Introduction to Compiler Construction

Parsing: Top-down Recursive Descent Algorithm

Outline

Top-down Parsing

Top-down Parsing

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

Top-down Parsing

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

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

Top-down Parsing

Top-down Parsing

Example (compilation unit in *j--*):

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

The goal of parsing a `compilationUnit` is 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 those
3. If there are type declarations, then parse those
4. Finally, parse the terminating `EOF` token

Top-down Parsing

Top-down Parsing

Parsing a token (eg, PACKAGE) is simple; if we see it, we simply scan it

Top-down Parsing

Parsing a token (eg, PACKAGE) is simple; if we see it, we simply scan it

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

Top-down Parsing

Parsing a token (eg, PACKAGE) is simple; if we see it, we simply scan it

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

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 Parsing

Top-down Parsing

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

Top-down Parsing

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

Example (statements in *j--*):

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

- If the next token is an LCURLY, then we parse a block
- If the next token is an IF, then we parse an if statement
- If the next token is a WHILE, then we parse a while statement
- If the next token is a RETURN, then we parse a return statement
- If the next token is a SEMI, then we parse an empty statement
- Otherwise, we parse a statement expression

Top-down Parsing

Top-down Parsing

Since we begin at the start symbol and continually rewrite the non-terminals using rules until we reach the tokens, we call this a top-down parsing technique

Top-down Parsing

Since we begin at the start symbol and continually rewrite the non-terminals using rules until we reach the tokens, we call 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 Parsing

Since we begin at the start symbol and continually rewrite the non-terminals using rules until we reach the tokens, we call 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, we must lookahead several tokens in the input to decide which rule to apply

Top-down Parsing

Since we begin at the start symbol and continually rewrite the non-terminals using rules until we reach the tokens, we call 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, we 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 call this a predictive parsing technique

Recursive Descent Algorithm

Recursive Descent Algorithm

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

Recursive Descent Algorithm

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

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

Recursive Descent Algorithm

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

Based on the next input token, the method 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--*

Helper Methods

Helper Methods

`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`

Helper Methods

`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

Helper Methods

`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

Parser.java

```java
    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;
        }
    }
```

Helper Methods

Helper Methods

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

Helper Methods

`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

Helper Methods

`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

Parser.java

```java
    private boolean isInError = false;

    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;
            }
        }
    }
```

Helper Methods

Helper Methods

Example (a *j*-- program with multiple errors):

</> Parser.java

```
package xyz

import java.lang.Math

public class T1 {
    public static void main(String[] args) {
        System.out.println("Hello, World");
    }
}
```

Helper Methods

Example (a *j*-- program with multiple errors):

</> Parser.java

```
package xyz

import java.lang.Math

public class T1 {
    public static void main(String[] args) {
        System.out.println("Hello, World");
    }
}
```

>_ ~/workspace/j--

```
$ ./bin/j-- -p T1.java
T1.java:3: error: import found where ; sought
T1.java:5: error: public found where ; sought
{
    "JCompilationUnit:1":
    {
        "source": "T1.java",
        "package": "xyz",
        "imports": ["java.lang.Math"],
        "JClassDeclaration:5":
        {
            ...
        }
    }
}
```

Helper Methods

Helper Methods

Example (another *j*-- program with multiple errors):

Parser.java

```java
package xyz

java.lang.Math;

public class T2 {
    public static void main(String[] args) {
        System.out.println("Hello, World");
    }
}
```

Helper Methods

Example (another *j*-- program with multiple errors):

</> Parser.java

```java
package xyz

java.lang.Math;

public class T2 {
    public static void main(String[] args) {
        System.out.println("Hello, World");
    }
}
```

>_ ~/workspace/j--

```
$ ./bin/j-- -p T2.java
T2.java:3: error: java found where ; sought
{
    "JCompilationUnit:1":
    {
        "source": "T2.java",
        "package": "xyz",
        "imports": [],
        "JClassDeclaration:3":
        {
            ...
        }
    }
}
```

Parsing a Compilation Unit

Parsing a Compilation Unit

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

Parsing a Compilation Unit

Parsing a Compilation Unit

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<>();
    while (have(IMPORT)) {
        imports.add(qualifiedIdentifier());
        mustBe(SEMI);
    }
    ArrayList<JAST> typeDeclarations = new ArrayList<>();
    while (!see(EOF)) {
        JAST typeDeclaration = typeDeclaration();
        typeDeclarations.add(typeDeclaration);
    }
    mustBe(EOF);
    return new JCompilationUnit(fileName, line, packageName, imports, typeDeclarations);
}

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);
}
```

Parsing a Qualified Identifier

Parsing a Qualified Identifier

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

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);
    }
```

Parsing a Statement

Parsing a Statement

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

Parsing a Statement

Parsing a Statement

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(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 if (have(WHILE)) {
            JExpression test = parExpression();
            JStatement statement = statement();
            return new JWhileStatement(line, test, statement);
        } else {
            JStatement statement = statementExpression();
            mustBe(SEMI);
            return statement;
        }
    }
```

Lookahead

Lookahead

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

Lookahead

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

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

Lookahead

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

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)

Parsing a Simple Unary Expression

Parsing a Simple Unary Expression

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

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();
        }
    }
```

Parsing a Simple Unary Expression

Parsing a Simple Unary Expression

</> Parser.java

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

Parsing a Simple Unary Expression

Parsing a Simple Unary Expression

Parser.java

```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;
    }
```