X) \notin c$ **then**
      add goto(s, X) to c.
    end if
  end for
until no new states are added to c

We now resume computing the LR(1) canonical collection for our simple expression grammar, beginning from state s_1

$$\begin{aligned}\text{goto}(s_1, \texttt{+}) &= s_6 \\ &= \{[E \ ::= E \ \texttt{+} \cdot T, \texttt{+/\#}], \\ &\quad [T \ ::= \cdot \ T \ \texttt{*} \ F, \texttt{+/*/\#}], \\ &\quad [T \ ::= \cdot \ F, \texttt{+/*/\#}], \\ &\quad [F \ ::= \cdot \ \texttt{(}E\texttt{)}, \ \texttt{+/*/\#}], \\ &\quad [F \ ::= \cdot \ \texttt{id}, \texttt{+/*/\#}]\}\end{aligned}$$

There are no more moves from s_1; similarly, from s_2

$$\begin{aligned}\text{goto}(s_2, \texttt{*}) &= s_7 \\ &= \{[T \ ::= T \ \texttt{*} \cdot F, \texttt{+/*/\#}], \\ &\quad [F \ ::= \cdot \ \texttt{(}E\texttt{)}, \texttt{+/*/\#}], \\ &\quad [F \ ::= \cdot \ \texttt{id}, \texttt{+/*/\#}]\}\end{aligned}$$

Notice that the closure of $\{[T ::= T \ \texttt{*} \cdot F, \texttt{+/*/\#}]\}$ carries along the same lookaheads since no symbol follows the F in the right hand side

Bottom-up Deterministic Parsing

There are no gotos from s_3, but several from s_4

$$\begin{aligned}
\text{goto}(s_4,\, E) &= s_8 \\
&= \{[F \;::=\; \texttt{(}E \cdot \texttt{)},\; \texttt{+/*/\#}], \\
&\quad\; [E \;::=\; E \cdot \texttt{+}\; T,\; \texttt{+/)}]\}
\end{aligned}$$

$$\begin{aligned}
\text{goto}(s_4,\, T) &= s_9 \\
&= \{[E \;::=\; T \cdot,\; \texttt{+/)}], \\
&\quad\; [T \;::=\; T \cdot \texttt{*}\; F,\; \texttt{+/*/)}]\}
\end{aligned}$$

$$\begin{aligned}
\text{goto}(s_4,\, F) &= s_{10} \\
&= \{[T \;::=\; F \cdot,\; \texttt{+/*/)}]\}
\end{aligned}$$

$$\begin{aligned}
\text{goto}(s_4,\, \texttt{(}) &= s_{11} \\
&= \{[F \;::=\; \texttt{(} \cdot E\texttt{)},\; \texttt{+/*/)}], \\
&\quad\; [E \;::=\; \cdot\; E\; \texttt{+}\; T,\; \texttt{+/)}], \\
&\quad\; [E \;::=\; \cdot\; T,\; \texttt{+/)}], \\
&\quad\; [T \;::=\; \cdot\; T\; \texttt{*}\; F,\; \texttt{+/*/)}], \\
&\quad\; [T \;::=\; \cdot\; F,\; \texttt{+/*/)}], \\
&\quad\; [F \;::=\; \cdot\; \texttt{(}E\texttt{)},\; \texttt{+/*/)}], \\
&\quad\; [F \;::=\; \cdot\; \texttt{id},\; \texttt{+/*/)}]\}
\end{aligned}$$

Notice that s_{11} differs from s_4 in only the lookaheads for the first item

$$\begin{aligned}
\text{goto}(s_4,\, \texttt{id}) &= s_{12} \\
&= \{[F \;::=\; \texttt{id} \cdot,\; \texttt{+/*/)}]\}
\end{aligned}$$

Bottom-up Deterministic Parsing

There are no moves from s_5, so consider s_6

$$\begin{aligned}\text{goto}(s_6, T) &= s_{13} \\ &= \{[E \ ::= E \texttt{ + } T \cdot, \texttt{+/\#}], \\ &\quad\ [T \ ::= T \cdot \texttt{ * } F, \texttt{+/*/\#}]\}\end{aligned}$$

Now, goto(s_6, F) = $\{[T ::= F\cdot, \texttt{+/*/\#}]\}$ but that is s_3. goto(s_6, `(`) is closure($\{[F ::= \texttt{(} \cdot E), \texttt{+/*/\#}]\}$), but that is s_4; and goto(s_6, `id`) is s_5

$$\begin{aligned}&\text{goto}(s_6, F) = s_3 \\ &\text{goto}(s_6, \texttt{(}) = s_4 \\ &\text{goto}(s_6, \texttt{id}) = s_5\end{aligned}$$

Consider s_7, s_8, and s_9

$$\begin{aligned}\text{goto}(s_7, F) &= s_{14} \\ &= \{[T \ ::= T \texttt{ * } F \cdot, \texttt{+/*/\#}]\} \\ \text{goto}(s_7, \texttt{(}) &= s_4 \\ \text{goto}(s_7, \texttt{id}) &= s_5\end{aligned}$$

Bottom-up Deterministic Parsing

$$
\begin{aligned}
\text{goto}(s_8, \texttt{)}) &= s_{15} \\
&= \{[F \ ::= \texttt{(}E\texttt{)} \cdot, \ \texttt{+/*/\#}]\} \\
\text{goto}(s_8, \texttt{+}) &= s_{16} \\
&= \{[E \ ::= E \texttt{ + } \cdot T, \texttt{ +/)}], \\
&\quad [T \ ::= \cdot \ T \texttt{ * } F, \texttt{ +/*/)}], \\
&\quad [T \ ::= \cdot \ F, \texttt{ +/*/)}], \\
&\quad [F \ ::= \cdot \texttt{ (}E\texttt{)}, \texttt{ +/*/)}], \\
&\quad [F \ ::= \cdot \texttt{ id}, \texttt{ +/*/)}]\} \\
\text{goto}(s_9, \texttt{*}) &= s_{17} \\
&= \{[T \ ::= T \texttt{ * } \cdot F, \texttt{ +/*/)}], \\
&\quad [F \ ::= \cdot \texttt{ (}E\texttt{)}, \texttt{ +/*/)}], \\
&\quad [F \ ::= \cdot \texttt{ id}, \texttt{ +/*/)}]\}
\end{aligned}
$$

There are no moves from s_{10}, but several from s_{11}

$$
\begin{aligned}
\text{goto}(s_{11}, E) &= s_{18} \\
&= \{[F \ ::= \texttt{(}E \cdot\texttt{)}, \texttt{ +/*/)}], \\
&\quad [E \ ::= E \cdot \texttt{ + } T, \texttt{ +/)}]\} \\
\text{goto}(s_{11}, T) &= s_9 \\
\text{goto}(s_{11}, F) &= s_{10} \\
\text{goto}(s_{11}, \texttt{(}) &= s_{11} \\
\text{goto}(s_{11}, \texttt{id}) &= s_{12}
\end{aligned}
$$

There are no moves from s_{10}, but several from s_{11}

$$\begin{aligned}
\text{goto}(s_{11},\, E) &= s_{18} \\
&= \{[F \ ::= \texttt{(}E\ \cdot\texttt{)},\ \texttt{+/*/)}], \\
&\quad\ [E \ ::= E\ \cdot\ \texttt{+}\ T,\ \texttt{+/)}]\} \\
\text{goto}(s_{11},\, T) &= s_9 \\
\text{goto}(s_{11},\, F) &= s_{10} \\
\text{goto}(s_{11},\, \texttt{(}) &= s_{11} \\
\text{goto}(s_{11},\, \texttt{id}) &= s_{12}
\end{aligned}$$

There are no moves from s_{12}, but there is a move from s_{13}

$$\text{goto}(s_{13},\, \texttt{*}) = s_7$$

There are no moves from s_{14} or s_{15}, but there are moves from s_{16}, s_{17}, s_{18}, and s_{19}

$$
\begin{aligned}
\text{goto}(s_{16}, T) &= s_{19} \\
&= \{[E \ ::= E \texttt{ + } T \cdot, \texttt{+/)}], \\
&\quad\ [T \ ::= T \cdot \texttt{ * } F, \texttt{+/*/)}]\} \\
\text{goto}(s_{16}, F) &= s_{10} \\
\text{goto}(s_{16}, \texttt{(}) &= s_{11} \\
\text{goto}(s_{16}, \texttt{id}) &= s_{12} \\
\text{goto}(s_{17}, F) &= s_{20} \\
&= \{[T \ ::= T \texttt{ * } F \cdot, \texttt{+/*/)}]\} \\
\text{goto}(s_{17}, \texttt{(}) &= s_{11} \\
\text{goto}(s_{17}, \texttt{id}) &= s_{12} \\
\text{goto}(s_{18}, \texttt{)}) &= s_{21} \\
&= \{[F \ ::= \texttt{(}E\texttt{)} \cdot, \texttt{+/*/)}]\} \\
\text{goto}(s_{18}, \texttt{+}) &= s_{16} \\
\text{goto}(s_{19}, \texttt{*}) &= s_{17}
\end{aligned}
$$

There are no moves from s_{20} or s_{21}, so we are done; the LR(1) canonical collection consists of twenty-two states $s_0 \ldots s_{21}$, and is shown in the following slide

Bottom-up Deterministic Parsing

$s_0 = \{[E' ::= \cdot\, E, \#],$
$[E ::= \cdot\, E + T, +/\#],$
$[E ::= \cdot\, T, +/\#],$
$[T ::= \cdot\, T * F, +/*/\#],$
$[T ::= \cdot\, F, +/*/\#],$
$[F ::= \cdot (E), +/*/\#],$
$[F ::= \cdot\, \texttt{id}, +/*/\#]\}$

$\text{goto}(s_0, E) = s_1$
$\text{goto}(s_0, T) = s_2$
$\text{goto}(s_0, F) = s_3$
$\text{goto}(s_0, \texttt{(}) = s_4$
$\text{goto}(s_0, \texttt{id}) = s_5$

$s_1 = \{[E' ::= E \cdot, \#],$
$[E ::= E \cdot + T, +/\#]\}$

$\text{goto}(s_1, +) = s_6$

$s_2 = \{[E ::= T \cdot, +/\#],$
$[T ::= T \cdot * F, +/*/\#]\}$

$\text{goto}(s_2, *) = s_7$

$s_3 = \{[T ::= F \cdot, +/*/\#]\}$

$s_4 = \{[F ::= (\cdot\, E), +/*/\#],$
$[E ::= \cdot\, E + T, +/)],$
$[E ::= \cdot\, T, +/)],$
$[T ::= \cdot\, T * F, +/*/)],$
$[T ::= \cdot\, F, +/*/)],$
$[F ::= \cdot\, (E), +/*/)],$
$[F ::= \cdot\, \texttt{id}, +/*/)]\}$

$\text{goto}(s_4, E) = s_8$
$\text{goto}(s_4, T) = s_9$
$\text{goto}(s_4, F) = s_{10}$
$\text{goto}(s_4, \texttt{(}) = s_{11}$
$\text{goto}(s_4, \texttt{id}) = s_{12}$

$s_5 = \{[F ::= \texttt{id} \cdot, +/*/\#]\}$

$s_6 = \{[E ::= E + \cdot\, T, +/\#],$
$[T ::= \cdot\, T * F, +/*/\#],$
$[T ::= \cdot\, F, +/*/\#],$
$[F ::= \cdot\, (E), +/*/\#],$
$[F ::= \cdot\, \texttt{id}, +/*/\#]\}$

$\text{goto}(s_6, T) = s_{13}$
$\text{goto}(s_6, F) = s_3$
$\text{goto}(s_6, \texttt{(}) = s_4$
$\text{goto}(s_6, \texttt{id}) = s_5$

$s_7 = \{[T ::= T * \cdot\, F, +/*/\#],$
$[F ::= \cdot\, (E), +/*/\#],$
$[F ::= \cdot\, \texttt{id}, +/*/\#]\}$

$\text{goto}(s_7, F) = s_{14}$
$\text{goto}(s_7, \texttt{(}) = s_4$
$\text{goto}(s_7, \texttt{id}) = s_5$

$s_8 = \{[F ::= (E \cdot\,), +/*/\#],$
$[E ::= E \cdot + T, +/)]\}$

$\text{goto}(s_8, \texttt{)}) = s_{15}$
$\text{goto}(s_8, +) = s_{16}$

$s_9 = \{[E ::= T \cdot, +/)],$
$[T ::= T \cdot * F, +/*/)]\}$

$\text{goto}(s_9, *) = s_{17}$

$s_{10} = \{[T ::= F \cdot, +/*/)]\}$

$s_{11} = \{[F ::= (\cdot\, E), +/*/)],$
$[E ::= \cdot\, E + T, +/)],$
$[E ::= \cdot\, T, +/)],$
$[T ::= \cdot\, T * F, +/*/)],$
$[T ::= \cdot\, F, +/*/)],$
$[F ::= \cdot\, (E), +/*/)],$
$[F ::= \cdot\, \texttt{id}, +/*/)]\}$

$\text{goto}(s_{11}, E) = s_{18}$
$\text{goto}(s_{11}, T) = s_9$
$\text{goto}(s_{11}, F) = s_{10}$
$\text{goto}(s_{11}, \texttt{(}) = s_{11}$
$\text{goto}(s_{11}, \texttt{id}) = s_{12}$

$s_{12} = \{[F ::= \texttt{id} \cdot, +/*/)]\}$

$s_{13} = \{[E ::= E + T \cdot, +/\#],$
$[T ::= T \cdot * F, +/*/\#]\}$

$\text{goto}(s_{13}, *) = s_7$

$s_{14} = \{[T ::= T * F \cdot, +/*/\#]\}$

$s_{15} = \{[F ::= (E) \cdot, +/*/\#]\}$

$s_{16} = \{[E ::= E + \cdot\, T, +/)],$
$[T ::= \cdot\, T * F, +/*/)],$
$[T ::= \cdot\, F, +/*/)],$
$[F ::= \cdot\, (E), +/*/)],$
$[F ::= \cdot\, \texttt{id}, +/*/)]\}$

$\text{goto}(s_{16}, T) = s_{19}$
$\text{goto}(s_{16}, F) = s_{10}$
$\text{goto}(s_{16}, \texttt{(}) = s_{11}$
$\text{goto}(s_{16}, \texttt{id}) = s_{12}$

$s_{17} = \{[T ::= T * \cdot\, F, +/*/)],$
$[F ::= \cdot\, (E), +/*/)],$
$[F ::= \cdot\, \texttt{id}, +/*/)]\}$

$\text{goto}(s_{17}, F) = s_{20}$
$\text{goto}(s_{17}, \texttt{(}) = s_{11}$
$\text{goto}(s_{17}, \texttt{id}) = s_{12}$

$s_{18} = \{[F ::= (E \cdot), +/*/)],$
$[E ::= E \cdot + T, +/)]\}$

$\text{goto}(s_{18}, \texttt{)}) = s_{21}$
$\text{goto}(s_{18}, +) = s_{16}$

$s_{19} = \{[E ::= E + T \cdot, +/)],$
$[T ::= T \cdot * F, +/*/)]\}$

$\text{goto}(s_{19}, *) = s_{17}$

$s_{20} = \{[T ::= T * F \cdot, +/*/)]\}$

$s_{21} = \{[F ::= (E) \cdot, +/*/)]\}$

Algorithm Constructing the LR(1) Parsing Tables for a Context-Free Grammar

Input: a context-free grammar $G = (N, T, S, P)$
Output: the LF(1) tables Action and Goto

1. Compute the LR(1) canonical collection $c = \{s_0, s_1, \ldots, s_n\}$. State i of the parser corresponds to the item set s_i. State 0, corresponding to the item set s_0, which contains the item $[S' ::= \cdot S, \#]$, is the parser's initial state
2. The Action table is constructed as follows:
   a. For each transition, goto(s_i, a) $= s_j$, where a is a terminal, set Action[i, a] $= sj$. The s stands for "shift"
   b. If the item set s_k contains the item $[S' ::= S\cdot, \#]$, set Action[k, #] $=$ accept
   c. For all item sets s_i, if s_i contains an item of the form $[Y ::= \alpha\cdot, a]$, set Action[i, a] $= rp$, where p is the number corresponding to the rule $Y ::= \alpha$. The r stands for "reduce"
   d. All undefined entries in Action are set to error
3. The Goto table is constructed as follows:
   a. For each transition, goto(s_i, Y) $= s_j$, where Y is a non-terminal, set Goto[i, Y] $= j$.
   b. All undefined entries in Goto are set to error

If all entries in the Action table are unique then the grammar G is said to be LR(1)

Bottom-up Deterministic Parsing

The Action and Goto tables for our grammar for simple expressions are shown below

	Action						Goto		
	+	*	(	)	id	#	E	T	F
0			s4		s5		1	2	3
1	s6					accept			
2	r2	s7				r2			
3	r4	r4				r4			
4			s11		s12		8	9	10
5	r6	r6				r6			
6			s4		s5			13	3
7			s4		s5				14
8	s16			s15					
9	r2	s17		r2					
10	r4	r4		r4					
11			s11		s12		18	9	10
12	r6	r6		r6					
13	r1	s7				r1			
14	r3	r3				r3			
15	r5	r5				r5			
16			s11		s12			19	10
17			s11		s12				20
18	s16			s21					
19	r1	s17		r1					
20	r3	r3		r3					
21	r5	r5		r5					

Bottom-up Deterministic Parsing

There are two different kinds of conflicts possible for an entry in the Action table

The first is the shift-reduce conflict, which can occur when there are items of the forms

$[Y ::= \alpha \cdot, \texttt{a}]$ and
$[Y ::= \alpha \cdot \texttt{a} \beta, \texttt{b}]$

The first item suggests a reduce if the next un-scanned token is an `a`; the second suggests a shift of the `a` onto the stack

Although such conflicts may occur for unambiguous grammars, a common cause are ambiguous constructs such as

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

Most parser generators that are based on LR grammars permit one to supply an extra disambiguating rule; for example, to favor a shift of the `else` over a reduce of the `if (`E`)` S to an S

Bottom-up Deterministic Parsing

The second kind of conflict that we can have is the reduce-reduce conflict, which can happen when we have a state containing two items of the form

$$[X ::= \alpha \cdot, \mathtt{a}]$$
$$[Y ::= \beta \cdot, \mathtt{a}]$$

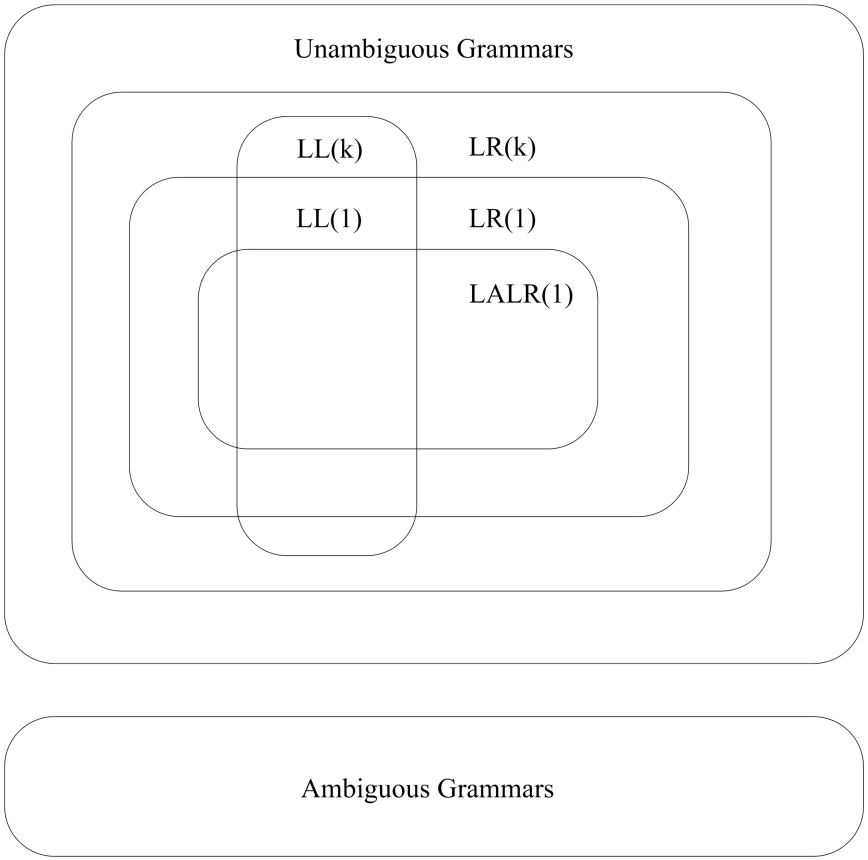
Here, the parser cannot distinguish which production rule to apply in the reduction

The LR(1) parsing table for a typical programming language such as Java can have thousands of states, and so thousands of rows

The LALR(1) parsing algorithm (see our text) is capable of reducing the number of states by an order of magnitude

Bottom-up Deterministic Parsing

The following figure illustrates the relationships among the various categories of grammars we have been discussing



Parser Generation Using JavaCC

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

Parser Generation Using JavaCC

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

Parser Generation Using JavaCC

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() |
```

Parser Generation Using JavaCC

```
        <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; }
}
```

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

For example, 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); } |
```

Parser Generation Using JavaCC

```
        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; }
}
```

Parser Generation Using JavaCC

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

Parser Generation Using JavaCC

```
    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);
    }
```