Programming Assignments 2: Java Parser

CSE360, Design and Implementation of Compiler

Shao-Hsuan Chu
Instructor: Ye-In Chang
Teaching Assistant: Sheng-Hsin Chiang

National Sun Yat-sen University

June 6, 2021

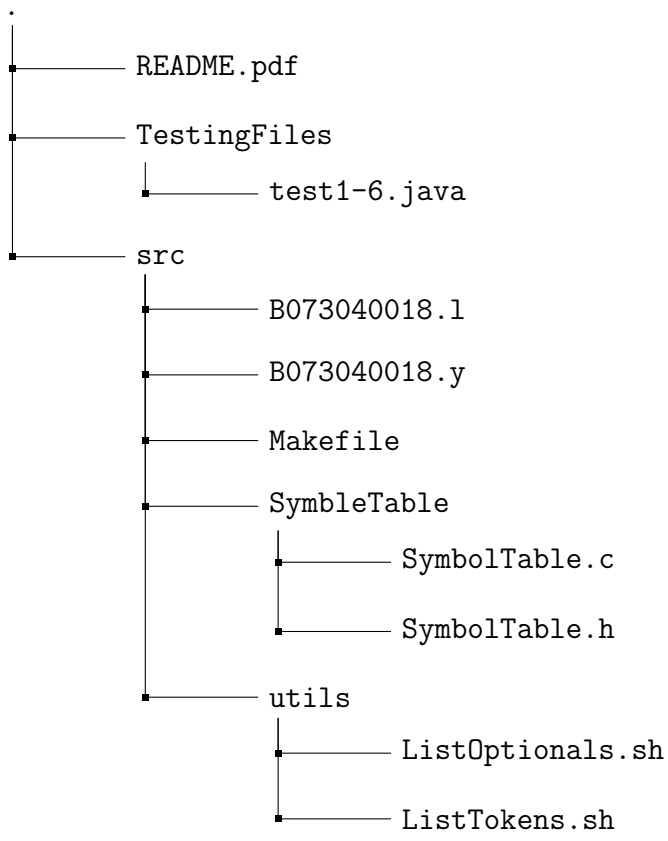
Contents

1 Introduction

In this assignment, we're required to implement a syntactic parser for Java programming language in Lex & Yacc. The parser need to have three main features below.

- A scanner to correctly extract tokens from the raw input and pass them onto the next stage. It also has to identify the redundant characters if they cannot be recognized as any token.
- A parser to examine the syntactic structure based on a pre-defined Java grammar. Upon encountering an error, it needs to be able to recover and keep parsing the rest of the input source file. An expressive error message is also preferred.
- A simple semantic check for redefinitions in the same scope and unused variables.

2 File structure



- `README.pdf`: This file
- `TestFiles`

  - `test1-6.java`: six testing files

- `src`: source code
  - `B073040018.l`: Lex code
  - `B073040018.y`: Lex code
  - `Makefile`: Compile Lex, Yacc and C source code
  - `SymbolTable`: Symbol table header and implementation using hash table
  - `utils`: Utilities
    * `ListOptionals.sh`: Take Lex source file and extract the tokens after the keyword `return` and removes the duplicates.
    * `ListTokens.sh`: Take Yacc source file and extract all of the optionals then generates the rules for them.

3 Environment

3.1 Operating systems

- macOS 11.4
- Ubuntu 18.04.5 LST

3.2 Lex compiler

- flex 2.5.35 Apple(flex-32)
- flex 2.6.4

3.3 Yacc compiler

- bison (GNU Bison) 2.3
- bison (GNU Bison) 3.0.4

4 Usage

4.1 Build

To build the `JavaParser` from source, use `make`.

```
cd src
make [DEBUG=<level>]
```

where you can set the optional flag `DEBUG` to a desired level. If the flag is not provided, then it defaults to level `0`.

4.2 Debug level

The available levels include:

- `Level 0`: Print the errors and warnings only.
- `Level 1`: Print the original source code and errors/warnings in the context.
- `Level 2`: Same as above but this one also prints the symbol table.
- `Level 3`: In addition to above, this also prints the entire parsing process. This sets `yydebug` to `1` in the Yacc source file and generate `y.output`, which contains all of the states and rules.

4.3 Execute

To parse a Java source file, redirect the content into the program. The `TestingFiles` directory contains six testing files. To test a single file

```
./JavaParser < ../TestingFiles/test1.java
```

to test all files altogether

```
cat ../TestingFiles/* | ./JavaParser
```

5 Implementation

5.1 Lexical analysis

Despite the Java's capability of allowing Unicode characters, only ASCII characters can be accepted by the parser in this work for the sake of simplicity. In lexical analysis, the mission is to combine one or more characters in to various tokens. The tokens can be divided into three main groups which are `literals`, `keywords & operators` and `identifier & others`.

5.1.1 Literals

There are six kinds of literals in Java. They are

Boolean literal. For the boolean literal, the accepted words can be either `true` or `false`.

Null literal. For the null literal, it just accepts `null`.

Character literal. For the character literal, the content must be included in two single quotes and it accept any character except `single quote`, `new line character`, and `backslash`. However, the escape sequence is also accepted by the content. See the following Lex source code.

```
EscapeSequence              \\[tbnrf\'\"\\]
CharacterLiteral            \'[^\'\\\n]|{EscapeSequence}\'
```

String literal. For the string literal, the content must be included in two double quotes and it accept any characters except `double quotes`, `new line characters`, and `backslashes`. However, the escape sequence is also accepted by the content. See the following Lex source code.

```
StringLiteral               \"([^\"\\\n]|{EscapeSequence})*\"
```

Integer literal. The integer literal can be further decomposed into `decimal`, `hexadecimal` and `octal` integer literals with a shared optional postfix `l` or `L`. See the following Lex source code.

```
Digits                      [0-9]+
DecimalIntegerLiteral       0|([1-9]{Digits}?)[lL]?
HexIntegerLiteral           0[xX][0-9a-fA-F]+[lL]?
OctalIntegerLiteral         0[0-7]+[lL]?
IntegerLiteral              {DecimalIntegerLiteral}|{HexIntegerLiteral}|{
   ↪ OctalIntegerLiteral}
```

Floating-point literal. The floating-point literal accepts the integer literal plus `decimal point` and `scientific notation` with an optional postfix `f` or `F` for single-precision floating-point and `d` or `D` for the double-precision one.

```
FloatingPointLiteral        ({Digits}\.{Digits}?([eE][/+/-]?{Digits})?[
   ↪ fFdD]?)|(\.{Digits}([eE][\+\-]?{Digits})?[fFdD]?)
```

5.1.2 Keywords & Operators

Keywords The keywords in Java is reserved. Defining them before the identifier prevents the word from being matched with the identifier token. Here's a list of keywords in the original Java. Noted that `const` and `goto` are keywords but never used in Java, hence none of the productions in the next stage use them. Consequently, we don't have to pass them as tokens onto the next stage of parsing.

```
abstract boolean break byte case catch char class const continue default
do double else extends final finally float for goto if implements import
instanceof int interface long native new package private protected public
return short static super switch synchronized this throw throws transient
try void volatile while
```

Operators We also need to capture the operators one by one and pass them as tokens onto the next stage. Here's a list of operators in Java.

```
= *= /= %= += -= <<= >>= >>>= &= ^= |= << >> >>> == != <= >= < > && || !
    ↪ ++ -- & | ^ ~ * / % + - ? : . { } [ ] ( ) , ;
```

5.1.3 Identifier & Others

Identifier The identifier in Java can start with any Unicode characters except digits and some symbols. However, as mentioned above, only ASCII characters are allowed in this work. As the regular expression goes

```
Identifier                  ([a-zA-Z_$])([a-zA-Z0-9_$])*
```

Others Other tokens include `space`, `new line character` and `comment`. There are recognized so they won't be redundant characters, but they won't be passed onto the next stage, either. The space includes single space characters and tabular characters while the comment allows both C-style (/* */) and C++-style (//) comments. See the regular expressions below.

```
Comment                     (\/\/.*\n)|(\/\*([^\*]|\*+[^\*\/])*\*+\/)
Space                       [ \t]
```

5.2 Syntax analysis

6 Screenshots

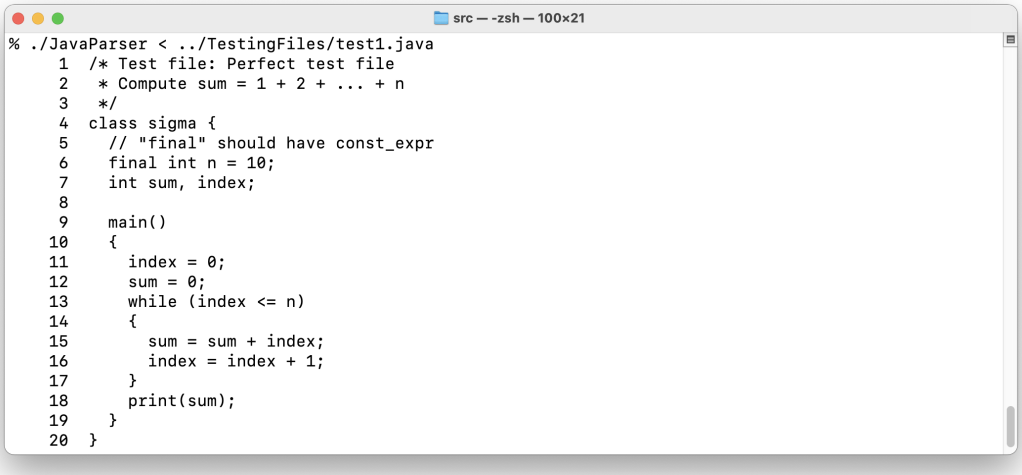


Figure 1: The output of parsing `test1.java` with `DEBUG=1`

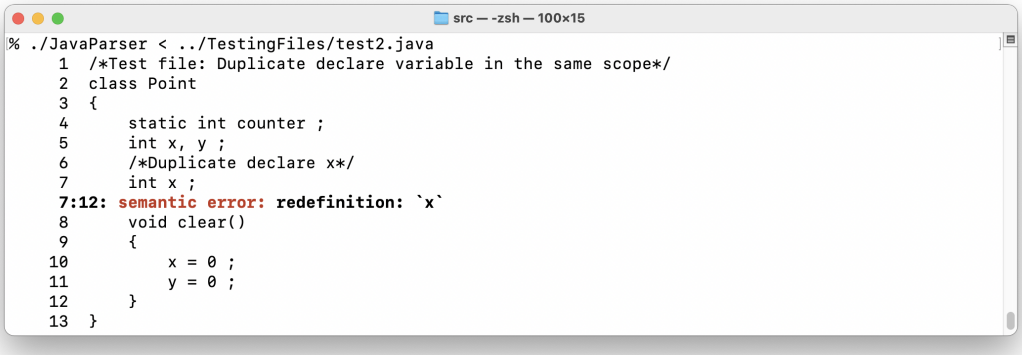


Figure 2: The output of parsing `test2.java` with `DEBUG=1`

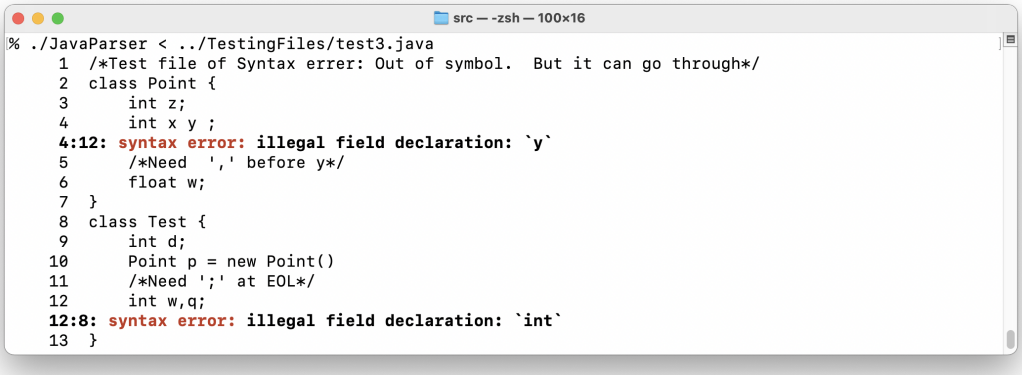


Figure 3: The output of parsing `test3.java` with `DEBUG=1`

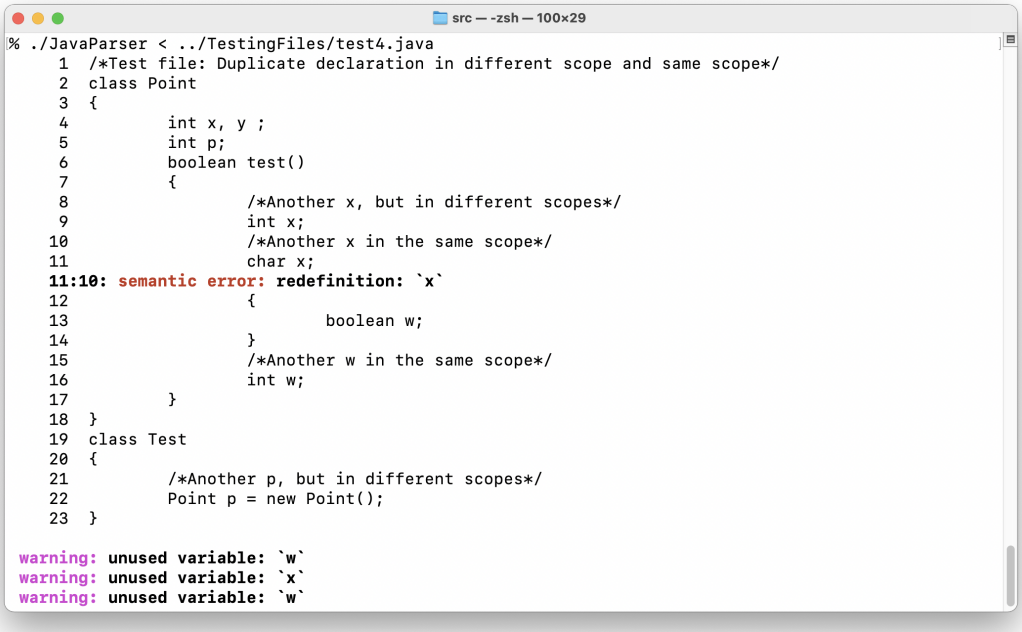


Figure 4: The output of parsing `test4.java` with `DEBUG=1`

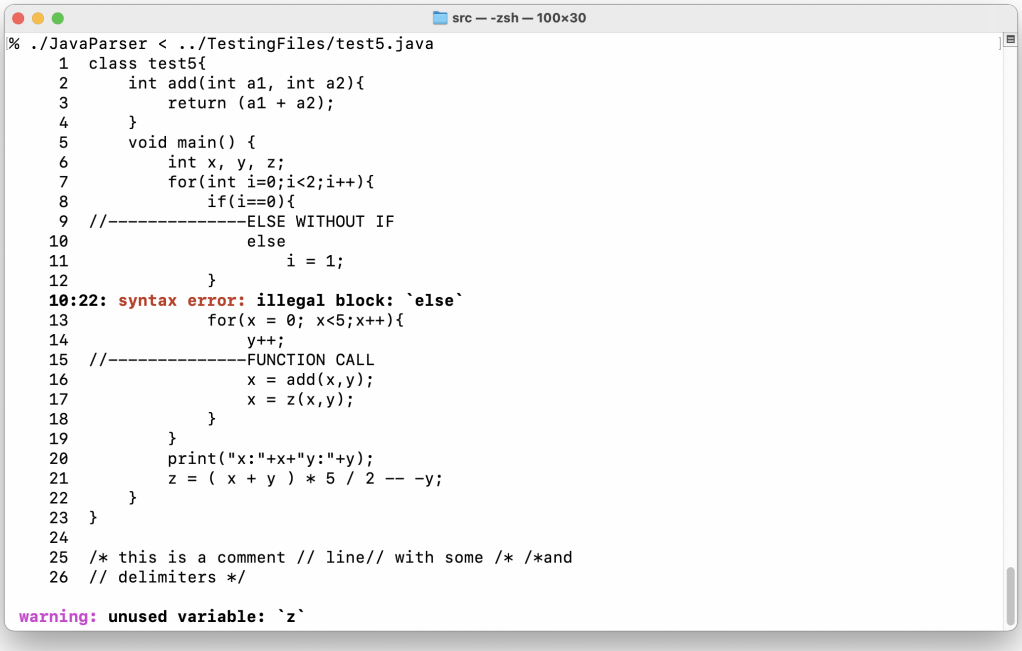


Figure 5: The output of parsing `test5.java` with `DEBUG=1`

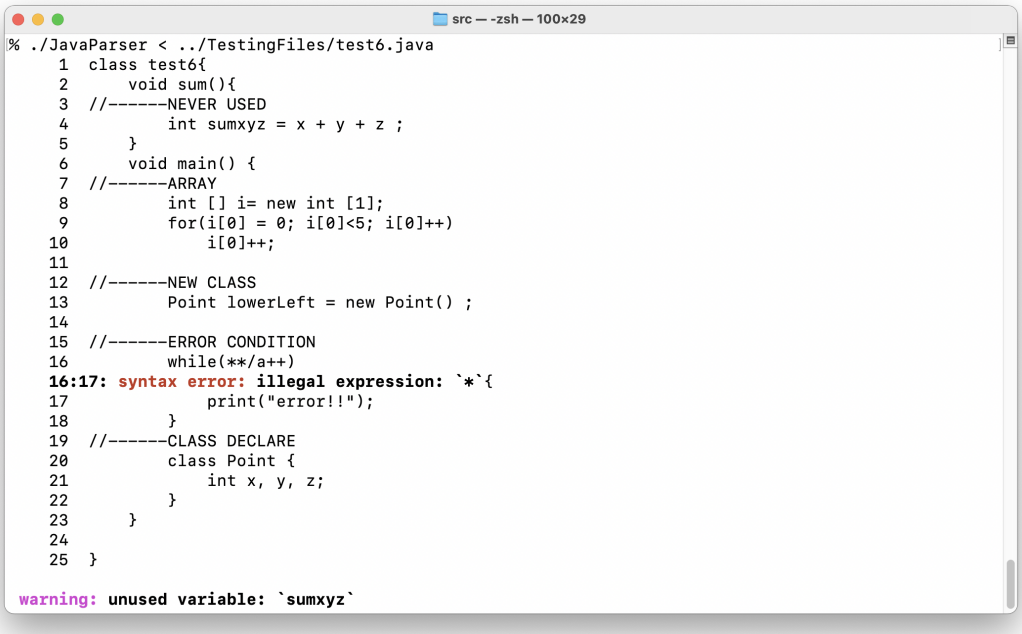


Figure 6: The output of parsing `test6.java` with `DEBUG=1`

References