Parsing

Outline

1. Parsing a Program
2. Context-free Grammars and Languages
3. Top-down Deterministic Parsing
4. Recursive Descent Parsing
5. LL(1) Parsing
6. Bottom-up Deterministic Parsing
7. LR(1) Parsing
8. JavaCC

Parsing a Program

Parsing a Program

The process of parsing a program is to determine its syntactic structure

Parsing a Program

The process of parsing a program is to determine its syntactic structure

A parser should:

Parsing a Program

The process of parsing a program is to determine its syntactic structure

A parser should:

- Make sure the program is syntactically valid, ie, conforms to the grammar describing the program's syntax

Parsing a Program

The process of parsing a program is to determine its syntactic structure

A parser should:

- Make sure the program is syntactically valid, ie, conforms to the grammar describing the program's syntax
- Identify syntax errors and report them along with the line numbers they appear on

Parsing a Program

The process of parsing a program is to determine its syntactic structure

A parser should:

- Make sure the program is syntactically valid, ie, conforms to the grammar describing the program's syntax
- Identify syntax errors and report them along with the line numbers they appear on
- Not stop on the first error, but report the error, and gracefully recover and look for additional errors

Parsing a Program

The process of parsing a program is to determine its syntactic structure

A parser should:

- Make sure the program is syntactically valid, ie, conforms to the grammar describing the program's syntax
- Identify syntax errors and report them along with the line numbers they appear on
- Not stop on the first error, but report the error, and gracefully recover and look for additional errors
- Produce a representation of the parsed program that is suitable for semantic analysis; in *j--*, the representation is an abstract syntax tree (AST)

Parsing a Program

Parsing a Program

HelloWorld.java

```java
// Copyright 2012- Bill Campbell, Swami Iyer and Bahar Akbal-Delibas
//
// Writes to standard output the message "Hello, World".

import java.lang.System;

public class HelloWorld {
    // Entry point.
    public static void main(String[] args) {
        System.out.println("Hello, World");
    }
}
```

Parsing a Program

Parsing a Program

```
{
    "JCompilationUnit:5":
    {
        "source": "tests/jvm/HelloWorld.java",
        "imports": ["java.lang.System"],
        "JClassDeclaration:7":
        {
            "modifiers": ["public"],
            "name": "HelloWorld",
            "super": "java.lang.Object",
            "JMethodDeclaration:9":
            {
                "name": "main",
                "returnType": "void",
                "modifiers": ["public", "static"],
                "parameters": [["args", "String[]"]],
                "JBlock:9":
                {
                    "JStatementExpression:10":
                    {
                        "JMessageExpression:10":
                        {
                            "ambiguousPart": "System.out", "name": "println",
                            "Argument":
                            {
                                "JLiteralString:10":
                                {
                                    "type": "", "value": "Hello, World"
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}
```

Parsing a Program

Parsing a Program

The nodes in the AST represent syntactic objects

Parsing a Program

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

Parsing a Program

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

Parsing a Program

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 of imported types, and a list of type declarations

Parsing a Program

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 of imported types, and a list of type declarations

The tree representation for a program is easier to analyze and decorate (with type information) than text

Parsing a Program

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 of imported types, and a list of type declarations

The tree representation for a program is easier to analyze and decorate (with type information) than text

The AST makes the syntax implicit in the program text, explicit

Context-free Grammars and Languages

Context-free Grammars and Languages

Inherently recursive programming languages such as *j--* are best described by context-free grammar rules, using a notation called Backus-Naur Form (BNF)

Inherently recursive programming languages such as *j*-- are best described by context-free grammar rules, using a notation called Backus-Naur Form (BNF)

For example, the 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

Context-free Grammars and Languages

There are abbreviations possible in the BNF notation

There are abbreviations possible in the BNF notation

For example, the rule

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

is shorthand for

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

There are abbreviations possible in the BNF notation

For example, the rule

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

is shorthand for

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

Square brackets indicate that a phrase is optional

There are abbreviations possible in the BNF notation

For example, the rule

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

is shorthand for

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

Square brackets indicate that a phrase is optional

For example, the two rules from above can be written as

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

Context-free Grammars and Languages

Curly braces denote the Kleene closure, indicating that the phrase may appear zero or more times

Curly braces denote the Kleene closure, indicating that the phrase may appear zero or more times

For example, the rule

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

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

$$T + T + T + T$$

Context-free Grammars and Languages

One may use the alternation sign $|$ to denote a choice, and parentheses for grouping

One may use the alternation sign | to denote a choice, and parentheses for grouping

For example, the rule

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

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

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

Context-free Grammars and Languages

Example (BNF rules in *j--*)

```
compilationUnit ::= [ PACKAGE qualifiedIdentifier SEMI ]
                    { IMPORT  qualifiedIdentifier SEMI }
                    { typeDeclaration }
                    EOF

qualifiedIdentifier ::= IDENTIFIER { DOT IDENTIFIER }

typeDeclaration ::= modifiers classDeclaration

modifiers ::= { ABSTRACT | PRIVATE | PROTECTED | PUBLIC | STATIC }

classDeclaration ::= CLASS IDENTIFIER [ EXTENDS qualifiedIdentifier ] classBody

classBody ::= LCURLY { modifiers memberDecl } RCURLY
```

Context-free Grammars and Languages

Context-free Grammars and Languages

A context-free grammar is a tuple $G = (N, T, S, P)$, where

Context-free Grammars and Languages

A context-free grammar is a tuple $G = (N, T, S, P)$, where

- N is a set non-terminals

Context-free Grammars and Languages

A context-free grammar is a tuple $G = (N, T, S, P)$, where

- N is a set non-terminals
- T is a set of terminals

Context-free Grammars and Languages

A context-free grammar is a tuple $G = (N, T, S, P)$, where

- N is a set non-terminals
- T is a set of terminals
- $S \in N$ is the start symbol

Context-free Grammars and Languages

A context-free grammar is a tuple $G = (N, T, S, P)$, where

- N is a set non-terminals
- T is a set of terminals
- $S \in N$ is the start symbol
- P is a set of productions or rules

Context-free Grammars and Languages

A context-free grammar is a tuple $G = (N, T, S, P)$, where

- N is a set non-terminals
- T is a set of terminals
- $S \in N$ is the start symbol
- P is a set of productions or rules

Example (arithmetic expression grammar)

$G = (N, T, S, P)$ where $N = \{E, T, F\}$, $T = \{\texttt{+}, \texttt{*}, \texttt{(}, \texttt{)}, \texttt{id}\}$, $S = E$, and $P = \{E ::= E \texttt{ + } T$, $E ::= T, T ::= T \texttt{ * } F, T ::= F, F ::= \texttt{(}E\texttt{)}, F ::= \texttt{id}\}$

Context-free Grammars and Languages

A context-free grammar is a tuple $G = (N, T, S, P)$, where

- N is a set non-terminals
- T is a set of terminals
- $S \in N$ is the start symbol
- P is a set of productions or rules

Example (arithmetic expression grammar)

$G = (N, T, S, P)$ where $N = \{E, T, F\}$, $T = \{\texttt{+}, \texttt{*}, \texttt{(}, \texttt{)}, \texttt{id}\}$, $S = E$, and $P = \{E ::= E \texttt{ + } T$, $E ::= T, T ::= T \texttt{ * } F, T ::= F, F ::= \texttt{(}E\texttt{)}, F ::= \texttt{id}\}$

A grammar can be specified informally as a sequence of productions

Context-free Grammars and Languages

Context-free Grammars and Languages

From the start symbol, using productions, we can generate strings in a language

From the start symbol, using productions, we can generate strings in a language

Example

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

From the start symbol, using productions, we can generate strings in a language

Example

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

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

Context-free Grammars and Languages

From the start symbol, using productions, we can generate strings in a language

Example

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

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 ($\stackrel{*}{\Rightarrow}$) the second string

Example

$$
\begin{aligned}
&E \stackrel{*}{\Rightarrow} E \text{ (in zero steps)} \\
&E \stackrel{*}{\Rightarrow} \texttt{id} + F * F \\
&T + T \stackrel{*}{\Rightarrow} \texttt{id} + \texttt{id} * \texttt{id}
\end{aligned}
$$

Context-free Grammars and Languages

The language $L(G)$ described by a grammar G consists of all the strings comprised of only terminal symbols, ie,
$L(G) = \{w | S \overset{*}{\Rightarrow} w \text{ and } w \in T*\}$

The language $L(G)$ described by a grammar G consists of all the strings comprised of only terminal symbols, ie, $L(G) = \{w | S \stackrel{*}{\Rightarrow} w \text{ and } w \in T*\}$

For example, in the arithmetic expression grammar G

$$E \stackrel{*}{\Rightarrow} \texttt{id}$$
$$E \stackrel{*}{\Rightarrow} \texttt{id + id * id}$$
$$E \stackrel{*}{\Rightarrow} \texttt{(id + id) * id}$$

so, $L(G)$ includes each of

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

and infinitely more finite strings

Context-free Grammars and Languages

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

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

Example

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

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 for rewriting the right-most non-terminal

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

Example

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

Context-free Grammars and Languages

A sentential form refers to any string of terminal and non-terminal symbols that can be derived from the start symbol, and a sentence is a string with only terminal symbols

A sentential form refers to any string of terminal and non-terminal symbols that can be derived from the start symbol, and a sentence is a string with only terminal symbols

For example,

E
$E + T$
$E + T * F$
...
F `+ id * id`
`id + id * id`

are all sentential forms, and `id + id * id` is a sentence

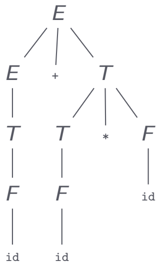
Context-free Grammars and Languages

A parse tree illustrates the derivation and the structure of an input string (at the leaves) from a start symbol (at the root)

Context-free Grammars and Languages

A parse tree illustrates the derivation and the structure of an input string (at the leaves) from a start symbol (at the root)

Example (parse tree for `id + id * id`)



Context-free Grammars and Languages

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

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

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

Context-free Grammars and Languages

Context-free Grammars and Languages

Example (ambiguous arithmetic expression grammar)

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

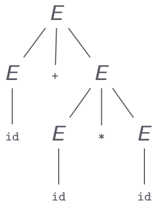
Context-free Grammars and Languages

Example (ambiguous arithmetic expression grammar)

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

A left-most derivation and corresponding parse tree for the sentence `id + id * id`

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

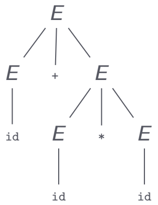


Example (ambiguous arithmetic expression grammar)

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

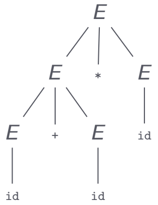
A left-most derivation and corresponding parse tree for the sentence $\texttt{id + id * id}$

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



Another left-most derivation and corresponding parse tree for $\texttt{id + id * id}$

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



Context-free Grammars and Languages

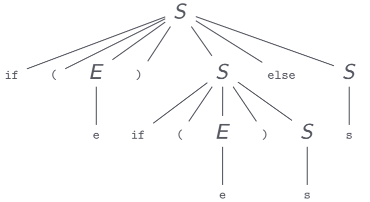
Context-free Grammars and Languages

Example (dangling-else problem)

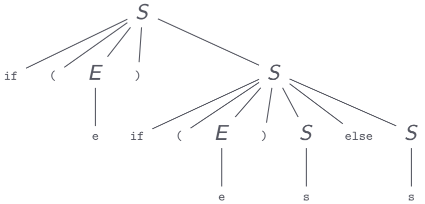
$$
\begin{aligned}
S &::= \texttt{if (}E\texttt{)}\ S \\
&\mid \texttt{if (}E\texttt{)}\ S\ \texttt{else}\ S \\
&\mid \texttt{s} \\
E &::= \texttt{e}
\end{aligned}
$$

Two left-most derivations and corresponding parse trees for the sentence `if (e) if (e) s else s`

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



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



Context-free Grammars and Languages

Resolving the dangling-else problem

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

Resolving the dangling-else problem

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

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

Resolving the dangling-else problem

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

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

Compiler writers handle the rule as a special case in the parser such that an `else` is grouped along with the closest preceding `if`

Context-free Grammars and Languages

Context-free Grammars and Languages

j-- has another ambiguity, which is the problem of parsing the expression `x.y.z.w`

Context-free Grammars and Languages

j-- has another ambiguity, which is the problem of parsing the expression `x.y.z.w`

Clearly `w` is a field, but what about `x.y.z`?

j-- has another ambiguity, which is the problem of parsing the expression `x.y.z.w`

Clearly `w` is a field, but what about `x.y.z`?

`x` might refer to an object with a field `y`, referring to another object with a field `z`, referring to the field `w`

j-- has another ambiguity, which is the problem of parsing the expression `x.y.z.w`

Clearly `w` is a field, but what about `x.y.z`?

`x` might refer to an object with a field `y`, referring to another object with a field `z`, referring to the field `w`

`x.y` might be a package in which the class `z` is defined, and `w` a static field in that class

j-- has another ambiguity, which is the problem of parsing the expression `x.y.z.w`

Clearly `w` is a field, but what about `x.y.z`?

`x` might refer to an object with a field `y`, referring to another object with a field `z`, referring to the field `w`

`x.y` might be a package in which the class `z` is defined, and `w` a static field in that class

The parser cannot determine how the expression `x.y.z` is parsed because types are not decided until semantic analysis

j-- has another ambiguity, which is the problem of parsing the expression `x.y.z.w`

Clearly `w` is a field, but what about `x.y.z`?

`x` might refer to an object with a field `y`, referring to another object with a field `z`, referring to the field `w`

`x.y` might be a package in which the class `z` is defined, and `w` a static field in that class

The parser cannot determine how the expression `x.y.z` is parsed because types are not decided until semantic analysis

The parser represents `x.y.z` in the AST as an `AmbiguousName` node, which gets reclassified during semantic analysis

Top-down Deterministic Parsing

Top-down Deterministic Parsing

Top-down parsing algorithms scan the input from left to right, looking at and scanning just one symbol at a time

Top-down Deterministic Parsing

Top-down parsing algorithms 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, which is then rewritten using a rule replacing the symbol with the right-hand-side sequence of symbols

Top-down Deterministic Parsing

Top-down parsing algorithms 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, which is then rewritten using a rule replacing the symbol with the right-hand-side sequence of symbols

Example (compilation unit in *j--*)

```
compilationUnit ::= [ PACKAGE qualifiedIdentifier SEMI ]
                    { IMPORT  qualifiedIdentifier SEMI }
                    { typeDeclaration }
                    EOF
```

Top-down parsing algorithms 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, which is then rewritten using a rule replacing the symbol with the right-hand-side sequence of symbols

Example (compilation unit in *j--*)

```
compilationUnit ::= [ PACKAGE qualifiedIdentifier SEMI ]
                    { IMPORT  qualifiedIdentifier SEMI }
                    { typeDeclaration }
                    EOF
```

The goal of parsing a `compilationUnit` can be rewritten as a number of sub-goals:

Top-down Deterministic Parsing

Top-down parsing algorithms 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, which is then rewritten using a rule replacing the symbol with the right-hand-side sequence of symbols

Example (compilation unit in *j--*)

```
compilationUnit ::= [ PACKAGE qualifiedIdentifier SEMI ]
                    { IMPORT  qualifiedIdentifier SEMI }
                    { typeDeclaration }
                    EOF
```

The goal of parsing a `compilationUnit` can be rewritten as a number of sub-goals:

1. If there is a package statement in the input sentence, then parse that

Top-down parsing algorithms 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, which is then rewritten using a rule replacing the symbol with the right-hand-side sequence of symbols

Example (compilation unit in *j--*)

```
compilationUnit ::= [ PACKAGE qualifiedIdentifier SEMI ]
                    { IMPORT  qualifiedIdentifier SEMI }
                    { typeDeclaration }
                    EOF
```

The goal of parsing a `compilationUnit` can be rewritten as a number of sub-goals:

1. If there is a package statement in the input sentence, then parse that
2. If there are import statements in the input, then parse them

Top-down Deterministic Parsing

Top-down parsing algorithms 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, which is then rewritten using a rule replacing the symbol with the right-hand-side sequence of symbols

Example (compilation unit in *j--*)

```
compilationUnit ::= [ PACKAGE qualifiedIdentifier SEMI ]
                    { IMPORT  qualifiedIdentifier SEMI }
                    { typeDeclaration }
                    EOF
```

The goal of parsing a `compilationUnit` can be rewritten as a number of sub-goals:

1. If there is a package statement in the input sentence, then parse that
2. If there are import statements in the input, then parse them
3. If there are any type declarations, then parse them

Top-down Deterministic Parsing

Top-down parsing algorithms 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, which is then rewritten using a rule replacing the symbol with the right-hand-side sequence of symbols

Example (compilation unit in *j--*)

```
compilationUnit ::= [ PACKAGE qualifiedIdentifier SEMI ]
                    { IMPORT  qualifiedIdentifier SEMI }
                    { typeDeclaration }
                    EOF
```

The goal of parsing a `compilationUnit` can be rewritten as a number of sub-goals:

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

Top-down Deterministic Parsing

Top-down Deterministic Parsing

Parsing a token, like PACKAGE, is simple; if we see it, we simply scan it

Top-down Deterministic Parsing

Parsing a token, like PACKAGE, is simple; if we see it, we simply scan it

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

Parsing a token, like PACKAGE, is simple; if we see it, we simply scan it

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

For example, in a package statement, once we scan the PACKAGE token, we are left with parsing a `qualifiedIdentifier`

```
qualifiedIdentifier ::= IDENTIFIER { DOT IDENTIFIER }
```

We scan an IDENTIFIER and so long as we see a DOT in the input, we scan the DOT and scan another IDENTIFIER

Top-down Deterministic Parsing

We decide which rule to apply by looking at the next un-scanned input token

Top-down Deterministic Parsing

We decide which rule to apply by looking at the next un-scanned input token

Example (statements in *j*--)

```
statement ::= block
            | IF parExpression statement [ ELSE statement ]
            | WHILE parExpression statement
            | RETURN [ expression ] SEMI
            | SEMI
            | statementExpression SEMI
```

Top-down Deterministic Parsing

We decide which rule to apply by looking at the next un-scanned input token

Example (statements in *j--*)

```
statement ::= block
            | IF parExpression statement [ ELSE statement ]
            | WHILE parExpression statement
            | RETURN [ expression ] SEMI
            | SEMI
            | statementExpression SEMI
```

1. If the next token is a `{`, then parse a block

Top-down Deterministic Parsing

We decide which rule to apply by looking at the next un-scanned input token

Example (statements in *j--*)

```
statement ::= block
            | IF parExpression statement [ ELSE statement ]
            | WHILE parExpression statement
            | RETURN [ expression ] SEMI
            | SEMI
            | statementExpression SEMI
```

1. If the next token is a `{`, then parse a block
2. If the next token is an `IF`, then parse an if statement

Top-down Deterministic Parsing

We decide which rule to apply by looking at the next un-scanned input token

Example (statements in *j--*)

```
statement ::= block
            | IF parExpression statement [ ELSE statement ]
            | WHILE parExpression statement
            | RETURN [ expression ] SEMI
            | SEMI
            | statementExpression SEMI
```

1. If the next token is a `{`, then parse a block
2. If the next token is an `IF`, then parse an if statement
3. If the next token is a `WHILE`, then parse a while statement

Top-down Deterministic Parsing

We decide which rule to apply by looking at the next un-scanned input token

Example (statements in *j*--)

```
statement ::= block
            | IF parExpression statement [ ELSE statement ]
            | WHILE parExpression statement
            | RETURN [ expression ] SEMI
            | SEMI
            | statementExpression SEMI
```

1. If the next token is a `{`, then parse a block
2. If the next token is an IF, then parse an if statement
3. If the next token is a WHILE, then parse a while statement
4. If the next token is a RETURN, then parse a return statement

Top-down Deterministic Parsing

We decide which rule to apply by looking at the next un-scanned input token

Example (statements in *j--*)

```
statement ::= block
            | IF parExpression statement [ ELSE statement ]
            | WHILE parExpression statement
            | RETURN [ expression ] SEMI
            | SEMI
            | statementExpression SEMI
```

1. If the next token is a `{`, then parse a block
2. If the next token is an IF, then parse an if statement
3. If the next token is a WHILE, then parse a while statement
4. If the next token is a RETURN, then parse a return statement
5. If the next token is a semicolon, then parse an empty statement

We decide which rule to apply by looking at the next un-scanned input token

Example (statements in *j*--)

```
statement ::= block
            | IF parExpression statement [ ELSE statement ]
            | WHILE parExpression statement
            | RETURN [ expression ] SEMI
            | SEMI
            | statementExpression SEMI
```

1. If the next token is a `{`, then parse a block
2. If the next token is an `IF`, then parse an if statement
3. If the next token is a `WHILE`, then parse a while statement
4. If the next token is a `RETURN`, then parse a return statement
5. If the next token is a semicolon, then parse an empty statement
6. Otherwise, parse a `statementExpression`

Top-down Deterministic Parsing

Top-down Deterministic Parsing

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

Top-down Deterministic Parsing

That we start at the start symbol, and continually rewrite non-terminals using rules until we eventually reach leaves (ie, tokens) 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 call this a goal-oriented parsing technique

Top-down Deterministic Parsing

That we start at the start symbol, and continually rewrite non-terminals using rules until we eventually reach leaves (ie, tokens) 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 call this a goal-oriented parsing technique

In some cases, one must lookahead several tokens in the input to decide which rule to apply

Top-down Deterministic Parsing

That we start at the start symbol, and continually rewrite non-terminals using rules until we eventually reach leaves (ie, tokens) 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 call this a goal-oriented parsing technique

In some cases, one must lookahead several tokens in the input to decide which rule to apply

In all cases, since we can predict which rule to apply, based on the next input token(s), we say this is a predictive parsing technique

Recursive Descent Parsing

Recursive Descent Parsing

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

Recursive Descent Parsing

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

Based on the next input token, the method chooses a rule to apply, scans any terminals, and parses any non-terminals by recursively invoking the corresponding methods

Recursive Descent Parsing

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

Based on the next input token, the method chooses a rule to apply, scans any terminals, and parses any non-terminals by recursively invoking the corresponding methods

This is the strategy we use in the hand-crafted parser (`Parser.java`) for *j--*

Recursive Descent Parsing

Recursive Descent Parsing

Example (parsing a compilation unit)

```
compilationUnit ::= [ PACKAGE qualifiedIdentifier SEMI ]
                    { IMPORT  qualifiedIdentifier SEMI }
                    { typeDeclaration }
                    EOF
```

Example (parsing a compilation unit)

```
compilationUnit ::= [ PACKAGE qualifiedIdentifier SEMI ]
                    { IMPORT  qualifiedIdentifier SEMI }
                    { typeDeclaration }
                    EOF
```

Parser.java

```java
    public JCompilationUnit compilationUnit() {
        int line = scanner.token().line();
        String fileName = scanner.fileName();
        TypeName packageName = null;
        if (have(PACKAGE)) {
            packageName = qualifiedIdentifier();
            mustBe(SEMI);
        }
        ArrayList<TypeName> imports = new ArrayList<TypeName>();
        while (have(IMPORT)) {
            imports.add(qualifiedIdentifier());
            mustBe(SEMI);
        }
        ArrayList<JAST> typeDeclarations = new ArrayList<JAST>();
        while (!see(EOF)) {
            JAST typeDeclaration = typeDeclaration();
            if (typeDeclaration != null) {
                typeDeclarations.add(typeDeclaration);
            }
        }
        mustBe(EOF);
        return new JCompilationUnit(fileName, line, packageName, imports, typeDeclarations);
    }
```

Recursive Descent Parsing

Example (parsing a qualified identifier)

```
qualifiedIdentifier ::= IDENTIFIER { DOT IDENTIFIER }
```

Example (parsing a qualified identifier)

```
qualifiedIdentifier ::= IDENTIFIER { DOT IDENTIFIER }
```

Parser.java

```java
    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);
    }
```

Recursive Descent Parsing

Recursive Descent Parsing

`have()` looks at the next input token, and if that token matches its argument, then it scans the token and returns `true`; otherwise, it scans nothing and returns `false`

Recursive Descent Parsing

`have()` looks at the next input token, and if that token matches its argument, then it scans the token and returns `true`; otherwise, it scans nothing and returns `false`

`see()` looks at the next input token and returns `true` if that token matches its argument, and `false` otherwise

Recursive Descent Parsing

`have()` looks at the next input token, and if that token matches its argument, then it scans the token and returns `true`; otherwise, it scans nothing and returns `false`

`see()` looks at the next input token and returns `true` if that token matches its argument, and `false` otherwise

`mustBe()` requires that the next input token match its argument; on a match, it scans the token, and raises an error otherwise

Recursive Descent Parsing

`have()` looks at the next input token, and if that token matches its argument, then it scans the token and returns `true`; otherwise, it scans nothing and returns `false`

`see()` looks at the next input token and returns `true` if that token matches its argument, and `false` otherwise

`mustBe()` requires that the next input token match its argument; on a match, it scans the token, and raises an error otherwise

`mustBe()` also implements error recovery

Recursive Descent Parsing

Recursive Descent Parsing

Example (parsing a statement)

```
statement ::= block
            | IF parExpression statement [ ELSE statement ]
            | WHILE parExpression statement
            | RETURN [ expression ] SEMI
            | SEMI
            | statementExpression SEMI
```

Recursive Descent Parsing

Recursive Descent Parsing

Parser.java

```java
    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);
        } 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 {
            JStatement statement = statementExpression();
            mustBe(SEMI);
            return statement;
        }
    }
```

Recursive Descent Parsing

Recursive Descent Parsing

Sometimes we must look ahead in the input stream of tokens to decide which rule to apply

Recursive Descent Parsing

Sometimes we must look ahead in the input stream of tokens to decide which rule to apply

Example (parsing a simple unary expression)

```
simpleUnaryExpression ::= LNOT unaryExpression
                        | LPAREN basicType RPAREN unaryExpression
                        | LPAREN referenceType RPAREN simpleUnaryExpression
                        | postfixExpression
```

Recursive Descent Parsing

Sometimes we must look ahead in the input stream of tokens to decide which rule to apply

Example (parsing a simple unary expression)

```
simpleUnaryExpression ::= LNOT unaryExpression
                        | LPAREN basicType RPAREN unaryExpression
                        | LPAREN referenceType RPAREN simpleUnaryExpression
                        | postfixExpression
```

Parser.java

```java
    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();
        }
    }

    private boolean seeBasicType() {
        return (see(BOOLEAN) || see(CHAR) || see(INT));
    }
```

Recursive Descent Parsing

Parser.java

```
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();
        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;
}
```

Recursive Descent Parsing

Recursive Descent Parsing

The parser scans using `LookaheadScanner` which encapsulates `Scanner`

Recursive Descent Parsing

The parser scans using `LookaheadScanner` which encapsulates `Scanner`

`LookaheadScanner` defines `recordPosition()` for marking a position in the input stream, and `returnToPosition()` for returning the scanner to that recorded position (ie, for backtracking)

Recursive Descent Parsing

When `mustBe()` comes across a token that it is not expecting, we have a syntax error

When `mustBe()` comes across a token that it is not expecting, we have a syntax error

The parser should report the error and continue parsing so that it might detect any additional syntax errors

When `mustBe()` comes across a token that it is not expecting, we have a syntax error

The parser should report the error and continue parsing so that it might detect any additional syntax errors

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

Recursive Descent Parsing

When `mustBe()` comes across a token that it is not expecting, we have a syntax error

The parser should report the error and continue parsing so that it might detect any additional syntax errors

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

In the *j*-- parser, we implement limited error recovery in `mustBe()`

Recursive Descent Parsing

Recursive Descent Parsing

Parser.java

```java
    private 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 {
            while (!see(sought) && !see(EOF)) {
                scanner.next();
            }
            if (see(sought)) {
                scanner.next();
                isRecovered = true;
            }
        }
    }

    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;
        }
    }
```

LL(1) Parsing

LL(1) Parsing

The first L indicates a left-to-right scan of the input; the second L signifies that it produces a left-most derivation; and the 1 indicates a single lookahead

LL(1) Parsing

The first L indicates a left-to-right scan of the input; the second L signifies that it produces a left-most derivation; and the 1 indicates a single lookahead

At the start, the start symbol S is pushed onto a stack, and based on the first input symbol, S is replaced by the right-hand-side of a rule defining S

LL(1) Parsing

The first L indicates a left-to-right scan of the input; the second L signifies that it produces a left-most derivation; and the 1 indicates a single lookahead

At the start, the start symbol S is pushed onto a stack, and based on the first input symbol, S is replaced by the right-hand-side of a rule defining S

The parser continues by parsing each symbol as it is removed from the top of the stack:

LL(1) Parsing

The first L indicates a left-to-right scan of the input; the second L signifies that it produces a left-most derivation; and the 1 indicates a single lookahead

At the start, the start symbol S is pushed onto a stack, and based on the first input symbol, S is replaced by the right-hand-side of a rule defining S

The parser continues by parsing each symbol as it is removed from the top of the stack:

- If the symbol is a terminal, it scans a terminal from the input; if they do not match, an error is raised

LL(1) Parsing

The first L indicates a left-to-right scan of the input; the second L signifies that it produces a left-most derivation; and the 1 indicates a single lookahead

At the start, the start symbol S is pushed onto a stack, and based on the first input symbol, S is replaced by the right-hand-side of a rule defining S

The parser continues by parsing each symbol as it is removed from the top of the stack:

- If the symbol is a terminal, it scans a terminal from the input; if they do not match, an error is raised
- If the symbol is a non-terminal, the input symbol is used to decide which rule to apply to replace that non-terminal

LL(1) Parsing

LL(1) Parsing

LL(1) parsing technique is table-driven, with a unique parse table produced for each grammar

LL(1) Parsing

LL(1) parsing technique is table-driven, with a unique parse table produced for each grammar

The parse table has a row for each non-terminal and a column for each terminal, including a special terminator # to mark the end of the sentence

LL(1) Parsing

LL(1) parsing technique is table-driven, with a unique parse table produced for each grammar

The parse table has a row for each non-terminal and a column for each terminal, including a special terminator `#` to mark the end of the sentence

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

LL(1) Parsing

LL(1) parsing technique is table-driven, with a unique parse table produced for each grammar

The parse table has a row for each non-terminal and a column for each terminal, including a special terminator `#` to mark the end of the sentence

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

No table entry may contain more than one rule

LL(1) Parsing

Example (arithmetic expression grammar redux)

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

LL(1) Parsing

Example (arithmetic expression grammar redux)

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

LL(1) parse table for the grammar

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

LL(1) Parsing

Algorithm LL(1) parsing algorithm

Input: LL(1) parse table *table*, productions *rules*, and a sentence w followed by #

Output: a left-most derivation for w

```
stk ← Stack(#, S)
sym ← first symbol in w#
while true do
    top ← stk.pop()
    if top = sym = # then
        Halt successfully
    else if top is a terminal then
        if top = sym then
            Advance sym to be the next symbol in w#
        else
            Halt with an error: sym found where 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]
            If Y ::= X1 X2 ... Xn−1 Xn, then stk.push(Xn, Xn−1, ..., X2, X1)
        else
            Halt with an error: no rule to follow
        end if
    end if
end while
```

LL(1) Parsing

LL(1) Parsing

Example (parsing `id+id*id`)

LL(1) Parsing

Example (parsing `id+id*id`)

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

LL(1) Parsing

Example (parsing `id+id*id`)

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. $F \ ::= \mathtt{id}$

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

LL(1) Parsing

Example (parsing `id+id*id`)

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	

Stack	Input	Output
$\#E$	id+id*id#	
$\#E'T$	id+id*id#	1
$\#E'T'F$	id+id*id#	4
$\#E'T'\texttt{id}$	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
$\#E'T'\texttt{id}$	id*id#	8
$\#E'T'$	*id#	
$\#E'T'F*$	*id#	5
$\#E'T'F$	id#	
$\#E'T'\texttt{id}$	id#	8
$\#E'T'$	#	6
$\#E'$	#	3
#	#	✓

LL(1) Parsing

LL(1) Parsing

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 $Y ::= X_1X_2 \ldots X_n$, if either:

LL(1) Parsing

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 $Y ::= X_1X_2 \ldots X_n$, if either:

1. $X_1X_2 \ldots X_n \stackrel{*}{\Rightarrow} a\alpha$, or

LL(1) Parsing

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 $Y ::= X_1 X_2 \dots X_n$, if either:

1. $X_1 X_2 \dots X_n \overset{*}{\Rightarrow} a\alpha$, or
2. $X_1 X_2 \dots X_n \overset{*}{\Rightarrow} \epsilon$, and there is a derivation $S_\# \overset{*}{\Rightarrow} \alpha Y a \beta$, ie, a can follow Y in a derivation

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 $S\# \stackrel{*}{\Rightarrow} \alpha Y a \beta$, ie, a can follow Y in a derivation

For this we need two helper functions, first and follow

LL(1) Parsing

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 $Y ::= X_1X_2 \ldots X_n$, if either:

1. $X_1X_2 \ldots X_n \stackrel{*}{\Rightarrow} a\alpha$, or
2. $X_1X_2 \ldots X_n \stackrel{*}{\Rightarrow} \epsilon$, and there is a derivation $S_\# \stackrel{*}{\Rightarrow} \alpha Y a \beta$, ie, a can follow Y in a derivation

For this we need two helper functions, first and follow

first($X_1X_2 \ldots X_n$) $= \{a | X_1X_2 \ldots X_n \stackrel{*}{\Rightarrow} a\alpha, a \in T\}$, ie, the set of all terminals that can start strings derivable from $X_1X_2 \ldots X_n$

LL(1) Parsing

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 $Y ::= X_1X_2 \ldots X_n$, if either:

1. $X_1X_2 \ldots X_n \overset{*}{\Rightarrow} a\alpha$, or
2. $X_1X_2 \ldots X_n \overset{*}{\Rightarrow} \epsilon$, and there is a derivation $S_\# \overset{*}{\Rightarrow} \alpha Y a \beta$, ie, a can follow Y in a derivation

For this we need two helper functions, first and follow

$\text{first}(X_1X_2 \ldots X_n) = \{a | X_1X_2 \ldots X_n \overset{*}{\Rightarrow} a\alpha, a \in T\}$, ie, the set of all terminals that can start strings derivable from $X_1X_2 \ldots X_n$

If $X_1X_2 \ldots X_n \overset{*}{\Rightarrow} \epsilon$, then we say that $\text{first}(X_1X_2 \ldots X_n)$ includes ϵ

LL(1) Parsing

LL(1) Parsing

Algorithm 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 $X \in T \cup N$

for $X \in T$ **do**
    first(X) $\leftarrow \{X\}$
end for
for $X \in N$ **do**
    first(X) $\leftarrow \{\}$
end for
if $X ::= \epsilon \in P$ **then**
    Add ϵ to first(X)
end if
repeat
    for $Y ::= X_1X_2 \ldots X_n \in P$ **do**
        Add first($X_1X_2 \ldots X_n$) to first(Y)
    end for
until no new symbols are added to any set

LL(1) Parsing

LL(1) Parsing

Algorithm first($X_1 X_2 \dots X_n$) for a sequence of symbols $X_1 X_2 \dots X_n$ in a grammar G

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

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

LL(1) Parsing

LL(1) Parsing

Example

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

LL(1) Parsing

Example

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

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

LL(1) Parsing

LL(1) Parsing

To determine when the rule $X ::= \epsilon$ is applicable, we need the notion of follow

LL(1) Parsing

To determine when the rule $X ::= \epsilon$ is applicable, we need the notion of follow

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

LL(1) Parsing

To determine when the rule $X ::= \epsilon$ is applicable, we need the notion of follow

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

Alternate definition:

LL(1) Parsing

To determine when the rule $X ::= \epsilon$ is applicable, we need the notion of follow

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

Alternate definition:

1. follow(S) contains #, ie, the terminator follows the start symbol

To determine when the rule $X ::= \epsilon$ is applicable, we need the notion of follow

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

Alternate definition:

1. $\text{follow}(S)$ contains #, ie, the terminator follows the start symbol
2. If there is a rule $Y ::= \alpha X \beta$ in P, $\text{follow}(X)$ contains $\text{first}(\beta) - \{\epsilon\}$

LL(1) Parsing

To determine when the rule $X ::= \epsilon$ is applicable, we need the notion of follow

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

Alternate definition:

1. follow(S) contains #, ie, the terminator follows the start symbol
2. If there is a rule $Y ::= \alpha X \beta$ in P, follow(X) contains first(β) $- \{\epsilon\}$
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)

LL(1) Parsing

Algorithm 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 symbols $X \in N$

follow(S) $\leftarrow \{\#\}$
for $X \in N$ **do**
    follow(X) $\leftarrow \{\}$
end for
repeat
    for $Y ::= X_1 X_2 \ldots X_n \in P$ **do**
        for $X_i \in X_1 X_2 \ldots X_n$ **do**
            Add first($X_{i+1} X_{i+2} \ldots X_n$) $- \{\epsilon\}$ to follow(X_i)
            If X_i is the last symbol or $\epsilon \in$ first($X_{i+1} \ldots X_n$), add follow(Y) to follow(X_i)
        end for
    end for
until no new symbols are added to any set

LL(1) Parsing

Example

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

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

LL(1) Parsing

Example

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

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

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

LL(1) Parsing

Algorithm LL(1) parse table for a grammar G

Input: a context-free grammar $G = (N, T, S, P)$

Output: LL(1) parse table for G

```
for Y ∈ N do
    for Y ::= X_1 X_2 ... X_n ∈ P with index i do
        for a ∈ first(X_1 X_2 ... X_n) − {ε} do
            table[Y, a] ← i
            if ε ∈ first(X_1 X_2 ... X_n) then
                for a ∈ follow(Y) do
                    table[Y, a] ← i
                end for
            end if
        end for
    end for
end for
```

LL(1) Parsing

LL(1) Parsing

Example

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

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

LL(1) Parsing

Example

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

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

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

LL(1) Parsing

LL(1) Parsing

We say a grammar is LL(1) if the parse table has no conflicts, ie, no entries with more than one rule

LL(1) Parsing

We say a grammar is LL(1) if the parse table has no conflicts, ie, no entries with more than one rule

If a grammar is LL(1), then it is unambiguous

LL(1) Parsing

We say a grammar is LL(1) if the parse table has no conflicts, ie, no entries with more than one rule

If a grammar is 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

LL(1) Parsing

We say a grammar is LL(1) if the parse table has no conflicts, ie, no entries with more than one rule

If a grammar is 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

Not all context-free grammars are LL(1), but for many that are not, one may define equivalent grammars that are LL(1)

LL(1) Parsing

LL(1) Parsing

One type of grammar that is not LL(1) is a grammar having a rule with direct left recursion

$$\mathrm{Y} ::= \mathrm{Y}\ \alpha$$
$$\mathrm{Y} ::= \beta$$

LL(1) Parsing

One type of grammar that is not LL(1) is a grammar having a rule with direct left recursion

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

Removing direct left recursion

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

LL(1) Parsing

LL(1) Parsing

Example (a non LL(1) grammar with direct left recursion)

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

LL(1) Parsing

Example (a non LL(1) grammar with direct left recursion)

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

Equivalent LL(1) grammar

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

LL(1) Parsing

LL(1) Parsing

Algorithm Remove left recursion for a grammar G

Input: a context-free grammar $G = (N, T, S, P)$
Output: G with left recursion eliminated

```
Arbitrarily enumerate the non-terminals of G
for i := 1 to n do
    for j := 1 to i − 1 do
        Replace pairs of rules of the form X_i ::= X_j α and X_j ::= β_1|β_2| ... |β_k by the rules X_i ::= β_1α|β_2α| ... |β_kα
        Eliminate any direct left recursion
    end for
end for
```

Bottom-up Deterministic Parsing

Bottom-up Deterministic Parsing

The bottom-up parser proceeds via a sequence of shifts and reductions, until the start symbol is on top of the stack and the input is just the terminator symbol #

Bottom-up Deterministic Parsing

The bottom-up parser proceeds via a sequence of shifts and reductions, until the start symbol is on top of the stack and the input is just the terminator symbol `#`

Example (parsing `id+id*id`)

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

Stack	Input	Action
	`id+id*id#`	shift
`id`	`+id*id#`	reduce 6
F	`+id*id#`	reduce 4
T	`+id*id#`	reduce 2
E	`+id*id#`	shift
E`+`	`id*id#`	shift
E`+id`	`*id#`	reduce 6
E`+`F	`*id#`	reduce 4
E`+`T	`*id#`	shift
E`+`T`*`	`id#`	shift
E`+`T`*id`	`#`	reduce 6
E`+`T`*`F	`#`	reduce 3
E`+`T	`#`	reduce 1
E	`#`	✓

Bottom-up Deterministic Parsing

Bottom-up Deterministic Parsing

The following questions arise:

Bottom-up Deterministic Parsing

The following questions arise:

- How does the parser know when to shift and when to reduce?

Bottom-up Deterministic Parsing

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?

Bottom-up Deterministic Parsing

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 the reduction?

Bottom-up Deterministic Parsing

Bottom-up Deterministic Parsing

The stack configuration combined with the un-scanned input stream represents a sentential form in a right-most derivation of the input

Bottom-up Deterministic Parsing

The stack configuration combined with the un-scanned input stream represents a sentential form in a right-most derivation of the input

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

The stack configuration combined with the un-scanned input stream represents a sentential form in a right-most derivation of the input

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

Formally, in a right-most derivation, $S \stackrel{*}{\Rightarrow} \alpha Y w \Rightarrow \alpha\beta w \stackrel{*}{\Rightarrow} uw$, a handle is a rule $Y ::= \beta$ and a position in $\alpha\beta w$ where β may be replaced by Y

Bottom-up Deterministic Parsing

The stack configuration combined with the un-scanned input stream represents a sentential form in a right-most derivation of the input

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

Formally, in a right-most derivation, $S \stackrel{*}{\Rightarrow} \alpha Y w \Rightarrow \alpha\beta w \stackrel{*}{\Rightarrow} uw$, a handle is a rule $Y ::= \beta$ and a position in $\alpha\beta w$ where β may be replaced by Y

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

The stack configuration combined with the un-scanned input stream represents a sentential form in a right-most derivation of the input

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

Formally, in a right-most derivation, $S \stackrel{*}{\Rightarrow} \alpha Y w \Rightarrow \alpha\beta w \stackrel{*}{\Rightarrow} uw$, a handle is a rule $Y ::= \beta$ and a position in $\alpha\beta w$ where β may be replaced by Y

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)

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

Bottom-up Deterministic Parsing

The stack configuration combined with the un-scanned input stream represents a sentential form in a right-most derivation of the input

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

Formally, in a right-most derivation, $S \stackrel{*}{\Rightarrow} \alpha Y w \Rightarrow \alpha\beta w \stackrel{*}{\Rightarrow} uw$, a handle is a rule $Y ::= \beta$ and a position in $\alpha\beta w$ where β may be replaced by Y

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)

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

If there is not a handle on top of the stack and shifting an input token onto the stack results in a viable prefix, a shift is called for

LR(1) Parsing

LR(1) Parsing

The LR(1) parsing algorithm is a state machine with a pushdown stack, and is driven by two tables: Action and Goto

LR(1) Parsing

The LR(1) parsing algorithm is a state machine with a pushdown stack, and is 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

LR(1) Parsing

The LR(1) parsing algorithm is a state machine with a pushdown stack, and is 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

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

LR(1) Parsing

LR(1) Parsing

Algorithm LR(1) parsing algorithm

Input: Action and Goto tables and the input sentence w 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

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

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 index of the rule $Y ::= X_j X_{j+1} \ldots X_m$, the parser replaces the symbols and states $X_j s_j X_{j+1} s_{j+1} \ldots X_m s_m$ by Ys, where $s =$ Goto$[s_{j-1}, Y]$, and outputs i

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

Otherwise, if Action$[s_m, a_k] =$ accept, the parser halts successfully

Otherwise, if Action$[s_m, a_k] =$ error, the parser raises an error

until either the sentence is parsed or an error is raised

LR(1) Parsing

LR(1) Parsing

Example (parsing `id+id*id`)

	Action						Goto		
	+	*	(	)	id	#	E	T	F
0			s4		s5		1	2	3
1	s6					✓			
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					

1. $E ::= E + T$
2. $E ::= T$
3. $T ::= T * \mathrm{F}$
4. $T ::= F$
5. $F ::= (E)$
6. $F ::= \texttt{id}$

LR(1) Parsing

Example (parsing `id+id*id`)

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

	Action						Goto		
	+	*	(	)	id	#	E	T	F
0			s4		s5		1	2	3
1	s6					✓			
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					

Stack	Input	Action
0	id+id*id#	s5
0id5	+id*id#	r6
0F3	+id*id#	r4
0T2	+id*id#	r2
0E1	+id*id#	s6
0E1+6	id*id#	s5
0E1+6id5	*id#	r6
0E1+6F3	*id#	r4
0E1+6T13	*id#	s7
0E1+6T13*7	id#	s5
0E1+6T13*7id5	#	r6
0E1+6T13*7F14	#	r3
0E1+6T13	#	r1
0E1	#	✓

LR(1) Parsing

LR(1) 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

LR(1) 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

The DFA is constructed from the 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 rule in P, α and β are (possibly empty) strings of symbols, and $\mathtt{a}$ is a lookahead symbol

LR(1) 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

The DFA is constructed from the 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 rule in P, α and β are (possibly empty) strings of symbols, and $\mathtt{a}$ is a lookahead symbol

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

LR(1) 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

The DFA is constructed from the 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 rule in P, α and β are (possibly empty) strings of symbols, and $\mathtt{a}$ is a lookahead symbol

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 $\mathtt{a}$ is a token that can follow Y (and so, $\alpha\beta$) in a legal right-most derivation of some sentence

LR(1) Parsing

LR(1) Parsing

The following item is called a possibility

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

LR(1) Parsing

The following item is called a possibility

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

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

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

LR(1) Parsing

The following item is called a possibility

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

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

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

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

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

LR(1) Parsing

The states in the DFA for recognizing viable prefixes and handles are constructed from items

LR(1) Parsing

The states in the DFA for recognizing viable prefixes and handles are constructed from items

We first augment our grammar G with an additional start symbol S' and an additional rule so as to yield an equivalent grammar G'

$$S' ::= S$$

The states in the DFA for recognizing viable prefixes and handles are constructed from items

We first augment our grammar G with an additional start symbol S' and an additional rule so as to yield an equivalent grammar G'

$$S' ::= S$$

Example (augmented arithmetic expression grammar)

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

LR(1) Parsing

LR(1) Parsing

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

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

which says that parsing an S' means parsing an S from the input, after which point the next (and last) remaining token is the terminator #

LR(1) Parsing

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

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

which says that parsing an S' means parsing an S from the input, after which point the next (and last) remaining token is the terminator #

The kernel may imply additional items, which are computed as the closure of the set

LR(1) Parsing

LR(1) Parsing

Algorithm Computing the closure of a set of items

Input: a set of items s

Output: closure(s)

$C \leftarrow \mathrm{Set}(s)$

repeat

If C contains an item of the form

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

then add the item

$[X ::= \cdot\ \gamma,\ \mathtt{b}]$

to C for every rule $X ::= \gamma$ in P and for every token b in first($\beta\mathtt{a}$)

until no new items may be added

return C

LR(1) Parsing

LR(1) Parsing

Example

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

Example

$$
\begin{array}{lll}
0. & E' & ::= E \\
1. & E & ::= E \mathtt{+} T \\
2. & E & ::= T \\
3. & T & ::= T \mathtt{*} F \\
4. & T & ::= F \\
5. & F & ::= \mathtt{(}E\mathtt{)} \\
6. & F & ::= \mathtt{id}
\end{array}
$$

closure($\{[E' ::= \cdot E, \mathtt{\#}]\}$) yields

$$
\begin{array}{l}
\{[E' ::= \cdot\ E, \mathtt{\#}], \\
\ [E\ ::= \cdot\ E \mathtt{+} T, \mathtt{+/\#}], \\
\ [E\ ::= \cdot\ T, \mathtt{+/\#}], \\
\ [T\ ::= \cdot\ T \mathtt{*} F, \mathtt{+/*/\#}], \\
\ [T\ ::= \cdot\ F, \mathtt{+/*/\#}], \\
\ [F\ ::= \cdot\ \mathtt{(}E\mathtt{)}, \mathtt{+/*/\#}], \\
\ [F\ ::= \cdot\ \mathtt{id}, \mathtt{+/*/\#}]\}
\end{array}
$$

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

LR(1) Parsing

LR(1) Parsing

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 $\text{goto}(s, X)$, take all items from s with a $\cdot$ before the X and move it after the X, and hence take the closure of that

LR(1) Parsing

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$(s, X)$, take all items from s with a $\cdot$ before the X and move it after the X, and hence 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)

$r \leftarrow$ Set()
for $[Y ::= \alpha \cdot X\beta,\ \mathtt{a}] \in s$ **do**
    r.add($[Y ::= \alpha X \cdot \beta,\ \mathtt{a}]$)
end for
return closure(r)

LR(1) Parsing

Example

$$
\begin{array}{lll}
0. & E' & ::= E \\
1. & E & ::= E \;\texttt{+}\; T \\
2. & E & ::= T \\
3. & T & ::= T \;\texttt{*}\; F \\
4. & T & ::= F \\
5. & F & ::= \texttt{(}E\texttt{)} \\
6. & F & ::= \texttt{id}
\end{array}
$$

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

LR(1) Parsing

Example

$$
\begin{aligned}
&0.\ E' ::= E\\
&1.\ E ::= E + T\\
&2.\ E ::= T\\
&3.\ T ::= T * F\\
&4.\ T ::= F\\
&5.\ F ::= (E)\\
&6.\ F ::= \texttt{id}
\end{aligned}
$$

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

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

Example

$$
\begin{array}{ll}
0. & E' ::= E \\
1. & E ::= E + T \\
2. & E ::= T \\
3. & T ::= T * F \\
4. & T ::= F \\
5. & F ::= (E) \\
6. & F ::= \mathtt{id}
\end{array}
$$

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

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

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

Example

$$
\begin{array}{lll}
0. & E' & ::= E \\
1. & E & ::= E + T \\
2. & E & ::= T \\
3. & T & ::= T * F \\
4. & T & ::= F \\
5. & F & ::= (E) \\
6. & F & ::= \mathtt{id}
\end{array}
$$

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

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

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

$$
\mathrm{goto}(s_0, F) = s_3 = \{ [T ::= F\, \cdot,\ +/*/\#] \}
$$

Example

$$
\begin{aligned}
0.\ & E' ::= E \\
1.\ & E ::= E + T \\
2.\ & E ::= T \\
3.\ & T ::= T * F \\
4.\ & T ::= F \\
5.\ & F ::= (E) \\
6.\ & F ::= \mathtt{id}
\end{aligned}
$$

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

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

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

$$
\mathrm{goto}(s_0, F) = s_3 = \{[T ::= F\, \cdot,\ +/*/\#]\}
$$

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

LR(1) Parsing

Example

$$
\begin{array}{lll}
0. & E' & ::= E \\
1. & E & ::= E \texttt{+} T \\
2. & E & ::= T \\
3. & T & ::= T \texttt{*} F \\
4. & T & ::= F \\
5. & F & ::= \texttt{(}E\texttt{)} \\
6. & F & ::= \texttt{id}
\end{array}
$$

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

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

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

$$
\mathrm{goto}(s_0, F) = s_3 = \{[T ::= F\ \cdot,\ \texttt{+/*/\#}]\}
$$

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

$$
\mathrm{goto}(s_0, \texttt{id}) = s_5 = \{[F ::= \texttt{id}\ \cdot,\ \texttt{+/*/\#}]\}
$$

LR(1) Parsing

LR(1) Parsing

Algorithm Computing the LR(1) collection

Input: a context-free grammar $G = (N, T, S, P)$

Output: the canonical LR(1) collection of states $\mathcal{C} = \{s_0, s_1, \ldots, s_n\}$

Define an augmented grammar G' which is G with the added non-terminal S' and added production rule $S' ::= S$

$s_0 \leftarrow \text{closure}(\{[S' ::= \cdot S, \#]\})$

$\mathcal{C} \leftarrow \text{Set}(s_0)$

repeat

  for $s \in \mathcal{C}$ **do**

    for $X \in T \cup N$ **do**

      if $\text{goto}(s, X) \neq \emptyset$ and $\text{goto}(s, X) \notin \mathcal{C}$ **then**

        $\mathcal{C}.\text{add}(\text{goto}(s, X))$

      end if

    end for

  end for

until no new states are added to $\mathcal{C}$

LR(1) Parsing

Example (the LR(1) canonical collection for the arithmetic expression grammar)

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

$\text{goto}(s_0, E) = \{[E' ::= E\cdot, \#], [E ::= E\cdot{+}T, +/\#]\} = s_1$
$\text{goto}(s_0, T) = \{[E ::= T\cdot, +/\#], [T ::= T\cdot{*}F, +/{*}/\#], \} = s_2$
$\text{goto}(s_0, F) = \{[T ::= F\cdot, +/{*}/\#]\} = s_3$
$\text{goto}(s_0, () = \{[F ::= (\cdot E), +/{*}/\#], [E ::= \cdot E{+}T, +/)], [E ::= \cdot T, +/)], [T ::= \cdot T{*}F, +/{*}/)], [T ::= \cdot F, +/{*}/)], [F ::= \cdot(E), +/{*}/)], [F ::= \cdot\mathtt{id}, +/{*}/)]\} = s_4$
$\text{goto}(s_0, \mathtt{id}) = \{[F ::= \mathtt{id}\cdot, +/{*}/\#]\} = s_5$

$\text{goto}(s_1, +) = \{[E ::= E{+}\cdot T, +/\#], [T ::= \cdot T{*}F, +/{*}/\#], [T ::= \cdot F, +/{*}/\#], [F ::= \cdot(E), +/{*}/\#], [F ::= \cdot\mathtt{id}, +/{*}/\#]\} = s_6$

$\text{goto}(s_2, *) = \{[T ::= T{*}\cdot F, +/{*}/\#], [F ::= \cdot(E), +/{*}/\#], [F ::= \cdot\mathtt{id}, +/{*}/\#]\} = s_7$

$\text{goto}(s_4, E) = \{[F ::= (E\cdot), +/{*}/\#], [E ::= E\cdot{+}T, +/)]\} = s_8$
$\text{goto}(s_4, T) = \{[E ::= T\cdot, +/)], [T ::= T\cdot{*}F, +/{*}/)]\} = s_9$
$\text{goto}(s_4, F) = \{[T ::= F\cdot, +/{*}/)]\} = s_{10}$
$\text{goto}(s_4, () = \{[F ::= (\cdot E), +/{*}/)], [E ::= \cdot E{+}T, +/)], [E ::= \cdot T, +/)], [T ::= \cdot T{*}F, +/{*}/)], [T ::= \cdot F, +/{*}/)], [F ::= \cdot(E), +/{*}/)], [F ::= \cdot\mathtt{id}, +/{*}/)]\} = s_{11}$
$\text{goto}(s_4, \mathtt{id}) = \{[F ::= \mathtt{id}\cdot, +/{*}/)]\} = s_{12}$

$\text{goto}(s_6, T) = \{[E ::= E{+}T\cdot, +/\#], [T ::= T\cdot{*}F, +/{*}/\#]\} = s_{13}$
$\text{goto}(s_6, F) = s_3$
$\text{goto}(s_6, () = s_4$
$\text{goto}(s_6, \mathtt{id}) = s_5$

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

LR(1) Parsing

$\text{goto}(s_8\,,\,)) = \{[F ::= (E)\cdot\,,\, +/*/\#]\} = s_{15}$
$\text{goto}(s_8\,,+) = \{[E ::= E{+}\cdot\; T, +/)], [T ::= \cdot T{*}F, +/*/)], [T ::= \cdot F, +/*/)], [F ::= \cdot(E), +/*/)], [F ::= \cdot\texttt{id}, +/*/)]\} = s_{16}$

$\text{goto}(s_9\,,*) = \{[T ::= T{*}\cdot\; F, +/*/)], [F ::= \cdot(E), +/*/)], [F ::= \cdot\texttt{id}, +/*/)]\} = s_{17}$

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

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

$\text{goto}(s_{16}\,,T) = \{[E ::= E{+}T\cdot, +/)][T ::= T\cdot *F, +/*/)]\} = s_{19}$
$\text{goto}(s_{16}\,,F) = s_{10}$
$\text{goto}(s_{16}\,,() = s_{11}$
$\text{goto}(s_{16}\,,\texttt{id}) = s_{12}$

$\text{goto}(s_{17}\,,F) = \{[T ::= T{*}F\cdot, +/*/)]\} = s_{20}$
$\text{goto}(s_{17}\,,() = s_{11}$
$\text{goto}(s_{17}\,,\texttt{id}) = s_{12}$

$\text{goto}(s_{18}\,,() = \{[F ::= (E)\cdot, +/*/)],\}= s_{21}$
$\text{goto}(s_{18}\,,+) = s_{16}$

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

LR(1) Parsing

Algorithm Constructing the LR(1) parse tables for a context-free grammar

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

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

LR(1) Parsing

LR(1) Parsing

Example (Action and Goto tables for the arithmetic expression grammar)

	Action						Goto		
	+	*	(	)	id	#	E	T	F
0			s4		s5		1	2	3
1	s6					✓			
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					

LR(1) Parsing

LR(1) Parsing

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

LR(1) Parsing

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

The shift-reduce conflict can occur when there are items of the forms

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

LR(1) Parsing

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

The shift-reduce conflict can occur when there are items of the forms

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

Example (the dangling else problem)

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

Most parser generators that are based on LR grammars favor a shift of the $\texttt{else}$ over a reduce of the $\texttt{if (} E \texttt{)} \ S$ to an S

LR(1) Parsing

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

The shift-reduce conflict can occur when there are items of the forms

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

Example (the dangling else problem)

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

Most parser generators that are based on LR grammars favor a shift of the $\texttt{else}$ over a reduce of the $\texttt{if (} E \texttt{)}\ S$ to an S

The reduce-reduce conflict can happen when we have a state containing two items of the form

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

JavaCC

JavaCC

Besides containing the regular expressions for the lexical structure for *j--*, the `j--.jj` file also contains the syntactic rules for the language

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

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 (eg, `reportParserError()`), which are available for use within the generated parser; some of the helpers include

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 (eg, `reportParserError()`), which are available for use within the generated parser; some of the helpers include

Following the block is the specification for the scanner for *j--*, and following that is the specification for the parser for *j--*

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 (eg, `reportParserError()`), which are available for use within the generated parser; some of the helpers include

Following the block is the specification for the scanner for *j--*, and following that is the specification for the parser for *j--*

We define a start symbol, which is a high level non-terminal (`compilationUnit` in case of *j--*) that references other lower level non-terminals, which in turn reference the tokens

JavaCC

BNF syntax is allowed in the syntactic specification:

BNF syntax is allowed in the syntactic specification:

- [a] for an "optional" occurrence of a

BNF syntax is allowed in the syntactic specification:

- [*a*] for an "optional" occurrence of *a*
- (a)$*$ for "zero or more" occurrences of *a*

BNF syntax is allowed in the syntactic specification:

- $[a]$ for an "optional" occurrence of a
- $(a)*$ for "zero or more" occurrences of a
- $a|b$ for either a or b

BNF syntax is allowed in the syntactic specification:

- $[a]$ for an "optional" occurrence of a
- $(a)*$ for "zero or more" occurrences of a
- $a|b$ for either a or b
- () for grouping

JavaCC

Syntax for a non-terminal declaration

j--.jj

```
private|public <type> <name>(<parameter1>, <parameter2>, ...):
{
    // Local variables.
    ...
}
{
    try {
      // BNF rules along with any syntactic actions that must be taken as the rules are parsed.
      ...
    } catch (ParseException e) {
        recoverFromError(new int[] { SEMI, EOF }, e);
    }
    {
        return <expression>;
    }
}
```

Syntax for a non-terminal declaration

j--.jj

```
private|public <type> <name>(<parameter1>, <parameter2>, ...):
{
    // Local variables.
    ...
}
{
    try {
      // BNF rules along with any syntactic actions that must be taken as the rules are parsed.
      ...
    } catch (ParseException e) {
        recoverFromError(new int[] { SEMI, EOF }, e);
    }
    {
        return <expression>;
    }
}
```

Syntactic actions, such as creating/returning an AST node, are Java statements embedded within curly braces

Syntax for a non-terminal declaration

j--.jj

```
private|public <type> <name>(<parameter1>, <parameter2>, ...):
{
    // Local variables.
    ...
}
{
    try {
      // BNF rules along with any syntactic actions that must be taken as the rules are parsed.
      ...
    } catch (ParseException e) {
        recoverFromError(new int[] { SEMI, EOF }, e);
    }
    {
        return <expression>;
    }
}
```

Syntactic actions, such as creating/returning an AST node, are Java statements embedded within curly braces

JavaCC turns the specification for each non-terminal into a Java method within the generated parser

JavaCC

Example (parsing a compilation unit)

```
compilationUnit ::= [ PACKAGE qualifiedIdentifier SEMI ]
                    { IMPORT  qualifiedIdentifier SEMI }
                    { typeDeclaration }
                    EOF
```

JavaCC

JavaCC

j--.jj

```
public JCompilationUnit compilationUnit():
{
    int line = 0;
    TypeName packageName = null, anImport = null;
    ArrayList<TypeName> imports = new ArrayList<TypeName>();
    JAST aTypeDeclaration = null;
    ArrayList<JAST> typeDeclarations = new ArrayList<JAST>();
}
{
    try {
        [
            <PACKAGE> { line = token.beginLine; }
            packageName = qualifiedIdentifier()
            <SEMI>
        ]
        (
            <IMPORT> { line = line == 0 ? token.beginLine : line; }
            anImport = qualifiedIdentifier()
            { imports.add(anImport); }
            <SEMI>
        )*
        (
            aTypeDeclaration = typeDeclaration()
            {
                line = line == 0 ? aTypeDeclaration.line() : line;
                typeDeclarations.add(aTypeDeclaration);
            }
        )*
        <EOF> { line = line == 0 ? token.beginLine : line; }
    } catch (ParseException e) {
        recoverFromError(new int[] { SEMI, EOF }, e);
    }
    { return new JCompilationUnit(fileName, line, packageName, imports, typeDeclarations); }
}
```

JavaCC

Example (parsing a qualified identifier)

```
qualifiedIdentifier ::= IDENTIFIER { DOT IDENTIFIER }
```

Example (parsing a qualified identifier)

```
qualifiedIdentifier ::= IDENTIFIER { DOT IDENTIFIER }
```

j--.jj

```
private TypeName qualifiedIdentifier():
{
    int line = 0;
    String qualifiedIdentifier = "";
}
{
    try {
        <IDENTIFIER>
        {
            line = token.beginLine;
            qualifiedIdentifier = token.image;
        }
        (
            LOOKAHEAD(<DOT> <IDENTIFIER>)
            <DOT> <IDENTIFIER>
            { qualifiedIdentifier += "." + token.image; }
        )*
    } catch (ParseException e) {
        recoverFromError(new int[] { SEMI, EOF }, e);
    }
    { return new TypeName(line, qualifiedIdentifier); }
}
```

JavaCC

Example (parsing a statement)

```
statement ::= block
            | IF parExpression statement [ ELSE statement ]
            | WHILE parExpression statement
            | RETURN [ expression ] SEMI
            | SEMI
            | statementExpression SEMI
```

JavaCC

j--.jj

```
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); } |
```

j--.jj

```
        <SEMI>
        {
           line = token.beginLine;
            statement = new JEmptyStatement( line );
        } |
        statement = statementExpression()
        <SEMI>
    } catch (ParseException e) {
        recoverFromError(new int[] { SEMI, EOF }, e);
    }
    { return statement; }
}
```

JavaCC

Example (parsing a simple unary expression)

```
simpleUnaryExpression ::= LNOT unaryExpression
                        | LPAREN basicType RPAREN unaryExpression
                        | LPAREN referenceType RPAREN simpleUnaryExpression
                        | postfixExpression
```

JavaCC

j--.jj

```
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); } |
        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 ; }
}
```

JavaCC

JavaCC

The error recovery mechanism in the 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

JavaCC

The error recovery mechanism in the 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` and the `skipTo` array is passed to the `recoverFromError()` error recovery function

The error recovery mechanism in the 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` and the `skipTo` array is passed to the `recoverFromError()` error recovery function

The exception instance has information about the token that was found and the token that was expected, and the `skipTo` array has tokens to skip to in order to recover from the error

The error recovery mechanism in the 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` and the `skipTo` array is passed to the `recoverFromError()` error recovery function

The exception instance has information about the token that was found and the token that was expected, and the `skipTo` array has tokens to skip to in order to recover from the error

In the current error recovery scheme, `skipTo` always consists of the two tokens `SEMI` and `EOF`

The error recovery mechanism in the 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` and the `skipTo` array is passed to the `recoverFromError()` error recovery function

The exception instance has information about the token that was found and the token that was expected, and the `skipTo` array has tokens to skip to in order to recover from the error

In the current error recovery scheme, `skipTo` always consists of the two tokens `SEMI` and `EOF`

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

JavaCC

j--.jj

```java
    private void recoverFromError(int[] skipTo, ParseException e) {
        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("...");
            }
        }
        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);
        }
        boolean loop = true;
        do {
            token = getNextToken();
            for (int i = 0; i < skipTo.length; i++) {
                if (token.kind == skipTo[i]) {
                    loop = false;
                    break;
                }
            }
        } while(loop);
    }
```