Compilation - Programming Assignment 2

Team:
Ehud Tamir 036934644
Arbel Zinger 034666610
Inon Holdengreber 026504183

Code Structure and description of major classes

Compiler.java is our program's main class. This class handles the input. It calls the parser to parse the source file. The Parser creates an AST for the source file, and in case an error occurs it throws an exception. If the user chooses to print the AST it uses the PrettyPrinter class in order to print the AST. The library file (either the default file or the one the user inputs), is parsed as well, and added into the program AST (as a child of Root).
The main method in Compiler.java accepts 3 arguments at most: A path to a source file, and optionally a path to a library file and a flag, "-print-ast".
Lexer.java is the lexical analysis scanner, class Lexer, is generated automatically using JFlex from the specification file IC.lex. The Lexer scans the input files and generates instances of class Token, which are returned to the caller (class Compiler in this case). In case of an invalid token the Lexer throws a LexicalError.
IC.lex contains our rules for tokenization. A full list of tokens and their respective regular expressions is given below.
sym.java defines a representation of token classes by integers. For example, the EOF token is represented by the constant sym.EOF, whose value is 0. This file is created automatically by the Java CUP Library according to the Terminals defined in the IC.cup file.
Token.java holds the representation of a token in a given input file. A token has 4 characteristics: The token ID (numeric representation of the token's name), the line where the token appears, the token's value (if applicable) and the token's name. The token's name is resolved according to its numeric value.
LexicalError.java implements an exception for errors in lexical analysis. Each instance of this exception has a line number where the error appeared, the string that caused the error and a custom message, sent by the Lexer.
Parser.java is the parser for IC language. It was generated automatically by the Java CUP library, by the grammar we defined in the IC.cup file. The parser receives tokens that the Lexer reads, and generates an AST (abstract syntax tree) using a set of rules that was defined in the IC.cup file (i.e the grammar of the language).The creation of the AST is done by instantiating object from the IC.AST package, more on this package will be explained later. If during the run of the Parser an error occurs, a syntax error will be thrown.
IC.cup is the input file for the Java CUP library. It contains a set of rules and definitions for creating the IC grammar.
Notes:

1. We perform no range checking on the integers.
2. Bonus 1: We perform error recovery on statements, method declaration and class declaration.
3. Bonus 2: We implemented the grammar fix for if, else and while statements.

LibraryParser.java this file is also created automatically by the Java CUP library. This java file receives a .sig file and parses it to an AST according to the grammar that is defined in the Library.cup file. It uses the same sym.java file that was created by the Java CUP during its run on IC.cup. The .sig file defines external methods that can be used with the IC language.

Library.cup is an input file for Java CUP library. it defines a set of rules and definition of the Library grammar. After running the Java CUP library on this file we will get LibraryParser.java as the output.

SyntaxError.java implements an exception for errors in syntax analysis. Each instance of this exception generates an error message containing the line number and token where the error occured.

IC.AST package includes various classes used for the creation of the AST. We used all the files that were given to us in the assignment skeleton, and added a few more: FieldMethodList.java. this class hold 2 Lists: one of Method class and the other from Field class. We needed it to define the fields and methods declared in a class. We also added 3 classes for error recovery mechanism: EmptyStatement, ErrorClass and ErrorMethod.

IC package includes the Compiler.java, BinaryOps.java, DataTypes.java, LiteralTypes.java and UnaryOps.java. All those files except Compiler.java were given to us in the skeleton.zip.

Class Hierarchy

Package IC:

- class **Compiler**
- enum **BinaryOps**
- enum **DataTypes**
- enum **LiteralTypes**
- enum **UnaryOps**

Package IC.AST

- class **ArrayLocation** extends **Location**
- class **Assignment** extends **Statement**
- abstract class **ASTNode**
- abstract class **BinaryOp** extends **Expression**
- class **Break** extends **Statement**
- abstract class **Call** extends **Expression**
- class **CallStatement** extends **Statement**
- class **Continue** extends **Statement**
- class **EmptyStatement** extends **Statement**
- class **ErrorClass** extends **ICClass**
- class **ErrorMethod** extends **Method**
- abstract class **Expression** extends **ASTNode**

class **ExpressionBlock** extends **Expression**
class **Field** extends **ASTNode**
class **FieldMethodList** extends **ASTNode**
class **Formal** extends **ASTNode**
class **ICClass** extends **ASTNode**
class **If** extends **Statement**
class **Length** extends **Expression**
class **LibraryMethod** extends **Method**
class **Literal** extends **Expression**
class **LocalVariable** extends **Statement**
abstract class **Location** extends **Expression**
class **LogicalBinaryOp** extends **BinaryOp**
class **LogicalUnaryOp** extends **UnaryOp**
class **MathBinaryOp** extends **BinaryOp**
class **MathUnaryOp** extends **UnaryOp**
abstract class **Method** extends **ASTNode**
abstract class **New** extends **Expression**
class **NewArray** extends **New**
class **NewClass** extends **New**
class **PrettyPrinter** implements **Visitor**
class **PrimitiveType** extends **Type**
class **Program** extends **ASTNode**
class **Return** extends **Statement**
abstract class **Statement** extends **ASTNode**
class **StatementsBlock** extends **Statement**
class **StaticCall** extends **Call**
class **StaticMethod** extends **Method**
class **This** extends **Expression**
abstract class **Type** extends **ASTNode**
abstract class **UnaryOp** extends **Expression**
class **UserType** extends **Type**
class **VariableLocation** extends **Location**
class **VirtualCall** extends **Call**
class **VirtualMethod** extends **Method**
interface **Visitor**
class **While** extends **Statement**

Package IC.Parser:

class **Lexer** implements **java_cup.runtime.Scanner**
class **LexicalError** extends **Exception**
class **sym**
class **Token** extends **java_cup.runtime.Symbol**

class **SyntaxError** extends **Exception**
class **Parser** extends **java_cup.runtime.lr_parser**
class **LibraryParser** extends **java_cup.runtime.lr_parser**

Testing Strategy

Our tests, consisting on both syntactically correct and on syntactically incorrect files, focused on verifying the following aspect of the syntax analysis:

- Correct behavior (precedence, associativity, etc) of binary and unary operations, both mathematical and logical.
- Edge cases: Empty class, empty file.
- Correct expression semantics: Assignments, calling methods, strings, arrays, etc.
- Correct behavior for if/else statements.
- Checking for errors inside/outside of methods.
- Testing the error recovery we wrote (bonus 1)
- Testing the “Nullified If” error (bonus 2)

In addition, more edge cases were tested. Such as supplying the program with an illegal amount of arguments, specifying a file that does not exist and more.

The Grammar

===== Terminals =====
[0]EOF [1]error [2]ASSIGN [3]BOOLEAN [4]BREAK
[5]CLASS [6]COMMA [7]CONTINUE [8]DIVIDE [9]DOT
[10]ELSE [11]EQUAL [12]EXTENDS [13]FALSE [14]GT
[15]GTE [16]IF [17]INT [18]LAND [19]LB
[20]LCBR [21]LENGTH [22]LNEG [23]LOR [24]LP
[25]LT [26]LTE [27]MINUS [28]MOD [29]MULTIPLY
[30]NEQUAL [31]NEW [32]NULL [33]PLUS [34]RB
[35]RCBR [36]RETURN [37]RP [38]SEMI [39]STATIC
[40]STRING [41]THIS [42]TRUE [43]UMINUS [44]VOID
[45]WHILE [46]INTEGER [47]CLASS_ID [48]ID [49]QUOTE

===== Non terminals =====
[0]program [1]classDecl [2]classList [3]fmList [4]fieldList
[5]idList [6]method [7]stmtList [8]formal [9]formalList
[10]type [11]stmt [12]expr [13]exprList [14]location
[15]call [16]staticCall [17]virtualCall [18]binExpr [19]unExpr
[20]literal [21]varDecl

===== Precedence =====
precedence right ASSIGN;
precedence left LOR;

precedence left LAND;
precedence left EQUAL, NEQUAL;
precedence left LT, LTE, GT, GTE;
precedence left PLUS, MINUS;
precedence left MULTIPLY, DIVIDE, MOD;
precedence right UMINUS, LNEG;
precedence left LB, RB, LP, RP, DOT;
precedence left ELSE;

===== Productions =====
[0] program ::= classList
[1] $START ::= program EOF
[2] classList ::= classList classDecl
[3] classList ::=
[4] classDecl ::= CLASS CLASS_ID EXTENDS CLASS_ID LCBR fmList RCBR
[5] classDecl ::= CLASS CLASS_ID LCBR fmList RCBR
[6] classDecl ::= error LCBR fmList RCBR
[7] fmList ::= fmList fieldList
[8] fmList ::= fmList method
[9] fmList ::=
[10] fieldList ::= type idList SEMI
[11] idList ::= ID
[12] idList ::= idList COMMA ID
[13] method ::= STATIC type ID LP formalList RP LCBR stmtList RCBR
[14] method ::= STATIC VOID ID LP formalList RP LCBR stmtList RCBR
[15] method ::= type ID LP formalList RP LCBR stmtList RCBR
[16] method ::= VOID ID LP formalList RP LCBR stmtList RCBR
[17] method ::= error LCBR stmtList RCBR
[18] formalList ::= formalList COMMA formal
[19] formalList ::= formal
[20] formalList ::=
[21] formal ::= type ID
[22] type ::= INT
[23] type ::= BOOLEAN
[24] type ::= STRING
[25] type ::= CLASS_ID
[26] type ::= type LB RB
[27] stmtList ::= stmtList stmt
[28] stmtList ::= stmtList varDecl
[29] stmtList ::=
[30] stmt ::= location ASSIGN expr SEMI

[31] stmt ::= call SEMI
[32] stmt ::= RETURN expr SEMI
[33] stmt ::= RETURN SEMI
[34] stmt ::= IF LP expr RP stmt ELSE stmt
[35] stmt ::= IF LP expr RP stmt
[36] stmt ::= WHILE LP expr RP stmt
[37] stmt ::= BREAK SEMI
[38] stmt ::= CONTINUE SEMI
[39] stmt ::= LCBR stmtList RCBR
[40] stmt ::= error SEMI
[41] varDecl ::= type ID ASSIGN expr SEMI
[42] varDecl ::= type ID SEMI
[43] exprList ::=
[44] exprList ::= exprList COMMA expr
[45] exprList ::= expr
[46] expr ::= location
[47] expr ::= call
[48] expr ::= THIS
[49] expr ::= NEW CLASS_ID LP RP
[50] expr ::= NEW type LB expr RB
[51] expr ::= expr DOT LENGTH
[52] expr ::= binExpr
[53] expr ::= unExpr
[54] expr ::= literal
[55] expr ::= LP expr RP
[56] binExpr ::= expr PLUS expr
[57] binExpr ::= expr MINUS expr
[58] binExpr ::= expr MULTIPLY expr
[59] binExpr ::= expr DIVIDE expr
[60] binExpr ::= expr MOD expr
[61] binExpr ::= expr LAND expr
[62] binExpr ::= expr LOR expr
[63] binExpr ::= expr LT expr
[64] binExpr ::= expr LTE expr
[65] binExpr ::= expr GT expr
[66] binExpr ::= expr GTE expr
[67] binExpr ::= expr EQUAL expr
[68] binExpr ::= expr NEQUAL expr
[69] unExpr ::= MINUS expr
[70] unExpr ::= LNEG expr
[71] call ::= staticCall

[72] call ::= virtualCall
[73] staticCall ::= CLASS_ID DOT ID LP exprList RP
[74] virtualCall ::= expr DOT ID LP exprList RP
[75] virtualCall ::= ID LP exprList RP
[76] location ::= ID
[77] location ::= expr DOT ID
[78] location ::= expr LB expr RB
[79] literal ::= INTEGER
[80] literal ::= QUOTE
[81] literal ::= TRUE
[82] literal ::= FALSE
[83] literal ::= NULL