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 7, 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
├── TestingFiles
│   └── test1-6.java
└── src
    ├── Scanner.l
    ├── Parser.y
    ├── Makefile
    ├── SymbleTable
    │   ├── SymbolTable.c
    │   └── SymbolTable.h
    └── utils
        ├── ListOptionals.sh
        └── ListTokens.sh
```

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

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

- `src`: source code
  - `Scanner.l`: Lex code
  - `Parser.y`: Yacc 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 Debugging 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

To maximize the power or error detection in Yacc, I've tried to minimize the use of Lex so the parser can identify the error at a critical state. This enables the parser to generate expressive error message instead of only indicating the redundant characters. Still, as mentioned above, the literals were packed into an atomic token in the lexical analysis stage. This prevents, for example, a floating-point prefix `f` (as in `float foo = 1.1f;`) from being recognized as an identifier `f` (as in `int f;`).

5.2.1 From BNF to LALR(1)

When writing production rules, in addition to follow the given Java grammar, extra modification must also be made. Since Yacc is an LALR(1) parser, a Backus normal form (BNF) grammar cannot be directly applied to our implementation. There are five problems if we want to use Java BNF grammar in Yacc.

1. Ambiguous names. Look at a snippet in the original Java BNF grammar.

```
<PackageName>    ::= <Identifier> | <PackageName> . <Identifier>
<TypeName>       ::= <Identifier> | <packageName> . <Identifier>

<MethodName>     ::= <Identifier> | <AmbiguousName> . <Identifier>
<AmbiguousName> ::= <Identifier> | <AmbiguousName> . <Identifier>
```

Now, consider the input

```
class foo { int f() { whichami.
```

When the parser is considering the token `whichami`, with one token lookahead to `.` (the dot), it cannot tell whether `whichami` is a `package name` that qualifies a `type name`, as in

```
class foo { int f() { whichami.type newOBJ; }
```

Or a `ambiguous name` that qualifies a `method name`, as in

```
class foo { int f() { whichami.method(); }
```

Solution. We can replace `PackageName`, `TypeName`, `ExpressionName`, `MethodName`, and `AmbiguousName` with an single nonterminal `Name`.

```
Name                        : SimpleName
                            | QualifiedName
                            ;
SimpleName                  : Identifier
                            ;
QualifiedName               : Name '.' Identifier
                            ;
```

And a later stage of compiler analysis would figure it out the precise role or the names. In other words, this cannot be determinant in the parsing stage.

2. Various sets of modifiers. Look at a snippet in the original Java BNF grammar.

```
<FieldDeclaration>  ::= <FieldModifiers>? <Type> <VariableDeclarators> ;
<FieldModifiers>    ::= <FieldModifier> | <FieldModifiers> <FieldModifier>
<FieldModifier>     ::= public | protected | private | static | final | transient | volatile

<MethodDeclaration> ::= <MethodHeader> <MethodBody>
<MethodHeader>      ::= <MethodModifiers>? <ResultType> <MethodDeclarator> <Throws>?
<ResultType>        ::= <Type> | void
<MethodModifiers>   ::= <MethodModifier> | <MethodModifiers> <MethodModifier>
<MethodModifier>    ::= public | protected | private | static | abstract | final | synchronized | native
```

The problem is similar to the problem 1, the parser may not be able to determine whether it should reduce the token to a `field declaration` or a `method declaration`. Plus, since the number of modifiers can be zero to infinite, a lookahead-1 parser can never make a decision based on the next token.

Solution. Same as above, we delay the decision which set of modifiers should be applied to the later stage of analysis. Again, in other words, the parser cannot determine which set of modifiers is allowed.

3. Field declaration versus method declaration. After we merged various sets of modifiers into one unique token, the parser still cannot determine whether it is a `field declaration` or a `method declaration`. Consider the input

```
class foo { int whichami
```

When the parser is considering the token `int`, with one token lookahead to `whichami`, it cannot tell whether the following input would be which one of the following

```
class foo { int whichami = 0; }

class foo { int whichami(int arg); }
```

And shifts the `int` or reduce it to `ResultType`, respectively.

Solution. we can eliminate the `ResultType` production and have separate alternatives of `Type` and `void`.

```
MethodHeader                  : ModifiersOpt Type MethodDeclarator ThrowsOpt
                              | ModifiersOpt VOID MethodDeclarator ThrowsOpt
                              ;
```

So the parser can proceed to consider `whichami`, with one token lookahead to `=` or `(`, hence being determinant.

4. Array type or access. Look at a snippet in the original Java BNF grammar.

```
<ArrayType> ::= <Type> [ ]

<ArrayAccess> ::= <ExpressionName> [ <Expression> ] | <PrimaryNoNewArray> [ <Expression>]
```

Now, consider the input

```
class foo { foo() { whichami[
```

The parser is now considering `whichami`, with one token lookahead to `[`. It cannot determine whether this is a variable declaration with type of `whichami[]`, as in

```
class foo { foo() { whichami[] var; } }
```

Or an array access, as in

```
class foo { foo() { whichami[0] = 1; } }
```

Solution. we should separate alternatives for `ArrayType`

```
ArrayType                     : PrimitiveType '[' ']'
                              | Name '[' ']'
                              | ArrayType '[' ']'
                              ;
```

So the parser can reduce `whichami` to `Name` and proceed to consider `[`, with one token lookahead to `]` or `0`, hence being determinant.

5. Cast versus parenthesized expression. The corresponding grammar

```
<CastExpression> ::= ( <PrimitiveType> ) <UnaryExpression> | ( <ReferenceType> ) <UnaryExpressionNotPlusMinus>
```

Consider the following input

```
class foo { foo() { super((whichami)
```

Supposed the parser is considering `whichami`, with one token lookahead to `)`. The ambiguity lies between

```
class foo { foo() { super((whichami)); }

class foo { foo() { super((whichami)toBeCasted); }
```

Although few people would parenthesize a single variable, but it is legal. Therefore, the parser cannot decide which one above to take.

Solution. The solution is to eliminate the use of the nonterminal `ReferenceType` in the definition of `CastExpression`, which requires some reworking of both alternatives to avoid other ambiguities

```
CastExpression                  : '(' PrimitiveType DimsOpt ')' UnaryExpression %prec CAST
                                | '(' Expression ')' UnaryExpressionNotPlusMinus %prec CAST
                                | '(' Name Dims ')' UnaryExpressionNotPlusMinus %prec CAST
                                ;
```

5.2.2 Precedence & Associativity

Specifying the precedence and the associativity among operators is crucial for eliminating shift-reduce conflict. Following the Java's specification, the associativity can be specified by `%right`, `%left` or `%nonassoc` in the Yacc source file as below.

```
%right ASS MUL_ASS DIV_ASS MOD_ASS ADD_ASS SUB_ASS LS_ASS RS_ASS URS_ASS EMP_ASS XOR_ASS OR_ASS
%right '?' ':'
%left  OR
%left  AND
%left  '|'
%left  '^'
%left  '&'
%left  EQ NE
%nonassoc LE GE LT GT INSTANCEOF
%left  LS RS URS
%left  '+' '-'
%left  '*' '%' '/'
%right CAST NEW
%right PRE UMINUS NOT '~'
```

```
%nonassoc POST
%left  '[' ']' '.' '(' ')'
```

where the precedence is implied in the order of these lines. The latter an associativity rule is specified, the higher precedence the operators get.

Typically, the precedence of a production is determined by the last terminal's precedence. However, in some cases, the terminal itself is insufficient to expressive the proper precedence. For example, the `-` operator in the production of an unary expression, i.e., `-` as a negative sign instead of a subtraction operator. At this point, the precedence of the subtration operator is insufficient to express the one for an unary expression, so we can use `%prec` followed by a flag defined above, which should be `UMINUS` in this case. This technique can also be applied to postfix and prefix increment/decrement operators with shared tokens `++`/`--`.

```
PostIncrementExpression      : PostfixExpression INC %prec POST
                             ;
PostDecrementExpression      : PostfixExpression DEC %prec POST
                             ;
UnaryExpression              : PreIncrementExpression
                             | PreDecrementExpression
                             | '+' UnaryExpression %prec UMINUS
                             | '-' UnaryExpression %prec UMINUS
                             | UnaryExpressionNotPlusMinus
                             ;
PreIncrementExpression       : INC UnaryExpression %prec PRE
                             ;
PreDecrementExpression       : DEC UnaryExpression %prec PRE
                             ;
```

5.2.3 Optionals

As we would've seen the optional symbols in the Java grammar, which are followed by the question mark `?`. We will not be able to directly use that notation in the implementation in Yacc since there's no such feature. Instead, in this work, all of the optionals are marked with the prefix of `Opt`, which can be identify by the script `ListOptionals.sh` as listed in the section 2.

```
#!/bin/sh
if [ $# != 1 ]
then
    echo "Usage: $0 file (e.g. Parser.y)"
    exit 1
fi

egrep -o '[^ ]+Opt' $1 | sort | uniq \
| sed 's,\(.*\)Opt,& : \1 | /* empty */ ;,'
```

This shell script finds all optionals and generates the corresponding production for each.

```
ArgumentListOpt : ArgumentList | /* empty */ ;
BlockStatementsOpt : BlockStatements | /* empty */ ;
```

```
CatchesOpt : Catches | /* empty */ ;
ClassBodyDeclarationsOpt : ClassBodyDeclarations | /* empty */ ;
DimsOpt : Dims | /* empty */ ;
ExpressionOpt : Expression | /* empty */ ;
ExtendsInterfacesOpt : ExtendsInterfaces | /* empty */ ;
ForInitOpt : ForInit | /* empty */ ;
ForUpdateOpt : ForUpdate | /* empty */ ;
FormalParameterListOpt : FormalParameterList | /* empty */ { $$ = ""; };
IdentifierOpt : Identifier | /* empty */ ;
ImportDeclarationsOpt : ImportDeclarations | /* empty */ ;
InterfaceMemberDeclarationsOpt : InterfaceMemberDeclarations | /* empty */ ;
InterfacesOpt : Interfaces | /* empty */ ;
ModifiersOpt : Modifiers | /* empty */ ;
PackageDeclarationOpt : PackageDeclaration | /* empty */ ;
SuperOpt : Super | /* empty */ ;
SwitchLabelsOpt : SwitchLabels | /* empty */ ;
ThrowsOpt : Throws | /* empty */ ;
TypeDeclarationsOpt : TypeDeclarations | /* empty */ ;
```

5.3 Semantic analysis

The semantic analysis in this work only provides two simple checks, redefinition in the scope and unused variables.

5.3.1 Redefinition check

The redefinition check can be achieved by the help of a well-designed symbol table. In this work, the scope is recorded by a stack, and each scope has its own dedicated symbol table. The structure can be found in the header file `SymbolTable.h`.

```
/*************************************************
 * Data Structure                                *
 *************************************************
 *
 * Root
 * |
 * V
 * Scope 0 -> table[NUM_CHARSET] -> subroot 0 -> node 0 ... -> NULL
 * |                             |
 * V                             -> subroot 1 -> node 0 ... -> NULL
 * Scope 1                       |
 * |                             -> subroot 2 -> node 0 ... -> NULL
 * V                             .
 * Scope 2                       .
 * |                             .
 * V                             -> subroot NUM_CHARSET-1
 * Scope 3
 * .
 * .
```

```
 * .
 * |
 * V
 * NULL
 */
```

5.3.2 Unused variable check

In order to track the number or access to a symbol, we need to add a column `Occurrence` in each entry in the symbol table. The `Occurrence` only increments when a local variable is accessed, e.g., in an expression or at the left-hand side of an assignment statement. Only the method-like blocks are being examined in this work, i.e., the field in a class declaration would not be taken into account.

6 Screenshots

```
src — -zsh — 100×21
% ./JavaParser < ../TestingFiles/test1.java
     1  /* Test file: Perfect test file
     2   * Compute sum = 1 + 2 + ... + n
     3   */
     4  class sigma {
     5    // "final" should have const_expr
     6    final int n = 10;
     7    int sum, index;
     8
     9    main()
    10    {
    11      index = 0;
    12      sum = 0;
    13      while (index <= n)
    14      {
    15        sum = sum + index;
    16        index = index + 1;
    17      }
    18      print(sum);
    19    }
    20  }
```

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

```
src — -zsh — 100×15
% ./JavaParser < ../TestingFiles/test2.java
     1  /*Test file: Duplicate declare variable in the same scope*/
     2  class Point
     3  {
     4      static int counter ;
     5      int x, y ;
     6      /*Duplicate declare x*/
     7      int x ;
     7:12: semantic error: redefinition: `x`
     8      void clear()
     9      {
    10          x = 0 ;
    11          y = 0 ;
    12      }
    13  }
```

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

src — -zsh — 100×16

```
% ./JavaParser < ../TestingFiles/test3.java
     1  /*Test file of Syntax error: Out of symbol.  But it can go through*/
     2  class Point {
     3      int z;
     4      int x y ;
     4:12: syntax error: illegal field declaration: `y`
     5      /*Need  ',' before y*/
     6      float w;
     7  }
     8  class Test {
     9      int d;
    10      Point p = new Point()
    11      /*Need ';' at EOL*/
    12      int w,q;
    12:8: syntax error: illegal field declaration: `int`
    13  }
```

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

src — -zsh — 100×29

```
% ./JavaParser < ../TestingFiles/test4.java
     1  /*Test file: Duplicate declaration in different scope and same scope*/
     2  class Point
     3  {
     4          int x, y ;
     5          int p;
     6          boolean test()
     7          {
     8                  /*Another x, but in different scopes*/
     9                  int x;
    10                  /*Another x in the same scope*/
    11                  char x;
    11:10: semantic error: redefinition: `x`
    12                  {
    13                          boolean w;
    14                  }
    15                  /*Another w in the same scope*/
    16                  int w;
    17          }
    18  }
    19  class Test
    20  {
    21          /*Another p, but in different scopes*/
    22          Point p = new Point();
    23  }

 warning: unused variable: `w`
 warning: unused variable: `x`
 warning: unused variable: `w`
```

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

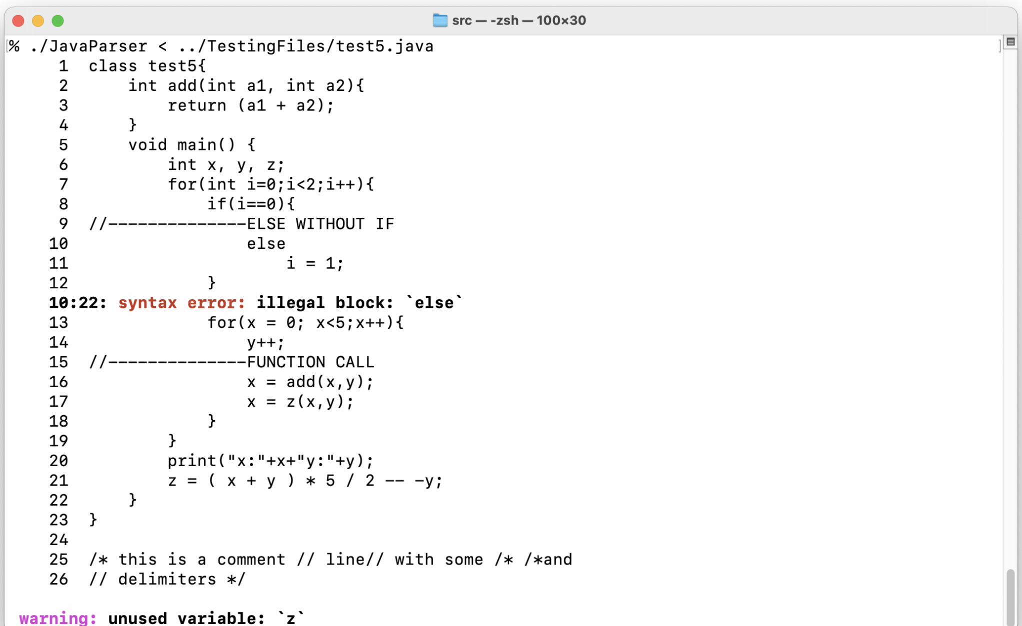
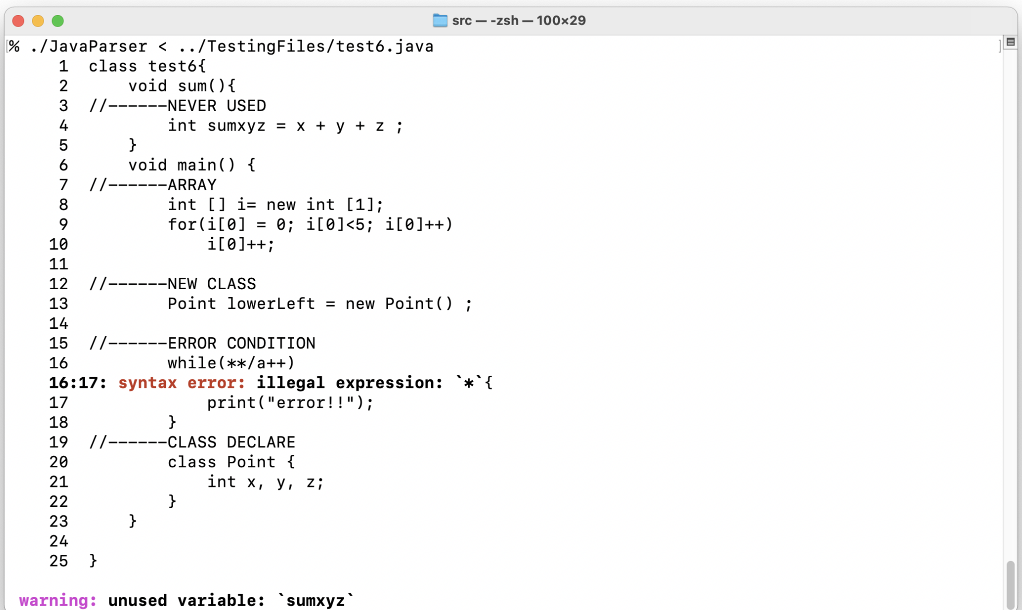


Figure 5: The output of parsing `test5.java` with `DEBUG=1`



```
src — -zsh — 100×29
% ./JavaParser < ../TestingFiles/test6.java
     1  class test6{
     2      void sum(){
     3  //------NEVER USED
     4          int sumxyz = x + y + z ;
     5      }
     6      void main() {
     7  //------ARRAY
     8          int [] i= new int [1];
     9          for(i[0] = 0; i[0]<5; i[0]++)
    10              i[0]++;
    11
    12  //------NEW CLASS
    13          Point lowerLeft = new Point() ;
    14
    15  //------ERROR CONDITION
    16          while(**/a++)
    16:17: syntax error: illegal expression: `*`{
    17              print("error!!");
    18          }
    19  //------CLASS DECLARE
    20          class Point {
    21              int x, y, z;
    22          }
    23      }
    24
    25  }

 warning: unused variable: `sumxyz`
```

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