The Hog Programming Language

Jason Halpern	Samuel Messing	Benjamin Rapaport
jrh2170	sbm2158	bar2150
Testing/Validation	Project Manager	System Architect

Kurry Tran	Paul Tylkin
klt2127	pt2302
System Integrator	Language Guru

May 6, 2012

Contents

1 Introduction **4**
- 1.1 Taming the Elephant . . . 4
  - 1.1.1 Data-Oriented . . . 4
  - 1.1.2 Simple . . . 5
  - 1.1.3 Distributed . . . 5
  - 1.1.4 Readable . . . 6
  - 1.1.5 A Sample Program . . . 6

2 Tutorial **8**

3 Language Reference Manual **9**
- 3.1 Introduction . . . 9
  - 3.1.1 The MapReduce Framework . . . 9
  - 3.1.2 The Hog Language . . . 10
  - 3.1.3 The "Ideal" Hog User . . . 11
- 3.2 Syntax Notation . . . 12
- 3.3 Program Structure . . . 12
  - 3.3.1 Overall Structure . . . 12
  - 3.3.2 `@Functions` . . . 12
  - 3.3.3 `@Map` . . . 14
  - 3.3.4 `@Reduce` . . . 15
  - 3.3.5 `@Main` . . . 15
- 3.4 Lexical Conventions . . . 16
  - 3.4.1 Tokens . . . 16
  - 3.4.2 Comments . . . 16
  - 3.4.3 Identifiers . . . 16
  - 3.4.4 Keywords . . . 17
  - 3.4.5 Constants . . . 17
  - 3.4.6 Text Literals . . . 18
  - 3.4.7 Variable Scope . . . 18
  - 3.4.8 Argument Passing . . . 18
  - 3.4.9 Evaluation Order . . . 18
- 3.5 Types . . . 18
  - 3.5.1 Basic Types . . . 18

3.5.2 Derived Types (Collections) . . . 19
3.5.3 Type Conversions . . . 19
3.6 Expressions . . . 19
3.6.1 Operators . . . 19
3.7 Declarations . . . 21
3.7.1 Type Specifiers . . . 21
3.7.2 Declarations . . . 22
3.8 Statements . . . 23
3.8.1 Expression Statement . . . 23
3.8.2 Compound Statement (Blocks) . . . 23
3.8.3 Flow-Of-Control Statements . . . 23
3.8.4 Iteration Statements . . . 23
3.9 Built-in Functions . . . 25
3.9.1 System-level Built-ins . . . 25
3.9.2 Object-level Built-ins . . . 25
3.10 System Configuration . . . 28
3.11 Compilation Structure . . . 28
3.12 Linkage and I/O . . . 29
3.12.1 Usage . . . 29
3.12.2 Example . . . 29
3.13 Exception Handling . . . 30
3.13.1 Compile-time Errors . . . 30
3.13.2 Internal Run-time Exceptions . . . 31
3.14 Grammar . . . 32

4 Project Plan **41**
4.1 Development Process . . . 41
4.1.1 Simplicity of Build System . . . 41
4.1.2 Similarity of Modules . . . 42
4.1.3 Document Everything . . . 42
4.1.4 Distributed Version Control . . . 42
4.1.5 Verbose Logging . . . 43
4.2 Roles and Responsibilities . . . 43
4.3 Hog's Developer Style Sheet . . . 44
4.4 Project Timeline . . . 44
4.5 Project Log . . . 45

5 Language Evolution **47**

6 Translator Architecture **48**
6.1 Architecture . . . 48
6.2 Module Authors . . . 49

7 Development and Run-Time Environment **51**

8 Test Plan **55**

9 Conclusions **59**
9.1 Lessons Learned 59
9.1.1 Jason's Lessons 60
9.1.2 Sam's Lessons 61
9.1.3 Ben's Lessons 61
9.1.4 Kurry's Lessons 61
9.1.5 Paul's Lessons 62
9.2 Advice for Other Teams 62
9.3 Suggestions for Instructor 62

A Code Listing **63**

B Complete Source Code **69**
B.1 `back_end` package 69
B.1.1 `CodeGeneratingVisitor.java` 69
B.1.2 `ErrorCheckingVisitor.java` 90
B.1.3 `SymbolTableVisitor.java` 98
B.1.4 `TypeCheckingVisitor.java` 108
B.1.5 `Visitor.java` 118
B.2 `front_end` package 119
B.2.1 `ConsoleLexer.java` 119
B.2.2 `Hog.java` 121
B.2.3 `Lexer.jflex` 125
B.2.4 `Parser.cup` 128
B.3 `test` package 148
B.3.1 `AbstractSyntaxTreeTester.java` 148
B.3.2 `CodeGeneratingTester.java` 151
B.3.3 `LexerTester.java` 155
B.3.4 `NodeTester.java` 200
B.3.5 `ParserTester.java` 206
B.3.6 `SymbolTableTexter.java` 207
B.3.7 `TypesCheckingTester.java` 210
B.3.8 `TypesTester.java` 213

Chapter 1

Introduction

1.1 Taming the Elephant

As data sets have grown in size, so have the complexities of dealing with them. For instance, consider wanting to generate counts for all the words in War and Peace by means of distributed computation. Writing in Java and using Hadoop MapReduce (TM),[1] a simple solution takes over 50 lines of code, as the programmer is required to specify intermediate objects not directly related to the desired computation, but required simply to get Hadoop to function properly. Our goal is to produce a language that can express the same computation in about 10 lines.

Hog is a **data-oriented**, high-level, scripting language for creating MapReduce[**?**] programs. Used alongside Hadoop, Hog enables users to efficiently carry out **distributed** computation. Hadoop MapReduce is an open-source framework for carrying out distributed computation, which is especially useful for working with large data sets. While it is possible to write code to carry out computations with Hadoop directly, the framework requires users to specify low-level details that are often irrelevant to their desired goal.

By building a scripting language on top of Hadoop, we aim to simplify the process. Built around a **simple** and highly **readable** syntax, Hog will let users focus on *what* computations they want done, and not *how* they want to do them. Hog takes care of all the low-level details required to run computations on Hadoops distributed network. All a user needs to do is tell Hog the location of their valid Hadoop instance, and Hog will do the rest.

1.1.1 Data-Oriented

Hog is a powerful language that allows for the efficient handling of structured, unstructured and semi-structured data. Specifically, Hog simplifies the process

[1] `http://hadoop.apache.org/`

of writing programs to handle the distributed processing of data-intensive applications. Programmers using Hog only have to express the steps for processing the data in the Map and Reduce functions without having to be concerned with relations and the constraints imposed by a traditional database schema. Hog also provides control flow structures to manipulate this data. In addition, Hog frees a programmer from having to write each step in a data processing task since many of those low-level processing details are handled by the language and the system.

Hog uses Hadoop MapReduce (TM), an open-source MapReduce framework written in Java. Hadoops run time system takes care of the details of partitioning the input data, scheduling the programs execution across machines, counteracting machine failures, and managing inter-machine communication. Hadoop also distributes data to machines and tries to collocate chunks of data with the nodes that need it, therefore maximizing data locality and giving good performance.

1.1.2 Simple

To write a simple word count program in Java using the Hadoop framework requires over 59 lines of code.[2] The same program written in Hog requires just 10 lines. The discrepancy comes from the fact that Hog takes care of the low-level details required to correctly communicate and interact with the Hadoop framework. This allows users to enhance the expressive potential of their programs, without sacrificing power. All that Hog requires a user to do is specify the location of their valid Hadoop instance, write a map function to process a segment of data, write a reduce function to combine the results, and Hog takes care of the rest.

1.1.3 Distributed

As datasets have exploded in size, programmers have had to deal with the challenge of writing programs for distributed systems that process data in a time-efficient manner. One of the benefits of using Hadoop is that it allows programmers to write parallel programs without needing to understand the intricacies of how the distributed computations are implemented. This benefit is one of the key reasons for the widespread adoption of Hadoop. Since Hog operates as a layer on top of Hadoop, and abstracts away even more of the implementation details of the distributed system, we remain committed to the ideal of a fully distributed language that is easy for programmers to use. Once again, this is paramount to Hogs focus on what computations are being done and not how they are being done.

[2] `http://hadoop.apache.org/common/docs/current/mapred_tutorial.html`

1.1.4 Readable

The syntax of Hog is designed to make programs as readable as possible. Hog is specifically developed to make simple calculations easy to carry out. While Hog may not be the best solution for highly sophisticated analysis, an individual desiring to learn more about Hadoop and the MapReduce technique will find Hog an inviting environment to get started. Hogs syntax is simple enough that someone with only a small amount of programming experience should have no trouble understanding the basics of what is happening in a sample Hog program. Our goal is to let users think about their data first and foremost, and not on using or learning an esoteric syntax. Where possible, our syntax decisions prefer simplicity over complexity.

1.1.5 A Sample Program

```
1  @Map (int lineNum, text line) -> (text, int) {
2    # for every word on this line, emit that word and the number 1
3    foreach text word in line.tokenize(" ") {
4       emit(word, 1); }
5  }

6  @Reduce (text word, iter<int> values) -> (text, int) {
7    # initialize count to zero
8    int count = 0;
9    while( values.hasNext() ) {
10     # for every instance of '1' for this word, add to count.
11     count = count + values.next(); }
12   # emit the count for this particular word
13   emit(word, count);
14 }

15 @Main {
16   # call map reduce
17   mapReduce();
18 }
```

This program generates a count of the number of instances of every word in a file. Here is the first 50 lines of output generated by `WordCount.hog` called on the full text of *War and Peace*[3]:

```
31784 the
21049 and
16389 to
14895 of
10056 a
```

[3] `http://www.gutenberg.org/ebooks/2600/` was used as the input text

```
8314 in
7847 he
7645 his
7425 that
7255 was
5540 with
5316 had
4492 not
4209 at
4162 her
4009 I
3757 it
3744 as
3495 on
3488 him
3308 for
3134 is
2888 but
2762 The
2718 you
2636 said
2620 she
2526 from
2390 all
2387 were
2354 be
2333 by
2031 who
2006 which
1910 have
1812 He
1777 one
1727 they
1693 this
1645 what
1566 or
1561 an
1554 Prince
1550 so
1541 Pierre
1466 been
1439 did
1424 up
1409 their
1342 woul
```

Chapter 2

Tutorial

Use your updated tutorial.

Chapter 3

Language Reference Manual

3.1 Introduction

As data sets have grown in size, so have the complexities of dealing with them. For instance, consider wanting to generate counts for all the words in *War and Peace* by means of distributed computation. Writing in Java and using Hadoop MapReduce (TM), a simple solution takes over 50 lines of code, as the programmer is required to specify intermediate objects not directly related to the desired computation, but required simply to get Hadoop to function properly. Our language can express the same computation in about 15 lines.

3.1.1 The MapReduce Framework

With the explosion in the size of datasets that companies have had to manage in recent years, there are many new challenges that they face. Many companies and organizations have to handle the processing of datasets that are terabytes or even petabytes in size. The first challenge in this large-scale processing is how to make sense of all this data. More importantly, the question is how they can process and manipulate the data in a time-efficient and reliable manner. The second challenge is how they handle this across their distributed systems. Writing distributed, fault-tolerant programs requires a high level of expertise and knowledge of parallel systems.

In response to this need, a group of engineers at Google developed the MapReduce framework in 2004. This high-level framework can be used for a variety of tasks, including handling search queries, indexing crawled documents, and processing logs. The software framework was developed to handle computations on massive datasets that are distributed across hundreds or even thousands of machines. The motivation behind MapReduce was to create a unified framework that abstracted away many of the low level details from programmers, so they would not have to be concerned with how the data is distributed, how the computation is parallelized and how all of this is done in a fault tolerant manner.

The MapReduce framework partitions input data across different machines, so that the computations are initially performed on smaller sets of data distributed across the cluster. Each cluster has a master node that is responsible for coordinating the efforts among the slave nodes. Each slave node sends periodic heartbeats to the master node so it can be aware of progress and failure. In the case of failure, the master node can reassign tasks to other nodes in the cluster. In conjunction with the underlying MapReduce framework created at Google, the company also had to build the distributed Google File System (GFS). This file system "allows programs to access files efficiently from any computer, so functions can be mapped everywhere."[**?**] GFS was designed with the same goals as other distributed file systems, including "performance, scalability, reliability and availability."[**?**] Another key aspect of the GFS design is fault tolerance and this is achieved by treating failures as normal and optimizing for "huge files that are mostly appended to and then read."[**?**]

Within the framework, a programmer is responsible for writing both map and reduce functions. The map function is applied to all of the input data "in order to compute a set of intermediate key/value pairs."[**?**] In the map step, the master node partitions the input data into smaller problems and distributes them across the worker nodes in the cluster. This step is applied in parallel to all of the input that has been partitioned across the cluster. Then, the reduce step is responsible for collecting all the processed data from the slave nodes and formatting the output. The reduce function is carried out over all the values that have the same key such that each key has a single value. which is the answer to the problem MapReduce is trying to solve. The output is done to files in the distributed file system.

The use of "a functional model with user-specified map and reduce operations allows (Google) to parallelize large computations easily and to use re-execution as the primary mechanism for fault tolerance."[**?**] A programmer only has to specify the functions described above and the system handles the rest of the details. Figure 3.1.1 illustrates the execution flow of a MapReduce program.

3.1.2 The Hog Language

Hog is a **data-oriented**, **high-level**, scripting language for creating MapReduce programs. Used alongside Hadoop, Hog enables users to efficiently carry out **distributed** computation. Hadoop MapReduce is an open-source implementation of the MapReduce framework, which is especially useful for working with large data sets. While it is possible to write code to carry out computations with Hadoop directly, the framework requires users to specify low-level details that are often irrelevant to their desired goal.

By building a scripting language on top of Hadoop, we aim to simplify the process. Built around a **simple** and highly **readable** syntax, Hog will let users focus on what computations they want done, and not how they want to do them. Hog takes care of all the low-level details required to run computations

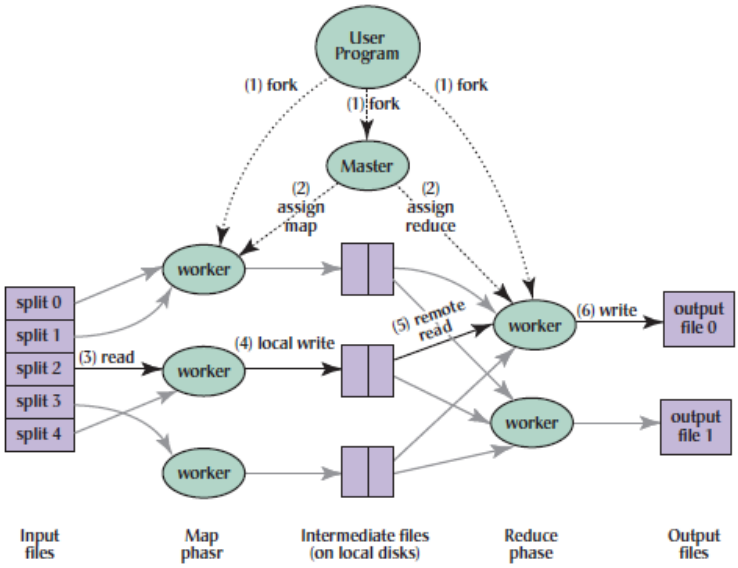


Figure 3.1: Overview of the MapReduce program, from [**?**].

on Hadoops distributed network. All a user needs to do is tell Hog the location of their valid Hadoop instance, and Hog will do the rest.

We intentionally have restricted the scope of Hog to deal with specific problems. For example, Hog only supports reading and writing plaintext files. While these limitations sacrifice the generality of the language, they promote ease of use.

Guiding Principles

The guiding principles of Hog are:

- Anyone can MapReduce
- Brevity over verbosity
- Simplicity over complexity

3.1.3 The "Ideal" Hog User

Hog was designed with a particular user in mind: one that has already learned the basics of programming in a different programming language (such as Java or Python), but is inexperienced with distributed computation and can benefit from a highly structured framework for writing MapReduce programs. The language was designed with the goal of making learning how to write MapReduce programs as easy as possible. However, the user should be adept with

programming concepts such as program structure, control flow (iteration and conditional operators), evaluation of boolean expressions, etc.

3.2 Syntax Notation

In the syntax notation used throughout the Hog manual, different syntactic categories are noted by *italic type*, and literal words and characters are in `typewriter style`. When specific terms are introduced, ***emboldened, italicized font*** is used.

3.3 Program Structure

3.3.1 Overall Structure

Every Hog program consists of a single source file with a .hog extension. This source file must contain three sections: `@Map`, and `@Reduce`, and `@Main` and can also include an optional `@Functions` section. These sections must be included in the following order:

```
@Functions {
   .
   .
   .
}
@Map <type signature> {
   .
   .
   .
}
@Reduce <type signature> {
   .
   .
   .
}
@Main {
   .
   .
   .
}
```

3.3.2 `@Functions`

At the top of every Hog program, the programmer has the option to define functions in a section called `@Functions`. Any function defined in this section can be called from any other section of the program, including `@Map`,

`@Reduce`, and `@Main` and can also be called from other functions defined in the `@Functions` section. The section containing the functions begins with the keyword `@Functions` on its own line, followed by the function definitions.

Function definitions have the form:

type functionName `(` *parameterList* `)` `{`
    expressionList;
`}`

where,

parameterList → *parameter* `,` *parameterList* | *parameter*

The return type can be any valid Hog type. The rules regarding legal function names are identical to those regarding legal variable identifiers. Each parameter in the parameter list consists of a valid Hog type followed by the name of the parameter, which must also follow the naming rules for identifiers. Parameters in the parameter list are separated by commas. The `@Functions` section ends when the next Hog section begins.

A complete example of an `@Functions` section:

```
@Functions {
  int min(int a, int b) {
    if (a < b) {
      return a;
    } else {
      return b;
    }
  }

  list<int> reverseList(list<int> oldList) {
    list<int> newList;
    for (int i = oldList.size() - 1; i >= 0; i--;) {
      newList.add(oldList.get(i));
    }
    return newList;
  }
}
```

User-defined functions can make reference to other user-defined functions. However, function names cannot be overloaded (i.e. it is not possible to use the same function name with a parameter list that differs in the number of arguments or argument types). Disallowing function overloading is a design choice consistent with Hog's guiding principle of simplicity.

3.3.3 `@Map`

The map function in a MapReduce program takes as input key-value pairs, performs the appropriate calculations and procedures, and emits intermediate key-value pairs as output. Any given input pair may map to zero, one, or multiple output pairs. The `@Map` section defines the code for the map function.

The `@Map` header must be followed by the signature of the map function, and then the body of the map function as follows:

```
@Map ( type identifier, type identifier ) -> ( type, type ) {
    .
    .
    .
}
```

The first *type identifier* defines the ***key*** and the second defines the ***value*** of the input key-value pair to the `@Map` function. The identifiers specified for the key and value can be made reference to later within the `@Map` block. The `@Map` signature is followed by an arrow and another key-value pair, defining the types of the output of the map function. Notice that identifiers are not specified for the output key and value (said to be ***unnamed***), as these pairs are only produced at the end of the map function.

The map function can include any number of calls to `emit()`, which outputs the resulting intermediate key-value pairs for use by the function defined in the `@Reduce` section. The types of the values passed to the `emit()` function must agree with the signature of the output key-value pair as defined in the `@Map` type signature. All output pairs from the map function are subsequently grouped by key by the framework, and passed as input to the `@Reduce` function.

Note: In the current version of the language, the only configuration available is for a file to be passed into the map function one line at a time, with the line of text being the value, and the corresponding line number as the key. This requires that the input key/value pair to the map function is of type `(int keyname, text valuename)`. Extending this to allow for other input formats is a future goal of the Hog language.

The following is an example of a complete `@Map` section for a program that counts the number of times each word appears in a set of files. The map function receives a single line of text, and for each word in the line (as delineated by whitespace), it emits the word as the key with a value of one. By emitting the word as the key, we can allow the framework to group by the word, thus calling the reduce function for every word.

```
@Map (int lineNum, text line) -> (text, int) {
    # for every word on this line, emit that word and the number 1
    foreach text word in line.tokenize(" ") {
        emit(word, 1);
     }
```

}

3.3.4 @Reduce

The reduce function in a MapReduce program takes a list of values that share the same key, as emitted by the map function, and outputs a smaller set of values to be associated with another key. The input and output keys do not have to match, though they often do.

The setup for the reduce section is similar to the map section. However, the input value for any reduce function is always an iterator over the list of values associated with its key. The type of the key must be the same as the type of the key emitted by the map function. The iterator must be an iterator over the type of the values emitted by the map function.

```
@Reduce ( type identifier, type identifier ) -> ( type, type ) {
    .
    .
    .
}
```

As with the map function, the reduce function can emit as many key/value pairs as the user would like. Any key/value pair emitted by the reduce function is recorded in the output file.

Below is a sample `@Reduce` section, which continues the word count example, and follows the `@Map` sample introduced in the previous section.

```
@Reduce (text word, iter<int> values) -> (text, int) {
  # initialize count to zero
  int count = 0;
  while (values.hasNext()) {
    # for every instance of '1' for this word, add to count
    count = count + values.next();
  }
  # emit the count for this particular word
  emit(word, count);
}
```

3.3.5 @Main

The `@Main` section defines the code that is the entry point to a Hog program. In order to run the MapReduce program defined by the user in the previous sections, `@Main` must contain a call to the system-level built-in function `mapReduce()` , which calls the `@Map` and `@Reduce` functions. Other arbitrary code can be run from the `@Main` section as well. In the current version of the

language, `@Main` does not have access to the results of the MapReduce program resulting from a call to `mapReduce()`. Therefore, it is quite common for the `@Main` section to contain the call to `mapReduce()` and nothing else.

Below is a sample `@Main` section which prints to the standard output and runs a map reduce job.

```
@Main {
      print("Starting mapReduce job.\n");
      mapReduce();
      print("mapReduce complete.\n");
}
```

3.4 Lexical Conventions

3.4.1 Tokens

The classes of tokens include the following: identifiers, keywords, constants, string literals, operators, and separators. Blanks, tabs, newlines, and comments are ignored. If the input is separated into tokens up to a given character, the next token is the longest string of characters that could represent a token.

3.4.2 Comments

Multi-line comments are identified by the enclosing character sequences `#{` and `}#`. Anything within these enclosing characters is considered a comment, and is completely ignored by the compiler. For example,

```
int i = 0;
#{ these are block
    comments and are ignored
     by the compiler }#
i++;
```

In the above example, the text `these are block comments \n comments and are ignored \n by the complier` is completely ignored during compilation. Compilation goes directly from the line `int i = 0;` to the line `i++;`.

Single-line comments are defined to be strings of text included between a '`#`' symbol on the left-hand side and a newline character ('`\n`') on the right-hand side.

3.4.3 Identifiers

A valid identifier in Hog is a sequence of contiguous letters, digits, or underscores, which are used to distinguish declared entities, such as methods, parameters, or variables from one another. A valid identifier also provide a means of

determining scope of an entity, and helps to determine whether the same valid identifier in another scope refers to the same entity. The first character of an identifier must not be a digit. Valid identifiers are case sensitive, so `foo` is not the same identifier as `Foo`.

3.4.4 Keywords

The following words are reserved for use as keywords, and may not be redefined by the programmer:

`add`	`final`	`iter`	`removeAll`
`and`	`for`	`list`	`return`
`bool`	`foreach`	`Map`	`size`
`break`	`Functions`	`mapReduce`	`sort`
`catch`	`get`	`next`	`text`
`clear`	`hadoop`	`not`	`text2int`
`contains`	`hasNext`	`or`	`text2real`
`containsAll`	`if`	`peek`	`throw`
`continue`	`in`	`print`	`tokenize`
`default`	`instanceof`	`real`	`try`
`else`	`int`	`real2int`	`void`
`elseif`	`int2real`	`real2text`	`while`
`emit`	`int2text`	`Reduce`	
	`isEmpty`	`remove`	

3.4.5 Constants

The word ***constant*** has two different meanings in Hog. It can refer to either a variable that is *fixed*, that is, once it is initialized cannot be changed, or can refer to an ***unnamed value***, such as "1.0". To declare a constant variable, use the following pattern,

`final` *type variableName* = *value*;

The following are a list of examples of unnamed values and their corresponding types:

`-1`, `0`, `1`, `2` (all of type `int`)

`-0.12`, `3.14159`, `2.7182`, `1.41421` (all of type `real`)

`true`, `false` (all of type `bool`)

3.4.6 Text Literals

A text literal consists of a sequence of zero of more contiguous characters enclosed in double quotes, such as `"hello"`. A text literal can also contain escape characters such as `"\n"` for the new line character or `"\t"` for the tab character. A text literal has many of the same built-in functions as the String class in Java. String literals are constant and their values cannot be changed after they are created. String literals can be concatenated with adjacent text literals by use of the `+` operator and are then converted into a single `text` variable. Hog implements concatenation by use of the Java `StringBuilder` (or `StringBuffer`) class and its append method. All text literals in Hog programs are implemented as instances of the `text` class, and then are mapped directly to the equivalent `String` class in Java.[1]

3.4.7 Variable Scope

Hog implements what is generally referred to as lexical scoping or block scope. An identifier is valid within its enclosing block. The identifier is also valid for any block nested within its enclosing block.

3.4.8 Argument Passing

Since Hog is compiled into Java, it passes arguments using call-by-value. However, similarly to Java, it is possible to imitate call-by-reference behavior.

3.4.9 Evaluation Order

Hog uses applicative order (eager) evaluation, similarly to Java.

3.5 Types

3.5.1 Basic Types

The basic types of Hog include `int` (integer numbers in base 10, 64 bytes in size), `real` (floating point numbers, 64 bytes in size), `bool` (boolean values, `true` or `false`) and `text` (Strings, variable in size). Unlike some languages, Hog includes no basic character type. Instead, a programmer makes use of `text`s of size 1.

Implementation details: Hogs primitive types are not so primitive. They are in fact wrappers around Hadoop classes. For instance, Hogs `int` type is

[1] Technically, `text` objects are implemented as instances of Hadoop's `Text` class, which is closely related to the Java `String` class.

a wrapper around Hadoop's `IntWritableclass`. The following lists for every primitive type in Hog the corresponding Hadoop class that the type is built on top of:

Hog Type	Enclosed Hadoop Class
`int`	`IntWritable`
`real`	`DoubleWritable`
`bool`	`BooleanWrtiable`
`text`	`Text`

3.5.2 Derived Types (Collections)

There are two derived types that can be created by the programmer: `list<T>`and `set<T>`. Future versions of Hog are expected to implement other derived types, including dictionaries/hash maps, user-defined iterators, and multisets. The `list<T>` type is an ordered collection of objects of the same type. The `set<T>` is an unordered collection of unique objects of the same type. Hog supports arbitrarily nested derived types, so it is possible, for example, to have a `list`of `list`s of `list`s of `int`s.

A special derived type is `iter<T>`, which is Hog's iterator object. An `iter` object is associated with a list, and allows one traversal of the elements in the list; this is used by Hog in the `@Reduce` section of a Hog program.

3.5.3 Type Conversions

In order to cast a variable to be of a different type, use the following notation:

primitiveType`(2`*otherPrimitiveType*`)`*variableName*

Hog supports casting between the primitive types `int`, `real`, and `text`, via the built-in functions `int2real`, `int2text`, `real2int`, `real2text`, `text2int`, and `text2real`. If casting a text to an int or real results in an invalid number (e.g. `text2int("1a4")`), a run-time exception will be thrown.

3.6 Expressions

3.6.1 Operators

Arithmetic Operators

Hog implements all of the standard arithmetic operators. All arithmetic operators are only defined for use between variables of numeric type (`int`, `real`) with the exception that the `+` operator is also defined for use between two `text` variables. In such instances, `+` is defined as concatenation. Thus, in the following,

```
text face = "face";
text book = "book";
text facebook = face + book;
```

After execution, the variable `facebook` will have the value "facebook". No other arithmetic operators are defined for use with `text` variables, and `+` is only valid if both variables are of type `text` . Otherwise, the program will result in a compile-time `TypeMismatchException`.

When an arithmetic operator is used between two numeric variables of different type, as in,

```
int a = 1;
real b = 2.0;
```

the non-`real` variable would first need to be cast into a `real` before operating on them, so that both operands have the same type. So thus

```
print(a + b);
```

would throw an error, while

```
print(int2real(a) + b);
```

would print 3.0.

If one of the operands happens to have a `null` value (for instance, if a variable is ***uninitialized***), then the resulting operation will cause a run-time `NullValueException`, and the program will crash.

Operator	Arity	Associativity	Precedence Level	Behavior
`+`	binary	left	0	addition
`-`	binary	left	0	minus
`*`	binary	left	1	multiplication
`/`	binary	left	1	division
`%`	binary	left	2	mod†
`++`	unary	left	3	increment
`--`	unary	left	3	decrement
`-`	unary	right	3	negate

†Follows Java's behavior: a modulus of a negative number is a negative number.

Logical Operators

The following are the logical operators implemented in Hog. Note that these operators only work with two operands of type `bool`. Attempting to use a logical operator with an object of any other type results in a compile-time exception (see §3.13.1).

Operator	Arity	Associativity	Precedence Level	Behavior
`or`	binary	left	0	logical or
`and`	binary	left	1	logical and
`not`	unary	right	2	negation

Comparators

The following are the comparators implemented in Hog (all are binary operations).

Operator	Associativity	Precedence Level	Behavior
`<`	none	0	less than
`<=`	none	0	less than or equal to
`>`	none	0	greater than
`>=`	none	0	greater than or equal to
`==`	none	0	equal
`!=`	none	0	not equal

Note: All comparators do not work with non-numeric or non-boolean types. Comparisons require that the two operands be either both numeric or both boolean, and a numeric value cannot be compared to a boolean value. If the two operands are numeric but of different types, one of them must be cast so that they are of the same type. The only valid comparators that can be used with boolean expressions are `==` and `!=`. The use of a comparison operator in Hog between any two derived types will result in a run-time error.

Assignment

There is one assignment operator, '`=`'. Expressions involving the assignment operator have the following form:

$$identifier_1 \texttt{ = } expression \mid identifier_2$$

At compile time, the compiler checks that both the result of the *expression* (or $identifier_2$) and $identifier_1$ have the same type. If not, a compile-time `TypeMismatchException` will be thrown.

3.7 Declarations

A user is only allowed to use variables/functions after they have been declared. When declaring a variable, a user must include both a type and an identifier for that variable. Otherwise, an exception will be thrown at compile time.

3.7.1 Type Specifiers

Every variable, whether its type is primitive or derived, must be assigned a type upon declaration, for instance,

```
list<int> myList;
```

declares the variable `myList` to be a `list` of `int`s,

```
list<list<int>> myOtherList;
```

declares the variable `myOtherList` to be a `list` of `list`s of `int` s, and

```
text myText;
```

declares the variable `myText` to be of type `text`.

3.7.2 Declarations

Null Declarations

If a variable is declared but not initialized, the variable becomes a ***null reference***, which means it points to nothing and holds no data (internally, this means that an entry has been added to Hog's symbol table with that variable name).

Primitive-Type Variable Declarations

Variables of one of the primitive types, including `int`, `real`, `text`, or `bool`, are declared using the following patterns:

1. *type identifier* (uninitialized)
2. *type identifier = expression* (initialized)

When the first pattern is used, we say that the variable is ***uninitialized***, and has the value `null`. When the second pattern is used, we say that the variable is ***initialized***, and has the same value as the value of the result of the *expression*. The *expression* must return a value of the right type, or the compiler will throw a `TypeMismatchError`. The *expression* may contain an expression involving both other variables and unnamed raw primitives (e.g. 1 or 2), an expression involving only other variables or unnamed raw primitives, or a single variable, or a single unnamed raw primitive.

Derived-Type Variable Declarations

Derived-type variables are declared using the following pattern:

1. *type identifier;*

When the derived type is first declared, we say that the variable is ***uninitialized***, and has the value `null`. If a user attempts to use any type-specific operations that are not meaningful (for instance, `myList.size()` on an uninitialized variable, the program will throw a runtime exception (see §3.13 for a discussion of exceptions)). The example code below initializes a `list`of integers and adds one element to it.

```
list<int> myList;
myList.add(5);
```

Function Declarations

In order to declare a function, use the following notation:

type functionName `(` *parameterList* `)` {
    expressionList
}

3.8 Statements

3.8.1 Expression Statement

An ***expression statement*** is either an individual assignment or a function call. All consequences of a given expression take effect before the next expression is executed.

3.8.2 Compound Statement (Blocks)

Compound statements are defined by { and } and are used to group a sequence of statements, so that they are syntactically equivalent to a single statement.

3.8.3 Flow-Of-Control Statements

The following are the ***flow-of-control*** statements included in Hog:

`if (` *expression* `)` *statement*

`if (` *expression* `)` *statement* `else` *statement*

`if (` *expression* `)` *statement* `elseif (` *statement* `) ... else` *statement*

In the above statements, the ... signifies an unlimited number of `elseif` statements, since there is no limit on the number of `elseif` statements that can appear before the final `else` statement. In the second statement above, when the expression in the `if` statement evaluates to `false`, then the `else` statement will execute. In the third statement above with `if`, `elseif` and `else` statements, the statement will be executed that follows the first expression evaluating to `true`. If none of these expressions evaluate to `true`, then the `else` statement is executed.

To increase the expressive power of Hog, flow-of-control statements can also be nested within each other.

3.8.4 Iteration Statements

Iteration statements signify looping and can appear in one of the two following forms:

`while` (*expression*) *statement*

`for` ($expression_1$; $expression_2$; $expression_3$;) *statement*

`foreach` *expression* `in` *iterable-object statement*

In the `while` pattern, the associated *statements* will be executed repeatedly until the *expression* evaluates to `false`. The *expression* is evaluated before every iteration. Please note that in a slight syntactical departure from Java, Hog requires a semicolon after the third expression (the increment step) in the `for`loop construct. Thus, an example of correct Hog syntax would be

```
 for (int i = 0; i <
10; i++;){...}
```

In the `for` pattern, $expression_1$ is the initialization step, $expression_2$ is the test or condition and $expression_3$ is the increment step. At each step through the for loop, $expression_2$ is evaluated. When $expression_2$ evaluates to false, iteration through the loop ends.

In the `foreach` pattern, the iteration starts at the first element in the *iterable-object statement* (a statement that evaluates to an object that supports the `iterator()` function). The *statement* executes during every iteration. The iteration ends when the *statement* has been executed for each item in the iterable object and there are no items left to iterate through.

Example of `while`

```
int i = 0;
while (i < 10) {
  print(i);
  i++;
}
```

Example of `for`

```
for (int i = 0; i < 10; i++;) {
  print(i)
}
```

Example of `foreach`

```
# we first initialize and populate the list as follows:
list<int> iList;
for (int i = 0; i < 10; i++;) {
   iList.add(i);
}
```

```
# This is an example of using foreach
# Note that the type of the iterable must be declared.

foreach int i in iList {
  print(i);
}
```

3.9 Built-in Functions

Hog includes both ***system-level*** and ***object-level*** built-in functions. Here ***built-in*** means functions provided by the language itself.

3.9.1 System-level Built-ins

Hog includes a number of systemlevel builtin functions that can be called from various sections of a Hog program. The functions are:

`void emit(key, value)`

This function can be called from the `@Map` and `@Reduce` sections in order to communicate the results of the map and reduce functions to the Hadoop platform. The types of the key/value pairs must match those defined as the output types in the header of each section.

`void mapReduce()`

This function can be called from the `@Main` section in order to initiate the mapreduce job, as definied in the `@Map` and `@Reduce` sections. Any Hog program that implements mapreduce will need to call this function in `@Main`.

`void print(toPrint)`

This function can be called from the `@Main` section in order to print to standard output. The argument must be a primitive type.

3.9.2 Object-level Built-ins

The derived type objects have several built-in functions that provide additional functionality. All of these functions are invoked using the following pattern:

identifier.functionName(parameterList)

Where *identifier* is the identifier for the object in question, *functionName* is the name of the function, and *parameterList* is a (possibly empty) list of parameters used to specify the behavior of the invocation.

Note: In what follows, if a function has return type T, it means that the return type of this function matches the parameterized type of this object (i.e. for an `iter<int>` object, these functions have return type `int`).

`iter`

`iter` is Hog's iteration object, and supports several built-in functions that are independent of the particular type of the `iter` object. The built-in functions are as follows:

> `bool hasNext()`
>
> This function returns `true` if the iterator object has a next object to return, and `false` otherwise.
>
> `T next()`
>
> This function returns the next object (if one exists) for the owning `iter` object. A call to `next()` differs from a call to `peek()` in that the function call advances the cursor of the iterator.
>
> `T peek()`
>
> This function returns the next object (if one exists) for the owning `iter` object. A call to `peek()` returns the object without advancing the iterator's cursor, thus multiple calls to `peek()` without any intermediate function calls will all return the same value.

`list`

> `void add(T itemToAdd)`
>
> Adds the object passed to the end of the list. The object must be of the same type as the list, or the operation will result in a **compile-time or run-time** exception.
>
> `void clear()`
>
> Removes all elements in this `list`.
>
> `T get(int index)`
>
> Returns the item from the list at the specified index.
>
> `void sort()`
>
> Function that sorts the items in the list in lexicographical ascending order.
>
> `int size()`
>
> Returns an int with the number of elements in the list.

`set`

`bool add(T element)`

Returns `true` if the element was successfully added to the `set`, `false` otherwise.

`void clear()`

Removes all elements from the `set` such that it is empty afterwards.

`bool contains(T element)`

Returns `true` if the `set` contains this element, `false` otherwise.

`bool containsAll(set<T> otherSet)`

Returns `true` if all elements in `otherSet` are found in this set.

`bool isEmpty()`

Returns `true` if there are no elements in this `set`, `false` otherwise.

`iter<T> iterator()`

Returns an iterator over the elements in this `set`.

`bool remove(T element)`

Returns `true` if the element was successfully removed from the `set`, `false` otherwise (i.e. the `list` didn't contain `element`).

`bool removeAll(set<T> otherSet)`

Returns `true` if all the elements in `otherSet` were successfully removed from this set.

`int size()`

Returns the number of elements in the set.

text

The following function can be called on a `text` object:

`int length()`

Returns the length (number of individual characters) of this `text`.

`text replace(text matchText, text replacementText)`

Returns a new `text` object with each sub-`text` that matches `matchText` replaced by `replacementText`. This function does **not** alter the original `text` object.

```
list<text> tokenize(text delimiter)
```

`tokenize()` can be called on a `text` object to tokenize it into a list of `text` objects based on the delimiter. The delimiter is not included in any of the `text` objects in the returned list.

3.10 System Configuration

The user must set configuration variables in the `hog.rb` build script to allow the Hog compiler to link the Hog program with the necessary jar files to run the MapReduce job. The user must also specify the job name within the Hog source file.

HADOOP_HOME absolute path of hadoop folder

HADOOP_VERSION hadoop version number

JAVA_HOME absolute path of java executable

JAVAC_HOME absolute path of javac executable

HOST where to job is rsynced to and run

LOCALMEM how much memory for java to use when running in local mode

REDUCERS the number of reduce tasks to run, set to zero for map only jobs

3.11 Compilation Structure

Currently, the Hog compiler is implemented as a translator into the Java programming language. The first phase of Hog compilation uses the JFlex as its lexical analyzer, which is designed to work with the Look-Ahead Left-to-Right (LALR) parser generator CUP. The lexical analyzer creates lexemes, which are logically meaningful sequences, and for each lexeme the lexical analyzer sends to the LALR parser a token of the form <`token-name, attribute-value`>. The second phase of Hog compilation uses Java CUP to create a syntax tree, which is a tree-like intermediate representation of the source program, which depicts the grammatical structure of the Hog source program.

In the last phase of compilation, the Hog semantic analyzer generates Java source code, which is then compiled into byte code by the Java compiler. Then with the Hadoop Java Archives (JARs) the bytecode is executed on the Java Virtual Machine (JVM). With the syntax tree and the information from the symbol table, the Hog compiler then checks the Hog source program to ensure semantic consistency with the language specification. The syntax tree is initially untyped, but after semantic analysis Hog types are added to the syntax tree.

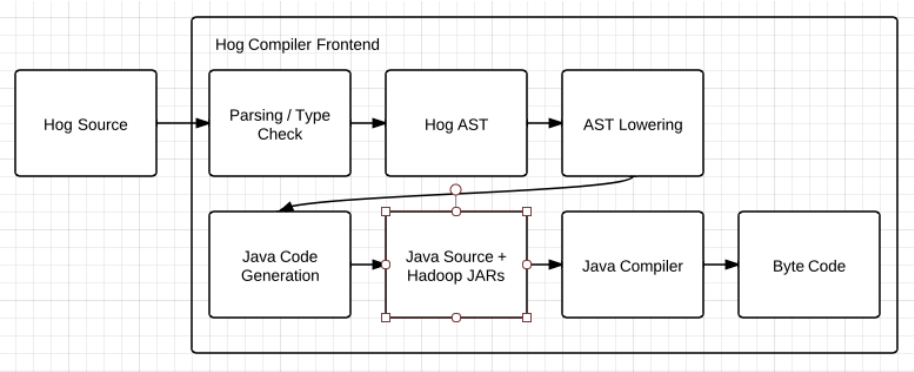


Figure 3.2: The overall structure of the Hog compiler.

Hog types are represented in two ways, either a translation of a Hog type into a new Java class, or by mapping Hog types to the equivalent Java types. Mapping Hog types directly to Java types improves performance because a JVM can handle primitive types much more efficiently than objects. Also, a JVM implements optimizations for well-known types, such as String, and thus Hog is built for optimal performance.

3.12 Linkage and I/O

3.12.1 Usage

To build and run a Hog source file there is an executable script `hog` that automates the compilation and linking steps for the user.

Usage: `hog [--hdfs|--local] job <job args>`

`--hdfs`: if job ends in '.hog' or '.java' and the file exists, link it against the hadoop JARFILE and then run it on HOST.

`--local`: run on local host.

3.12.2 Example

```
hog --local WordCountJob.hog --input someInputFile.txt --output ./someOutputFile.csv
```

This runs the `wordCount` job in *local* mode (i.e. not on a Hadoop cluster).

3.13 Exception Handling

Similar to some other programming languages (such as Java and C++), Hog uses an exception model in which an exception is thrown and can be caught by a catch block. Code should be surrounded by a try block and then any exceptions occurring within the try block will subsequently be caught by the catch block. Each try block should be associated with at least one catch block. However, there can be multiple catch blocks to handle specific types of exceptions. In addition, an optional finally block can be added. The finally block will execute in all circumstances, whether or not an exception is thrown. The structure of exception handling should be similar to this, although there can be multiple catch blocks and the finally block is optional:

```
try {
    expression;
} catch ( exception ) {
    expression;
} finally {
    expression;
}
```

The current version of the language does not support the programmer throwing exceptions, only catching them.

Because the proper behavior of a Hog program is dependent on resources outside of the language (i.e. the proper behavior of the users Hadoop software), there are more sources exceptions in Hog than most general purpose languages. These sources can be divided into two categories: ***compile-time exceptions*** and ***internal run-time exceptions***.

3.13.1 Compile-time Errors

The primary cause of most compile-time exceptions in Hog are semantic errors. Such errors are unrecoverable because it is impossible for the compiler to properly interpret the user program. Some compilers for other languages offer a limited amount of compile-time error correction. Because Hog programs are often designed to process gigabytes or terabytes of data at a time, the standard Hog compiler offers no compile-time error correction. The assumption is that a user would rather retool their program than risk the chance of discovering, only after hours of processing, that the compilers has incorrectly assumed what the user meant. The following are Hog compile-time exceptions:

`FunctionNotDefinedError`

Thrown when a program attempts to carry out an operations of the sort `variable.builtInFunction()` where `variable` is some variable and `builtInFunction` is a built-in function, and either `builtInFunction` cannot operate on variables of that type or `builtInFunction` is not defined as a built-in function.

`InvalidFunctionArgumentsError`

Thrown when a program calls a function with the wrong number or type of parameters. For example, if we define the function `max(int a, int b)` , this error will be thrown if the program contains a construct like `max(2,3,4)` or `max("hello", 3)` .

`TypeMismatchError`

Thrown when a program attempts to carry out an operation on a variable of the wrong type (like adding a `text` and an `int` together).

`UnreachableCodeError`

Thrown when code is included in a part of a program that will never be executed (e.g. code after a return statement that can never be reached).

3.13.2 Internal Run-time Exceptions

Internal runtime exceptions include such problems as I/O exceptions (i.e. a specified file is not found on either the users local file system or the associated Hadoop file system), type mismatch exceptions (i.e. a program attempts to place two elements of different types into the same list) and parsing exceptions. The following are Hog internal run-time exceptions:

`FileNotFoundException`

Thrown when the the Hog program attempts to open a non-existent file.

`FileLoadException`

Thrown when an error occurs while Hog is attempting to read a file (e.g. the file is deleted while reading).

`ArrayOutOfBoundsException`

Thrown when a program tries to access a non-valid index of a `list`.

`IncorrectArgumentException`

Thrown when a derived-type object is instantiated with invalid parameters, or a function is called with invalid parameters.

`TypeMismatchException`

Thrown when a program attempts to carry out an operation on a variable of the wrong type (like adding a `text` and an `int` together).

`NullReferenceException`

Thrown whenever the value of a variable cannot be `null` (e.g. in `myList.get(i)`, if `i` is `null`, the operation with throw a `NullPointerException`).

`ArithmeticException`

Thrown whenever an arithmetic operation is attempted on non-numeric operands.

3.14 Grammar

Note: The presented grammar has one minor ambiguity relating to the ***dangling-else*** problem. If the grammar is run through the parser generator `yacc`, `yacc` will identify 7 shift/reduce parsing-action conflicts. However, the ambiguity is handled by the default behavior of `yacc`, which preferences shift to reduce, associating `else` and `elseif` clauses with the closest `if` clause.

```
terminal DECR, INCR, RETURN, CONTINUE;
terminal TIMES, DIVIDE, MOD;
terminal LESS, GRTR, LESS_EQL, GRTR_EQL, DBL_EQLS, NOT_EQLS, ASSIGN;
terminal TEXT, BOOL, INT, REAL, VOID;
terminal MINUS, UMINUS, PLUS;
terminal ARROW, DOT;
terminal String TEXT_LITERAL;
terminal String ID;
terminal String INT_CONST;
terminal String REAL_CONST;
terminal String BOOL_CONST;
terminal String CASE;
terminal BREAK, DEFAULT;
terminal AND, OR, NOT;
terminal WHILE, FOR, FOREACH, IN, IF, ELSE, ELSEIF, SWITCH;
terminal FUNCTIONS, MAIN, MAP, REDUCE;
terminal L_BRACE, R_BRACE, L_BRKT, R_BRKT, L_PAREN, R_PAREN, SEMICOL, COL, COMMA;
terminal LIST, ITER, SET;
terminal TRY, CATCH, FINALLY;
terminal ExceptionTypeNode EXCEPTION;

nonterminal GuardingStatementNode GuardingStatement;
nonterminal CatchesNode Catches;
nonterminal IdNode CatchHeader;
nonterminal StatementListNode Finally;
nonterminal StatementListNode Block;
nonterminal StatementListNode ExpressionStatements;
nonterminal ExpressionNode ForExpr;
nonterminal StatementListNode ForInit;
nonterminal StatementListNode ForIncr;
nonterminal DerivedTypeNode DictType;

nonterminal ProgramNode Program;
nonterminal SectionNode Functions;
nonterminal SectionNode Main;
nonterminal SectionNode Map;
nonterminal SectionNode Reduce;
nonterminal SectionTypeNode SectionType;
```

```
nonterminal StatementNode Statement;
nonterminal ExpressionNode ExpressionStatement;
nonterminal StatementNode FunctionList;
nonterminal StatementNode IterationStatement;
nonterminal StatementNode LabeledStatement;
nonterminal SelectionStatementNode SelectionStatement;
nonterminal StatementNode DeclarationStatement;
nonterminal StatementListNode StatementList;
nonterminal ElseIfStatementNode ElseIfStatement;
nonterminal ElseStatementNode ElseStatement;
nonterminal JumpStatementNode JumpStatement;
nonterminal ExpressionNode EqualityExpression;
nonterminal ExpressionNode LogicalExpression;
nonterminal ExpressionNode LogicalTerm;
nonterminal ExpressionNode RelationalExpression;
nonterminal ExpressionNode Expression;
nonterminal ExpressionNode AdditiveExpression;
nonterminal ExpressionNode MultiplicativeExpression;
nonterminal ExpressionNode CastExpression;
nonterminal ExpressionNode UnaryExpression;
nonterminal ExpressionNode PostfixExpression;
nonterminal ExpressionNode PrimaryExpression;
nonterminal ExpressionNode Constant;
nonterminal ExpressionNode ArgumentExpressionList;
nonterminal FunctionNode Function;
nonterminal ParametersNode ParameterList;
nonterminal TypeNode Type;
nonterminal UnOpNode.OpType UnaryOperator;
nonterminal Types.Derived DerivedType;

precedence left MINUS, PLUS;
precedence right UMINUS;
precedence right ELSE;
precedence right ELSEIF;
precedence right L_PAREN;

start with Program;

Program::=
    Functions Map Reduce Main
  ;

Functions::=
    FUNCTIONS L_BRACE FunctionList R_BRACE
  |
    /* epsilon */
```

```
  ;

FunctionList::=
    Function
  |
    FunctionList Function
  ;

Function::=
    Type ID L_PAREN ParameterList R_PAREN L_BRACE StatementList R_BRACE
  ;

ParameterList::=
    ParameterList COMMA Type ID
  |
    Type ID
  |
    /* epsilon */
  ;

Map::=
    MAP SectionType L_BRACE StatementList R_BRACE
  ;

Reduce::=
    REDUCE SectionType L_BRACE StatementList R_BRACE
  ;

SectionType::=
    L_PAREN Type ID COMMA Type ID R_PAREN ARROW L_PAREN Type COMMA Type R_PAREN
  ;

Main::=
    MAIN L_BRACE StatementList R_BRACE
  ;

StatementList::=
    Statement
  |
    StatementList Statement
  ;

Statement::=
    ExpressionStatement
  |
    SelectionStatement
```

```
  |
    IterationStatement
  |
    LabeledStatement
  |
    JumpStatement
  |
    DeclarationStatement
  |
    GuardingStatement
  |
    Block
  ;

GuardingStatement::=
     TRY Block Finally
   |
     TRY Block Catches
   |
     TRY Block Catches Finally
   ;

Block::=
    L_BRACE StatementList R_BRACE
  |
    L_BRACE R_BRACE
  ;

Finally::=
    FINALLY Block
  ;

Catches::=
     CatchHeader Block
   |
     Catches CatchHeader Block
   ;

CatchHeader::=
     CATCH L_PAREN EXCEPTION ID R_PAREN
   ;

DeclarationStatement::=
    Type ID
  |
    Type ID ASSIGN Expression
```

```
  ;

JumpStatement::=
    CONTINUE
  |
    BREAK
  |
    RETURN ExpressionStatement
  ;

ExpressionStatement::=
    SEMICOL
  |
    Expression SEMICOL
  ;

Expression::=
    LogicalExpression
  |
    UnaryExpression ASSIGN Expression
  ;

LogicalExpression::=
    LogicalExpression OR LogicalTerm
  |
    LogicalTerm
  ;

LogicalTerm::=
    LogicalTerm AND EqualityExpression
  |
    EqualityExpression
  ;

EqualityExpression::=
    RelationalExpression
  |
    EqualityExpression DBL_EQLS RelationalExpression
  |
    EqualityExpression NOT_EQLS RelationalExpression
  ;

RelationalExpression::=
    AdditiveExpression
  |
    RelationalExpression LESS AdditiveExpression
```

```
  |
    RelationalExpression GRTR AdditiveExpression
  |
    RelationalExpression LESS_EQL AdditiveExpression
  |
    RelationalExpression GRTR_EQL AdditiveExpression
  ;

AdditiveExpression::=
    MultiplicativeExpression
  |
    AdditiveExpression PLUS MultiplicativeExpression
  |
    AdditiveExpression MINUS MultiplicativeExpression
  ;

MultiplicativeExpression::=
    CastExpression
  |
    MultiplicativeExpression TIMES CastExpression
  |
    MultiplicativeExpression DIVIDE CastExpression
  |
    MultiplicativeExpression MOD CastExpression
  ;

CastExpression::=
    UnaryExpression
  |
    L_PAREN Type R_PAREN CastExpression
  ;

UnaryExpression::=
    UnaryOperator CastExpression
  |
    PostfixExpression
  ;

UnaryOperator::=
    MINUS
    %prec UMINUS
  |
    NOT
  ;

PostfixExpression::=
```

```
    PrimaryExpression
  |
    ID DOT ID
  |
    ID DOT ID L_PAREN ArgumentExpressionList R_PAREN
  |
    ID L_PAREN ArgumentExpressionList R_PAREN
  |
    PostfixExpression INCR
  |
    PostfixExpression DECR
  ;

ArgumentExpressionList::=
    Expression
  |
    ArgumentExpressionList COMMA Expression
  |
    /* epsilon */
  ;

PrimaryExpression::=
    ID
  |
    Constant
  |
    L_PAREN Expression R_PAREN
  ;

Constant::=
    INT_CONST
  |
    REAL_CONST
  |
    BOOL_CONST
  |
    TEXT_LITERAL
  ;

SelectionStatement::=
    IF Expression Block ElseIfStatement ElseStatement
  |
    SWITCH Expression L_BRACE StatementList R_BRACE
  ;

ElseIfStatement::=
```

```
    ELSEIF Expression Block ElseIfStatement
  |
    /* epsilon */
  ;

ElseStatement::=
    ELSE Block
  |
    /* epsilon */
  ;

IterationStatement::=
    WHILE L_PAREN Expression R_PAREN Block
  |
    FOR L_PAREN ForInit ForExpr ForIncr R_PAREN Block
  |
    FOR L_PAREN ForInit ForExpr R_PAREN Block
  |
    FOREACH Type ID IN Expression Block
  ;

ForInit::=
    ExpressionStatements
  |
    DeclarationStatement SEMICOL
  ;

ForExpr::=
    ExpressionStatement
  ;

ForIncr::=
    ExpressionStatements
  ;

ExpressionStatements::=
    ExpressionStatement
  |
    ExpressionStatements COMMA ExpressionStatement
  ;

LabeledStatement::=
    CASE LogicalExpression COL Statement
  |
    DEFAULT COL Statement
  ;
```

```
Type::=
    VOID
  |
    TEXT
  |
    BOOL
  |
    INT
  |
    REAL
  |
    DerivedType:d LESS Type:t GRTR
  ;

DerivedType::=
    LIST
  |
    ITER
  |
    SET
  ;
```

Chapter 4

Project Plan

Written by Samuel Messing (sbm2158).

4.1 Development Process

The scope of the Hog programming language was ambitious from the start. Our stated goal was to create a general-purpose scripting language which made carrying out distributed computation simple and intuitive. As such, from the beginning we were interested in ways to make the implementation of the language as simple as possible. The following goals were identified early on:

- make the build system as simple as possible,
- make the logic of our individual modules as similar as possible,
- document everything,
- use a distributed version control system,
- write verbose and informative log statements.

Focusing on these goals throughout the development enabled use to work concurrently on different aspects of the compiler and maintain a codebase that was both readable and easy to understand.

4.1.1 Simplicity of Build System

As project manager, I worked early on with both the System Architect (Ben) and the System Integrator (Kurry) to come up with a build system that was simple and easily extensible. After trying a few different options, we decided on Ant, a build system similar to Make, specialized for the Java programming language. Another advantage of going with Ant is that both JFlex and Cup, the frameworks we used to construct the lexer and parser, respectively, have native

Ant support. Identifying and implementing our build system early on enabled us to move quickly and write code that we were sure worked across all of our machines.

4.1.2 Similarity of Modules

Throughout the project, I worked very closely with the System Architect (Ben) to develop and build common data structures that could be used across all of our code. The abstract syntax tree was made in a generic enough way so that all of our different tree walks could use the exact same tree class, without having to support and debug different implementations of the same interface or abstract parent class.

Personally, I also developed our Types class, which was a static class that contained several convenience methods for handling types across the entirety of the compiler. These methods include type checking, type conversion and as well as additional functionality required for internal functionality. I set out to write the class as early as possible so that both elements of the frontend and the backend could make use of it. Simplifying and unifying how different modules handled the same information enabled everyone on the team to read each other's code qand quickly understand how it functioned.

4.1.3 Document Everything

One of the most undersold parts of Java is it's well thought out documentation schema (JavaDocs). Early on I realized that in order for us to be able to work semi-independently on different modules we would need to have a robust set of documentation. By using JavaDoc instead of regular comments, we were able to generate HTML documentation, which more clearly provides an overview of the entire architecture of our compiler, and allowed everyone on our team to work quickly and respond to updated classes appropriately.

One of the largest challeneges in this project was developing a set of node classes for our abstract syntax trees that captured the right granularity of information, without beeing too complex that handling corner cases became intractable. Our System Architect (Ben) found a great tool that generated UML diagrams for our class hierarchies, which in concert with our JavaDocs helped to make development as simple and efficient as possible.

4.1.4 Distributed Version Control

As soon as our team was formed I created a git repository on Github.com[1] for use by the team. One of the first things we discussed as a team was what workflow pattern we wanted to use throughout the course of the project. Very quickly we decided on a continuous-build pattern, where the main branch of our git repository (master) was reserved for compiling, tested, and finalized code.

[1] `http://www.github.com/smessing/Hog`

Any classes that were currently in development existed in separate branches, and were only merged into master after sufficient amount of testing. Each programmer maintained their own branch for development. If two or more programmers were working on the same class, a new, shared branch was created. By being conservative about what code was merged into the master branch, we were able to work independently, without fear that someone else's work would be interrupted by leaving our individual code in an unfinished state.

4.1.5 Verbose Logging

Another advantage of programming in Java is the robust and sophisticated logging libraries available to the programmer. Around the same time that the build system was developed, the System Architect (Ben) investigated several different logging libraries and wrote a tutorial for the rest of us on how to use it. The logging library supported several levels of log statements, FINEST, FINER, FINE, INFO, WARNING and SEVERE (from most verbose to least). We decided that FINEST and FINER were to be used strictly for debugging, while FINE was to be used to document normal behavior, at a level of detail that was concise enough for all developers to look at, but still too verbose for the user. INFO, WARNING and SEVERE were reserved for statements that the user would see. By identifying and keeping to these log levels early on, we were able to quickly identify bugs and inefficient or errant behavior.

4.2 Roles and Responsibilities

- Ben, System Architect

  Ben's major responsibilities included developing the fundamental data structures used by the compiler, working out the different elements of the compiler and how they interrelate, and developing the symbol table.

- Jason, Testing/Validation

  Jason's major responsibilities included testing all of the elements of our compiler, and working on the aspects of the compiler related to type checking, and developing the symbol table.

- Kurry, System Integrator

  Kurry's major responsibilties included developing a clean interface between Hog and Hadoop and working on the Hog wrapper program that builds, compiles and runs Hog source programs.

- Paul, Language Guru

  Paul's major responsibility was determining the syntax and semantics of our language, and developing the semantic analyzer.

- Sam, Project Manager

  As project manager, my major responsibilities included setting project deadlines, assigning work, and making sure that we met our goals. I was also responsible for developing the classes to translate Hog programs into Java programs.

4.3 Hog's Developer Style Sheet

We made use of the standard Java style guide, including such conventions as camel case, verbs for functions and method names, and hierachical object classes. For formatting, we used Eclipse's auto-format feature to keep our code looking as consistent as possible.

4.4 Project Timeline

January

Developed several potential ideas for languages. Met with Aho and decided on implementing Hog, a MapReduce language.

February

Worked on the White Paper for our language, developed both the goals of our language and the overall "feel" (simple, minimal boilerplate code, easy-to-read syntax). Started to sketch out overall compiler architecture, and decided on frameworks (JFlex for the lexer, CUP for parser, Hadoop framework for executing distributed computation, and Java as target language) and development environments (Eclipse, Git, Github, LaTeX for documentation).

March

Wrote the language reference manual and tutorial for our language. Developed the build system (Ant for compiling compiler code, Make for running the compiler on Hog source programs), implemented and tested the parser and lexer, and developed the fundamental data structures (abstract syntax tree, node classes).

April

Implemented tree walking algorithms to populate the symbol table, perform type checking, perform semantic analysis and generate Java source code. Wrote tests for the walkers.

May

Refactored code and worked on documentation. Developed more tests and worked on fixing bugs.

4.5 Project Log

January

Week of January 22nd

* Met to discuss language ideas.

Week of January 29th

* Decided on Hog, and Java as implementation language.

February

Week of Feburary 5th

* Decided on Hadoop as the framework for executing distributed computation.
* Decided on JFlex framework for implementing the lexer.
* Decided on CUP framework for implementing the parser.

Week of February 12th

* Discussed and figured out development environment (Java, Ant, Eclipse, Git).
* Started working on white paper.

Week of February 19th

* Started git repository.
* Finished white paper.

Week of February 26th

* Began the language reference manual (LRM).

March

Week of March 4th

* Started Eclipse project.
* Worked on LRM and tutorial.

Week of March 11th

* Began developing the Hog grammar.
* Worked on LRM and tutorial.
* Started working on wrapper program functionality (program that runs the Hog compiler to compile source programs).

Week of March 18th

* Finished the Hog grammar.

* Finished the tutorial and LRM.
* Began developing the lexer.

Week of March 25th

* Took the week off to study for the midterm.

April

Week of April 1st

* Worked on the lexer.
* Started developing the abstract syntax tree and node classes.
* Started developing the parser.
* Implemented developer build system.

Week of April 8th

* Developed ConsoleLexer class for development and testing of lexer.
* Developed lexer JUnit tests.
* Finished abstract syntax tree, including iterators for post- and pre-order traversals.
* Developed mock classes for testing.

Week of April 15th

* Further development/refinement of node classes.
* Developed more semantic actions for parser, mainly to construct node classes.
* Parsed our first program!

Week of April 22nd

* Developed/implemented basic type functionality.
* Further refinement of the grammar.
* Implemented logging details.
* Refinement of node classes and ASTs.
* Started tree walking algorithms (identified the visitor pattern as our common design pattern for tree walks).
* Begain developing symbol table class.

Week of April 29th

* Finished implementation of symbol table class.
* Finished type checking / symbol table population walks.
* Implemented java source generator.
* Implemented tests for walkers and parser.
* Finished compiler.

Chapter 5

Language Evolution

- Describe how the language evolved during the implementation and what steps were used to try to maintain the good attributes of the original language proposal.
- Describe the compiler tools used to create the compiler components.
- Describe what unusual libraries are used in the compiler.
- Describe what steps were taken to keep the LRM and the compiler consistent.

The initial intent was to make Hog stylistically and aesthetically resemble Python. In our first discussions about the language, we had envisioned statements being separated by line breaks and dynamic typing.

Chapter 6

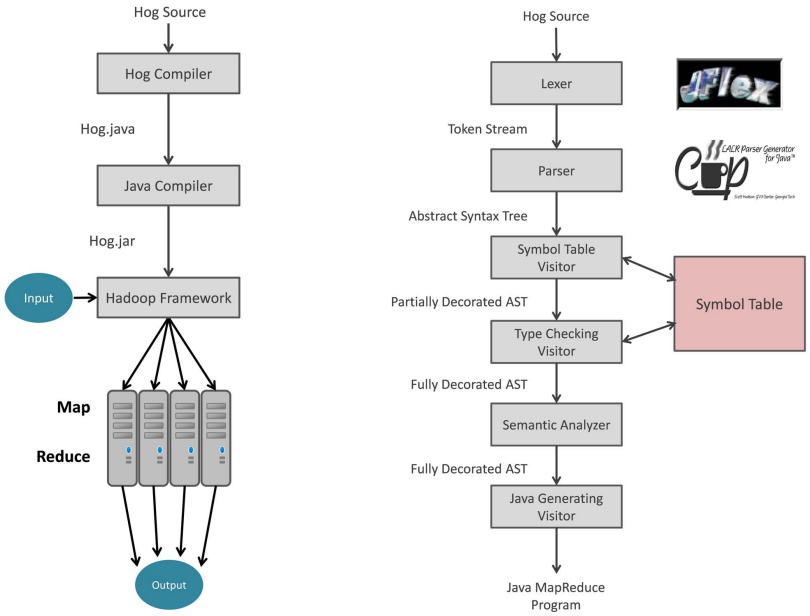
Translator Architecture

6.1 Architecture

At the highest level, the Hog architecture has three main phases (Figure 6.1a). It begins with the Hog compilation phase, which puts a Hog source program through the Hog compiler, and translates it into an equivalent valid Java Hadoop source program. This Java program is then compiled into a jar file by the Java compiler. Finally, the jar file is sent to a valid Hadoop instance (the cluster), along with the input to the program, to perform the a distributed computation and output the results.

The Hog compilation phase is further broken down into several key stages (Figure 6.1b). We begin by using the lexer JFlex to provide our parser Cup with a stream of tokens. Cup parses these tokens to output an abstract syntax tree. The Symbol Table Visitor uses this abstract syntax tree to populate the symbol table with any functions and variables defined in the program. It also ensures that all variables are declared before they are used and partially decorates the tree with some type information. The Type Checking Visitor checks for any type conflicts, and fully decorates the tree with type information for every expression. The Semantic Analyzer uses the decorated AST to check for semantic errors, such as dead code blocks, or non-void functions that do contain return statements. Finally, the Java Generating Visitor walks the AST, and generates the Java Hadoop source program.

All communication between these modules occurred through the Symbol Table and the Abstract Syntax Tree. Therefore, the interfaces used to pass information were defined by the Node classes making up the tree, and the SymbolTable and its contained Symbol classes.



(a) Hog Platform

(b) Hog Compiler

Figure 6.1: Hog Architecture

6.2 Module Authors

- Lexer
  - Samuel Messing
- Parser
  - Samuel Messing
  - Benjamin Rapaport
  - Paul Tylkin
- Abstract Syntax Tree
  - Samuel Messing
  - Benjamin Rapaport
- Symbol Table
  - Jason Halpern
  - Benjamin Rapaport
  - Kurry Tran

- Symbol Table Visitor
  - Benjamin Rapaport
  - Jason Halpern
- Type Checking Visitor
  - Benjamin Rapaport
  - Jason Halpern
  - Samuel Messing
- Semantic Analyzer
  - Paul Tylkin
- Java Generating Visitor
  - Paul Tylkin

Chapter 7

Development and Run-Time Environment

To be written by Kurry. @Kurry wrote this section.

- Describe the software development environment used to create the compiler. The Hog team used the Eclipse integrated development environment (IDE) that consisted of a source code editor, build automation tool, debugger, and a JUnit automated testing suite. The source code editor provided syntax highlighting in Java, as well as automatically formatted source code, and automatically generated HTML Java documentation, which made language reference very easy. The build automation tool that was used was Apache Ant. Apache Ant is a Java library and command-line tool that has built-in tasks allowing a number of routine tasks such as compiling, assembling, testing, and running Java applications, to be conifigured by an XML file that is standardized across the team. With the build standardized across the team, we were able to track what parts of the source code failed after merging, which allowed for quick debugging and resolutation. The Java Development Tools (JDT) project in Eclipse featured a built-in Java debugger that provided us the ability to perform step execution, to set breakpoints and values, and to inspect variables and values, and allowed the ability to suspend and resume threads. The debugger allowed us to track and eliminate the bugs in our source code early on, which made for a smoother development process. The JUnit testing framework is the standard unit testing Application Programming Interface (API) for Java development. Our Ant build configuration was integrated with JUnit to allow executing out test suites as part of the build process, and printed the results to the console, which prompted us if there were any test failures.
- Show the makefile used to create and test the compiler during development.

The Hog compiler contains both a makefile and a shell script for compilation. The shell script provides a simple interface for working with the makefile. If the user does not have a working Hadoop configuration on there computer, they can run the command "make compile" which will compile the Hog source program, which will create a Hog.java file, which will then be compiled with the Java compiler with the appropriate Hadoop libraries being packaged with the job jar which is generated at the end compilation.

```
######################################################################
######################################################################
## Makefile for Hog Compilation ##
## Programming Languages and Translator ##
## Version 1.0 ##
##  Kurry Tran ##
## 04 May 2012 ##
## ##
## ##
######################################################################
######################################################################

# Set Up Compiler Directories
# TODO Fill In Directories
JFLAGS = -g -classpath
HOGCOMPILER=compiler/Hog.jar

# Input Hog Source Name
INPUT_HOG_SOURCE=WordCount.hog
SOURCENAME=WordCount

# DO NOT CHANGE
INPUT_JAVA_SOURCE=Hog.java

# Java Compiler
JAVAC =javac
# Hadoop Home
HADOOP_HOME=/Users/ktran/hadoop-1.0.1/
HADOOP_VERSION=1.0.1
JOBNAME=Hog
CLASSPATH = -classpath $(HADOOP_HOME)hadoop-core-$(HADOOP_VERSION).jar
JAR=jar
JARFLAGS=-cf
HADOOP=$(HADOOP_HOME)bin/hadoop
FS=dfs
PUTINTODFS=-put
```

```
INPUTDATA=input/big.txt input/input.txt input/words.txt

# HDFS Input/Output Directories
DFSINPUTDIRECTORY=/users/ktran/input/
DFSOUTPUTDIRECTORY=/users/ktran/output

CLASSFILES=*.class
CAT=-cat
RMR=-rmr
JOBJAR=hog.jar
JOBNAME=Hog
USERNAME=ktran
MKDIR=-mkdir

default: buildandrun

makedir:
$(HADOOP) $(FS) $(MKDIR) $(DFSINPUTDIRECTORY)

buildandrun:
$(JAVAC) $(CLASSPATH) $(INPUT_JAVA_SOURCE)
$(JAR) $(JARFLAGS) $(JOBJAR) $(CLASSFILES)
$(HADOOP) $(FS) $(PUTINTODFS) $(INPUTDATA) $(DFSINPUTDIRECTORY)
$(HADOOP) $(JAR) $(JOBJAR) $(JOBNAME) $(DFSINPUTDIRECTORY) $(DFSOUTPUTDIRECTORY)
$(HADOOP) $(FS) $(CAT) $(DFSOUTPUTDIRECTORY)/part-00000 > $(SOURCENAME).txt

compile:
java -jar $(HOGCOMPILER) --local $(INPUT_HOG_SOURCE)
$(JAVAC) $(CLASSPATH) $(INPUT_JAVA_SOURCE)

clean:
$(RM) *~  *#
$(HADOOP) $(FS) $(RMR) $(DFSINPUTDIRECTORY)
$(HADOOP) $(FS) $(RMR) $(DFSOUTPUTDIRECTORY)
```

- Describe the run-time environment for the compiler. The Hog compiler runs on the Java Virtual Machine (JVM) which requires programs to be in a standardized portable binary format which are typically .class files. For distribution of large programs, multiple class files may be packaged together in a .jar file (short for Java archive), which is how the Hog compiler is transported, as a single java archive. The JVM executes the Hog compiler, Hog.jar, and emulates the JVM instruction set by interpretting it, and linking the appropriate libraries from Java and Hadoop, and running the parser and lexer that were generated on Java Cup, and prints

any errors to the console. Once the Hog source program is compiled, the resulting jar can be uploaded to the Amazon ElasticMapReduce cloud to be run.

Chapter 8

Test Plan

As the tester and validator for Team Hog, I set out to create a systematic, automated set of tests at each step in the process of building a compiler. In order to make sure that each part of the design worked according to our specification, I tried to include tests that touched as many aspects of the language as possible.

I considered each of the testing phases to be a two-step process. First, create a basic set of tests with the assumption that the compiler worked as expected. These tests would touch a variety of areas of the language. These were our black box tests because they were built without the need to know what was going on under the hood. Then, the second step of the process was to attempt to break the language in as many ways as possible. These tests required an intimate knowledge of the nuances of the language and were therefore our white box tests. I tried to incorporate as many boundary cases as possible into these tests. At each phase of the testing, we uncovered various bugs and unimplemented aspects of the language that we fixed on subsequent iterations. I will briefly touch upon each phase of testing and the challenges and outcomes faced throughout the process. All of these tests are in the test package in our source code. The tests were developed using Javas JUnit development framework.

Lexer Testing (`LexerTester.java`)

In order to test the lexical analysis of Hog programs, I created a large variety of short code snippets, passed them to the lexer and made sure that the correct tokens were being returned. For example, when the string "a++" was passed to the lexer, I created tests with `assertEquals()`to make sure that the first token returned was ID and the second token returned was INCR. I started with small tests that only touched two to five token streams and built towards strings that were thirty tokens long. This phase helped us discover certain tokens that were not being returned correctly and needed to be added/modified in the lexer, such as `TEXT`, `TEXT_LITERAL`, and `UMINUS`. A sample lexer test can be seen at the end of this section.

Parser Testing (`ParserTester.java`)

This was the most challenging aspect of the testing process. Due to the limitation of built-in parsing methods, it was difficult to create an automated set of tests for the parser that tested each part of the grammar. This phase relied more heavily on manual testing than I would have preferred. We were able to run a variety of programs through the parser and focused on breaking the parser and touching as many edge cases as possible. This allowed us to uncover the bugs and produce code that was not correctly parsed. We had to modify and expand the grammar from the results of this testing. The tests that we created for the parser were the motivation for creating such specific node subclasses that captured the different details associated with each production. In addition, information that we gathered in testing the parser also allowed us to create a clean design for the symbol table, which is constructed during parsing and the first walk of our AST.

Symbol Table Testing (`SymbolTableTester.java`)

I found when I reached this phase of creating tests that there were certain details of the node classes that I needed to gain a better understanding of in order to write tests. For this reason, I worked closely with Ben and Paul in designing and implementing the Symbol Table and worked with Ben on the Symbol Table Visitor and Type Checking Visitor. In order to test the construction of the Symbol Table, we created several sample programs, created the Symbol Table from these programs and analyzed the symbol table to make sure the information was being correctly captured. In addition, we also made sure reserved words and functions were in the reserved symbol table at the root of the Symbol Table structure. There were two key issues related to creating nested scopes that were uncovered during testing and an important issue related to adding function parameters and argument lists to the symbol table. This phase also focused on making sure the correct exceptions were being thrown i.e. VariableRedefinedException, VariableUndeclaredException, etc.

Abstract Syntax Tree Testing (`AbstractSyntaxTreeTester.java`)

In order to test the AST, we created an automated set of tests that was based on the pre and post order traversals of the AST. First, we created an AST during the set up phase of the testing and made sure that we included a variety of node structures on the tree. Then, we did both a preorder walk of the tree and a postorder walk of the tree and made sure the traversals were occurring in the correct order.

Type Checking Testing (`TypesTester.java` and `TypeCheckingTester.java`)

During this walk of the AST, we did type checking and decorated the tree with the correct types. I created many of the tests for this part of the design as

Ben and I implemented functionality in the type check walk. The first part of type checking testing was to make sure the functions that we wrote around type compatibility were operating correctly. For example, we had to make sure if we visit a BiOpNode with the plus operator that the operands are both text (concatenation) or numbers (addition). Once the tests proved that these functions were all valid, we moved to implementing tests on the walk of an actual AST to make sure type decorating was occurring according to our rules. This part of the testing uncovered the fact that our IdNodes were not being decorated at all during our initial walk, so we added the functionality to the TypeCheckingVisitor to handle this.

Code Generation Testing (`CodeGeneratingTester.java`)

The goal for the code generation tests was to determine whether or not our programs were being correctly mapped to Hadoop programs written in Java. These tests focused on using the code generating visitor walk of the tree to make sure that the structure, meaning and types of our Hog programs were being captured during the transformation to Java. Besides writing tests, this step of the testing involved actually running the Hadoop programs on our local machines and on Amazon Elastic MapReduce to see if the programs would run without errors and if the results in the output files were in line with what we expected.

Testing Hog Programs

In order to prepare us for testing, I set us up on Amazon Web Services to run our programs on Amazons Elastic MapReduce platform. We upload the jar of our compiled program and the input files to Amazons S3 storage platform, then we launch the Elastic MapReduce job on a small cluster with 2 instances. The output files are stored in S3 after the processing has successfully completed. The instructions for running a Hog program on AWS are detailed in the report.

For several aspects of our implementation, we focused on a pair approach to programming and to testing. Since Sam handled a lot of the implementation regarding lexical analysis and parsing, he also worked with me to create additional tests for these phases to make sure we captured everything. In addition, he also added type tests for some additional type functions that he wrote. Kurry created some tests for the node structure since he also worked on creating ASTs. In addition, since I wanted to really understand the node structure, symbol table and visitor pattern, I worked together with Ben and Paul in designing and implementing the symbol table, designing the visitor pattern for Hog and implementing the Type Checking walk. Working on these aspects of the project enabled me to write better tests since I had a deeper understanding of these classes. I also think that writing tests helped Kurry and Sam better understand the aspects of the project that they were working on.

One of the main challenges during testing was capturing the breadth and depth of the Hog language in all of the tests. Testing, in conjunction with devel-

opment, was an iterative process that required us to add and modify the testing suites as functionality was added to and removed from the language specification. As the tester and validator for this project, I believe I have developed the skills to more rigorously test software. More importantly, I learned a lot about the principles related to strong software design and software engineering during the entire process.

Sample Test from `LexerTester.java`

```
/**
* Tests for correct parsing of the postfix increment operator
*
* Specifically, ensures that Lexer produces a token stream of ID * INCR
*     for strings like "a++"
*
* @throws IOException

@Test
public void incrementSymbolTest() throws IOException {

String text = "a++";
StringReader stringReader = new StringReader(text);
Lexer lexer = new Lexer(stringReader);
List<Integer> tokenList = new ArrayList<Integer>();
Symbol token = lexer.next_token();

while (token.sym != sym.EOF) {
tokenList.add(token.sym);
token = lexer.next_token();
}

assertEquals(
"It should produce 2 tokens for the string '" + text + "'", 2, tokenList.size());
assertEquals("The first token should be a ID", sym.ID, tokenList.get(0)
.intValue());
assertEquals("The second token should be a INCR",
               sym.INCR, tokenList.get(1).intValue());
}
```

Chapter 9

Conclusions

9.1 Lessons Learned

The following lessons were learned by the group:

1. Spend more time assessing the viability of different tools.

   > We very quickly decided to use JFLex and CUP to generate our lexer and parser, respectively. While the two tools proved useful, the interface between the two was not trivial to figure out, and required a lot of development time. In addition, the documentation for the two frameworks is rather sparse, and at times we had to resort to trial-and-error to figure out exactly how to get the frameworks to do what we want (a great example of this was determining how to properly specify precedence). Only after it was too late did we learn about ANTLR, another framework for writing lexers and parsers in Java, which seems to be a much better framework all around. If we were starting this project over today, we would probably go with ANTLR over JFlex and CUP.

2. Design the AST with a focus on the output (i.e. the program you're generating), rather than a focus on the input (i.e. the grammar of your source language).

   > We initially designed our AST classes to mimic the parse tree generated by our grammar. While this was instructive for debugging the grammar, it was not very useful for the backend, as we ended up with a lot of redundancy in our tree. In order to make things more efficient, we needed to refactor our entire node structure at some point, which took a long time, and was a source of confusion as we moved forward.

3. Spend more time in general on design before implementation.

The above lesson applies more generally. Before writing a single character of code, we found it very important to spend a few hours (if not longer) sketching out designs, to make sure our interfaces between classes were as clean as possible. Sometimes we didn't spend enough time on this stage, and ultimately paid the price in increased development and debugging time.

4. Go out of your way to make things modular.

   At times we were unable to work contemporaenously because several developers needed to wait for another to finish something before they could begin development of their next assignment. By the end of the project, we found a good way to make things modular enough to work concurrently, but at the beginning we didn't emphasize this goal nearly enough, and it slowed down development.

5. Be skeptical about what you can accomplish.

   We were very excited to implement our own programming language, and as such, were over zealous in the scope of the structures we wanted to support. Instead of spending so much time thinking about fancy syntactic sugar, we would've benefited from spending more time looking skeptically at the scope of what would could accomplish. By the end, there were several elements of the language that we needed to pull support for (e.g. switch statements, declaring your own iterators, etc.), as the scale of our aspirations was simply too large for the development timeframe.

9.1.1 Jason's Lessons

One of the important lessons I learned as a tester was the importance of having rigorous, thorough tests in place at each part of the process. Having these tests in place not only helps the tester find bugs, but it is crucial in allowing the other developers on the team to run tests as they complete certain aspects of the design. In addition, having tests in place for each part of the compiler design made integration much easier since we already knew that the individual components were working properly. A challenge in developing the testing suites was to make sure that I was capturing all aspects of the design as we iterated through each step of the development process. Because we were modifying aspects of our design as we expanded, I also had to make sure that the tests were designed to capture the new changes. This really brought to light the importance of strong design in the software development process. In fact, I think we should have spent a little more time thinking about design early in the process.

Since we did a lot of our development in pairs, I gained a strong appreciation for pair programming during the project. I think our group made our greatest strides when components were being done in pairs, especially when it came to the more challenging aspects of the project, such as node designs and the different walks of the AST. This was the first time that I had significant experience with pair programming and I think it will be valuable to have had this experience.

One of the best parts about the implementing the language was that it connected a lot of the theory that we learned in class with real-life implementation. I have taken computer science classes in the past in which the theory gets lost because it remains disconnected from practice. I really appreciated how this class managed to bridge the gap between both theory and practice, because it created a holistic learning experience.

9.1.2 Sam's Lessons

Work in the same room as much as possible. If I've learned anything on this project it's that while working individually can sometimes feel more productive, the lack of communication during development leads to a lot of overhead. Developers will work with different conceptions of how interfaces are designed, and will sometimes end up writing libraries that other developers have written already, simply because they didn't know they were there. The end of the semester went much more smoothly once we realized that all development should be done together. Not only does it allow you to avoid duplication of effort, but when a particular developer is stuck, he or she can speak to his or her teammates and get their support.

9.1.3 Ben's Lessons

To be written by Ben.

9.1.4 Kurry's Lessons

I think the most important lesson learned was about how to make good judgments on what our language should support and should not support, and I think that came from a lot of bad judgements we made early on about what we were going to support in our language. Our language was by far much more sophisticated than the other languages made by the class, but we also had to cut some things out of our language just because we did not have the man power or time frame necessary to implement them.

From a project planning standpoint, I think that the team should have all worked together on building the abstract syntax tree and node classes, instead of making that work modular in the beginning. This is due to the fact that all parts of the team rely on using a working abstract syntax tree, and if the AST is not complete early on in the project, it make creating the other parts of the compiler very difficult.

9.1.5 Paul's Lessons

To be written by Paul.

9.2 Advice for Other Teams

Don't take this class.

9.3 Suggestions for Instructor

Things we'd like to see more of:

- More details
- More discussion of functional languages

Appendix A

Code Listing

- `back_end` package
    1. `CodeGenerationVisitor.java`

       An AST-walker that translates Hog programs into Java programs. Written by Samuel Messing.
    2. `ErrorCheckingVisitor.java`

       An AST-walker that performs semantic analysis. Written by Paul Tylkin.
    3. `SymbolTableVisitor.java`

       An AST-walker that populates symbol tables. Written by Benjamin Rapaport and Jason Halpern.
    4. `TypeCheckingVisitor.java`

       An AST-walker that performs basic type checking and type inference. Written by Jason Halpern and Benjamin Rapaport.
    5. `Visitor.java`

       The abstract class that all visitors inherit, specifies the behavior of the Visitor design pattern. Written by Jason Halpern.
- `front_end` package
    1. `ConsoleLexer.java`

       A console front-end to the Lexer class for dynamically testing the Lexer, and development. Not intended to be used/accessible by/to users. Written by Samuel Messing
    2. `Hog.java`

       Driver program for the compiler, handles generating the parser, paring the input file, and performing the tree walks. Written by Samuel Messing and Kurry Tran.

3. `Lexer.java`

   Auto-generated file.

4. `Parser.java`

   Auto-generated file.

5. `sym.java`

   Auto-generated file.

6. `Lexer.jflex`

   Lexer specification, written by Samuel Messing.

7. `Parser.cup`

   Parser specification, written by Samuel Messing, Benjamin Rapaport and Paul Tylkin.

- `test` package

  1. `AbstractSyntaxTreeTester.java`

     Tests for the functionality provided by the AbstractSyntaxTree class. Written by Samuel Messing and Jason Halpern.

  2. `CodeGeneratingTester.java`

     Tests functionality of the CodeGeneratingVisitor. Written by Jason Halpern.

  3. `LexerTester.java`

     Tests Lexer's performance on decomposing different inputs into the correct sequence of tokens. Written by Jason Halpern and Samuel Messing.

  4. `NodeTester.java`

     Tests for the functionality provided by the Node class. Written by Samuel Messing and Kurry Tran.

  5. `Parser.java`

     Tests basic functionality of Parser.java. Written by Samuel Messing.

  6. `SymbolTableTester.java`

     Tests functionality of the Symbol Table classes. Written by Jason Halpern.

  7. `TypeCheckingTester.java`

     Tests the TypeCheckingVisitor. Written by Jason Halpern.

  8. `TypesTester.java`

     A method to test the convenience class for the Types convenience class. Written by Samuel Messing and Jason Halpern.

- `util.ast` package

    1. `AbstractSyntaxTree.java`

        Class for specifying common behavior for ASTs. Written by Samuel Messing.

    2. `TreeTraversalBuilder.java`

        Constructs an Iterator¡@link Node¿ over a given AST for pre-order and post-order traversal. Written by Samuel Messing.

- `util.ast.node` package

    1. `ArgumentsNode.java`
    2. `BiOpNode.java`
    3. `CatchesNode.java`
    4. `ConstantNode.java`
    5. `DerivedTypeNode.java`
    6. `ElseIfStatementNode.java`
    7. `ElseStatementNode.java`
    8. `ExceptionTypeNode.java`
    9. `ExpressionNode.java`
    10. `FunctionNode.java`
    11. `GuardingStatmenetNode.java`
    12. `IdNode.java`
    13. `IfElseStatementNode.java`

14. IterationStatementNode.java

15. JumpStatementNode.java

16. MockExpressionNode.java

17. MockNode.java

18. Node.java

19. ParametersNode.java

20. PostfixExpressionNode.java

21. PrimaryExpressionNode.java

22. PrimitiveTypeNode.java

23. ProgramNode.java

24. RelationalExpressionNode.java

25. ReservedWordTypeNode.java

26. SectionNode.java

27. SectionTypeNode.java

28. SelectionStatementNode.java

29. StatementListNode.java

30. StatementNode.java

31. SwitchStatementNode.java

32. `TypeNode.java`

33. `UnOpNode.java`

- `util.error` package

    1. `FunctionNotDefinedError.java`

        Written by Benjamin Rapaport and Jason Halpern.

    2. `InvalidFunctionArgumentError.java`

        Written by Benjamin Rapaport and Jason Halpern.

    3. `MissingReturnError.java`

        Written by Paul Tylkin.

    4. `TypeMismatchError.java`

        Written by Samuel Messing.

    5. `UnreachableCodeError.java`

        Written by Paul Tylkin.

    6. `VariableRedefinedError.java`

        Written by Benjamin Rapaport and Jason Halpern.

    7. `VariableUndeclaredError.java`

        Written by Benjamin Rapaport and Jason Halpern.

- `util.logging` package

    1. `BreifLogFormatter.java`

        Specifies default behavior of the logger. Written by Benjamin Rapaport.

- `util.symbol_table` package

    1. `FunctionSymbol.java`

        Used by the Symbol Table class, represents a Function. Written by Jason Halpern and Benjamin Rapaport.

    2. `Method.java`

        Deprecated. Old class used by Symbol Table. Written by Jason Halpern and Kurry Tran.

    3. `ReservedSymTable.java`

        Deprecated. Old class used by Symbol Table. Written by Jason Halpern.

4. `ReservedWordSymbol.java`

   Represents reserved words in the Symbol Table. Written by Jason Halpern and Benjamin Rapaport.

5. `Symbol.java`

   Represents a symbol in the Symbol Table. Written by Jason Halpern and Benjamin Rapaport.

6. `SymbolTable.java`

   The symbol table class, used to create the Symbol Table. Written by Jason Halpern, Benjamin Rapaport, Kurry Tran and Paul Tylkin.

7. `VariableSymbol.java`

   Represents a variable in the Symbol Table. Written by Jason Halpern and Benjamin Rapaport.

8. `Word.java`

   Deprecated Old class used by Symbol Table. Written by Jason Halpern.

- `util.type` package

  1. `Types.java`

     A convenience class for defining and manipulating internal type representations. Written by Samuel Messing, Benjamin Rapaport, and Jason Halpern.

Appendix B

Complete Source Code

B.1 back_end package

B.1.1 CodeGeneratingVisitor.java

```
package back_end;

import util.ast.node.*;
import util.ast.AbstractSyntaxTree;
import util.type.Types;

import java.util.logging.Logger;

/**
 * Visitor class for generating Java source.
 *
 * This is the fourth (and final) walk performed after construction of the AST
 * from source. CodeGeneratingVisitor generates a massive String representing
 * the translated Hog program.
 *
 *
 * @author Samuel Messing
 * @author Kurry Tran
 *
 */
public class CodeGeneratingVisitor implements Visitor {

protected final static Logger LOGGER = Logger
.getLogger(CodeGeneratingVisitor.class.getName());
/**
 * The format of the input files to the <code>map</code> class. Currently
 * Hog only supports text formats, so this doesn't need to be set by the
```

```
 * constructor.
 */
protected final String inputFormatClass = "TextInputFormat.class";
/**
 * The format of the output files from the <code>reduce</code> class. See
 * {@link #inputFormatClass inputFormatClass} for more details.
 */
protected final String outputFormatClass = "TextOutputFormat.class";

protected AbstractSyntaxTree tree;
protected StringBuilder code;
protected String outputKeyClass;
protected String outputValueClass;
protected String inputFile = "args[0]";
protected String outputFile = "args[1]";
/**
 * Remember when recursing if we're dealing with a declaration statement, as
 * the handling both DerivedTypeNodes and IdNodes is context-specific.
 */
protected boolean declarationStatement = false;
protected boolean rValue = false;
/**
 * Remember if we're writing the emit() function, as we need to cast to
 * Hadoop's Writable types;
 */
protected boolean emit = false;

/**
 * Construct a CodeGeneratingVisitor, but don't specify input file or output
 * file.
 * <p>
 * Mainly used for testing/development purposes.
 *
 * @param root
 *            the root of the AST representing the Hog source program.
 */
public CodeGeneratingVisitor(AbstractSyntaxTree root) {

this.tree = root;
this.code = new StringBuilder();

}

/**
 * Construct a CodeGeneratingVisitor, specifying the input file name and the
 * output file name for the corresponding Hadoop job.
```

```
 *
 * <pre>
 * public {@link CodeGeneratingVisitor}({@link AbstractSyntaxTree} root, {@link String} inpu
 * </pre>
 *
 * @param root
 *            The root node of the Hog source program's AST.
 * @param inputFile
 *            The inputFile for the <code>map</code> class to read from.
 * @param outputFile
 *            The outputFile for the <code>reduce</code> class to write to.
 */
public CodeGeneratingVisitor(AbstractSyntaxTree root, String inputFile,
String outputFile) {

this(root);
this.inputFile = inputFile;
this.outputFile = outputFile;

}

/**
 * Return the Java source code translated from the AST.
 *
 * @return a string representation (formatted) of the java source code.
 */
public String getCode() {
formatCode();
return code.toString();
}

@Override
public void walk() {

writeHeader();

// start recursive walk:
walk(tree.getRoot());

code.append("}");

}

private void walk(Node node) {

node.accept(this);
```

```
if (node.isEndOfLine()) {
writeStatement();
}

// base cases (sometimes recursion needs to go through visit methods
// as with If Else statements).

boolean baseCase = false;

if (node instanceof IfElseStatementNode) {
baseCase = true;
} else if (node instanceof SectionNode) {
baseCase = true;
} else if (node instanceof FunctionNode) {
baseCase = true;
} else if (node instanceof JumpStatementNode) {
baseCase = true;
} else if (node instanceof StatementNode) {
baseCase = true;
} else if (node instanceof StatementListNode) {
baseCase = true;
} else if (node.getChildren().isEmpty()) {
baseCase = true;
}

// continue recursion if not base case:

if (!baseCase) {
for (Node child : node.getChildren()) {
walk(child);
}
}

}

private void writeHeader() {
LOGGER.fine("Writing header to code.");
code.append("import java.io.IOException;");
code.append("import java.util.*;");
code.append("import org.apache.hadoop.fs.Path;");
code.append("import org.apache.hadoop.conf.*;");
code.append("import org.apache.hadoop.util.*;");
code.append("import org.apache.hadoop.io.*;");
code.append("import org.apache.hadoop.mapred.*;");
code.append("public class Hog {");
```

```
}

private void writeMapReduce() {
LOGGER.fine("Writing mapReduce initialization code.");
code.append("JobConf conf = new JobConf(Hog.class);");
code.append("conf.setJobName(\"hog\");");
code.append("conf.setOutputKeyClass(" + outputKeyClass + ".class);");
code
.append("conf.setOutputValueClass(" + outputValueClass
+ ".class);");
code.append("conf.setMapperClass(Map.class);");
code.append("conf.setCombinerClass(Reduce.class);");
code.append("conf.setReducerClass(Reduce.class);");
code.append("conf.setInputFormat(" + inputFormatClass + ");");
code.append("conf.setOutputFormat(" + outputFormatClass + ");");
code.append("FileInputFormat.setInputPaths(conf, new Path(" + inputFile
+ "));");
code.append("FileOutputFormat.setOutputPath(conf, new Path("
+ outputFile + "));");
code.append("JobClient.runJob(conf);");
}

/**
 * Write the end of a statement.
 *
 * Protects against writing multiple semicolons, as an ease for the
 * programmer.
 */
private void writeStatement() {
if (!code.toString().endsWith("}") && !code.toString().endsWith(";")) {
code.append(";");
}
}

/**
 * <code>this.code</code> is originally built as a monolithic string without
 * newlines and other formatting. <code>formatCode()</code> adds both
 * newlines after statements and proper indentation based on scope.
 */
private void formatCode() {
int scopeCount = 0;
StringBuilder indentedCode = new StringBuilder();
// all the booleans below are strictly for pretty printing for loops
boolean withinForDeclaration = false;
boolean f = false;
boolean o = false;
```

```
int forSemicolonCount = 0;
for (int i = 0; i < code.length(); i++) {
switch (code.charAt(i)) {
case '{':
scopeCount++;
indentedCode.append("{\n");
indentedCode.append(repeat(' ', 4 * scopeCount));
break;
case '}':
scopeCount--;
// we're reducing scope, so need to undo the spaces previously
// written
indentedCode.delete(indentedCode.length() - 4, indentedCode
.length());
indentedCode.append("}\n");
indentedCode.append(repeat(' ', 4 * scopeCount));
break;
case ';':
if (withinForDeclaration && forSemicolonCount < 2) {
indentedCode.append(';');
forSemicolonCount++;
} else {
indentedCode.append(";\n");
forSemicolonCount = 0;
withinForDeclaration = false;
}
if (!withinForDeclaration)
indentedCode.append(repeat(' ', 4 * scopeCount));
break;
case 'f':
f = true;
indentedCode.append(code.charAt(i));
break;
case 'o':
if (f)
o = true;
indentedCode.append(code.charAt(i));
break;
case 'r':
if (f && o) {
withinForDeclaration = true;
}
indentedCode.append(code.charAt(i));
break;
case ':':
f = false;
```

```
o = false;
withinForDeclaration = false;
indentedCode.append(code.charAt(i));
break;
default:
if (!withinForDeclaration) {
f = false;
o = false;
}
indentedCode.append(code.charAt(i));
}
}
code = indentedCode;
}

/**
 * Repeat a character n times.
 *
 * @param toRepeat
 *            the character to repeat
 * @param times
 *            the number of times to repeat <code>toRepeat</code>
 * @return the String formed by repeating <code>toRepeat</code> n=
 *         <code>times</code> times in a row.
 */
private String repeat(char toRepeat, int times) {
StringBuilder repeated = new StringBuilder();
for (int i = 0; i < times; i++)
repeated.append(toRepeat);
return repeated.toString();
}

@Override
public void visit(ArgumentsNode node) {
LOGGER.finer("visit(ArgumentsNode node) called on " + node);

if (node.hasMoreArgs()) {
walk(node.getMoreArgs());
code.append(", ");
}

walk(node.getArg());

}

@Override
```

```
public void visit(BiOpNode node) {
LOGGER.finer("visit(BiOpNode node) called on " + node);

walk(node.getLeftNode());
switch (node.getOpType()) {
case ASSIGN:
code.append(" = ");
rValue = true;
break;
case DBL_EQLS:
code.append(" == ");
break;
case NOT_EQLS:
code.append(" != ");
break;
case PLUS:
code.append(" + ");
break;
case OR:
code.append(" || ");
break;
case TIMES:
code.append(" * ");
break;
case MINUS:
code.append(" - ");
break;
case LESS:
code.append(" < ");
break;
case LESS_EQL:
code.append(" <= ");
break;
case GRTR:
code.append(" > ");
break;
case GRTR_EQL:
code.append(" >= ");
break;
case DIVIDE:
code.append(" / ");
break;
case MOD:
code.append(" % ");
break;
case AND:
```

```
code.append(" && ");
break;
}

walk(node.getRightNode());

// unset declaration flag that may have been set (when node.getOpType ==
// ASSIGN)
declarationStatement = false;
// unset the rValue flag that may have been set (when node.getOpType ==
// ASSIGN)
rValue = false;

}

@Override
public void visit(CatchesNode node) {
LOGGER.finer("visit(CatchesNode node) called on " + node);

if (node.hasNext())
walk(node.getNext());
code.append("catch (");
walk(node.getHeader());
code.append(") {");
if (node.hasBlock())
walk(node.getBlock());
code.append(" }");

}

@Override
public void visit(ConstantNode node) {
LOGGER.finer("visit(ConstantNode node) called on " + node);
LOGGER.finer("Value: " + node.getValue());

if (emit) {
walk(node.getType());
code.append("(");
}

code.append(node.getValue());

if (emit)
code.append(")");
}
```

```
@Override
public void visit(DerivedTypeNode node) {
LOGGER.finer("visit(DerivedTypeNode node) called on " + node);

if (declarationStatement)
code.append("new ");

switch (node.getLocalType()) {
case LIST:
if (declarationStatement && rValue)
code.append("ArrayList<");
else
code.append("List<");
break;
case ITER:
if (declarationStatement) {
throw new UnsupportedOperationException("Cannot declare Iterators!");
}
code.append("Iterator<");
break;
case SET:
if (declarationStatement && rValue)
code.append("HashSet<");
else
code.append("Set<");
break;
}

// remember state for this particular node, but forget it for recursing
boolean declaration = declarationStatement;
declarationStatement = false;

if (node.getInnerTypeNode() instanceof PrimitiveTypeNode) {
code.append(Types.getJavaObjectType((PrimitiveTypeNode) node
.getInnerTypeNode()));
} else
walk(node.getInnerTypeNode());

// close inner types
code.append(">");

if (declaration)
code.append("()");

}
```

```
@Override
public void visit(ElseIfStatementNode node) {
LOGGER.finer("visit(ElseIfStatementNode node) called on " + node);
code.append("} else if (");
walk(node.getCondition());
code.append(") {");
walk(node.getIfCondTrue());
if (node.getIfCondFalse() != null) {
walk(node.getIfCondFalse());
}

}

@Override
public void visit(ElseStatementNode node) {
LOGGER.finer("visit(ElseStatementNode node) called on " + node);
code.append("} else {");
walk(node.getBlock());

}

@Override
public void visit(ExceptionTypeNode node) {
LOGGER.finer("visit(ExceptionTypeNode node) called on " + node);

switch (node.getExceptionType()) {
case ARITHMETIC:
code.append("ArithmeticException");
break;
case ARRAY_OUT_OF_BOUNDS:
code.append("IndexOutOfBoundsException");
break;
case FILE_LOAD:
case FILE_NOT_FOUND:
code.append("IOException");
break;
case INCORRECT_ARGUMENT:
code.append("IllegalArgumentException");
break;
case TYPE_MISMATCH:
code.append("TypeMismatchException");
break;
case NULL_REFERENCE:
code.append("NullPointerException");
break;
}
```

```
}

@Override
public void visit(ExpressionNode node) {
LOGGER.finer("visit(ExpressionNode node) called on " + node);
// ExpressionNode is to general, so move to a more specific case:
node.accept(this);
}

@Override
public void visit(FunctionNode node) {
LOGGER.finer("visit(FunctionNodeNode node) called on " + node);
code.append("public static ");
walk(node.getType());
code.append(" ");
code.append(node.getIdentifier());
ParametersNode params = node.getParametersNode();
code.append("(");
walk(params);
code.append(")");
code.append(" {");
walk(node.getInstructions());
code.append(" }");

}

@Override
public void visit(GuardingStatementNode node) {
LOGGER.finer("visit(GuardingStatementNode node) called on " + node);

code.append("try {");
walk(node.getBlock());
code.append(" }");
if (node.hasCatches())
walk(node.getCatches());
if (node.hasFinally()) {
code.append("finally {");
walk(node.getFinally());
code.append(" }");
}

}

@Override
public void visit(IdNode node) {
```

```
LOGGER
.finer("visit(IdNode node) called on " + node + ", emit: "
+ emit);

if (node.isDeclaration()) {
walk(node.getType());
code.append(" ");
// set a flag so when writing the right side of an assignment
// statement
// we handle things appropriately.
declarationStatement = true;
}

if (emit) {
walk(node.getType());
code.append("(");
}

code.append(node.getIdentifier());

// derived IdNodes need to be instantiated manually
if (declarationStatement && node.getType() instanceof DerivedTypeNode) {
code.append(" = ");
rValue = true;
walk(node.getType());
rValue = false;
} else if (emit) {
code.append(")");
}

}

@Override
public void visit(IfElseStatementNode node) {
LOGGER.finer("visit(IfElseStatementNode node) called on " + node);
code.append("if (");
walk(node.getCondition());
code.append(") {");
walk(node.getIfCondTrue());
// check that buffer cleared
if (node.getCheckNext() != null) {
walk(node.getCheckNext());
}
if (node.getIfCondFalse() != null) {
walk(node.getIfCondFalse());
}
```

```
code.append("}");
}

@Override
public void visit(IterationStatementNode node) {
LOGGER.finer("visit(IterationStatementNode node) called on " + node);

switch (node.getIterationType()) {
case FOR:
code.append("for (");
walk(node.getInitial());
code.append("; ");
walk(node.getCheck());
code.append("; ");
walk(node.getIncrement());
code.append(") {");
walk(node.getBlock());
writeStatement();
break;
case FOREACH:
code.append("for (");
code.append(Types.getJavaType(node.getPart().getType()));
code.append(" ");
walk(node.getPart());
code.append(" : ");
walk(node.getWhole());
code.append(") {");
walk(node.getBlock());
writeStatement();
break;
case WHILE:
code.append("while (");
walk(node.getCheck());
code.append(") {");
walk(node.getBlock());
writeStatement();
break;
}
code.append(" }");
}

@Override
public void visit(JumpStatementNode node) {
LOGGER.finer("visit(JumpStatementNode node) called on " + node);

switch (node.getJumpType()) {
```

```
case RETURN:
code.append("return ");
break;
case BREAK:
code.append("break");
break;
case CONTINUE:
code.append("continue");
break;
}

if (node.getExpressionNode() != null) {
walk(node.getExpressionNode());
}

writeStatement();
}

@Override
public void visit(MockNode node) {
LOGGER.finer("visit(MockNode node) called on " + node);

}

@Override
public void visit(MockExpressionNode node) {
LOGGER.finer("visit(MockExpressionNode node) called on " + node);

}

@Override
public void visit(Node node) {
LOGGER.finer("visit(Node node) called on " + node);

}

@Override
public void visit(ParametersNode node) {
LOGGER.finer("visit(ParametersNode node) called on " + node);

if (node.hasParamChild()) {
walk(node.getParamChild());
code.append(", ");
}

walk(node.getType());
```

```
code.append(" ");
code.append(node.getIdentifier());

// walk(node.getParamChild());

}

@Override
public void visit(PostfixExpressionNode node) {
LOGGER.finer("visit(PostfixExpressionNode node) called on " + node);

switch (node.getPostfixType()) {
case METHOD_NO_PARAMS:
IdNode objectOfMethod = node.getObjectOfMethod();
IdNode methodNameNoParam = node.getMethodName();
code.append(objectOfMethod.getIdentifier() + "."
+ methodNameNoParam.getIdentifier() + "()");
if (methodNameNoParam.getIdentifier().equals("next")) {
code.append(".get()");
}
break;
case METHOD_WITH_PARAMS:
IdNode object = node.getObjectName();
IdNode method = node.getMethodName();
code.append(object.getIdentifier());
code.append(".");
if (method.getIdentifier().equals("tokenize"))
code.append("split");
else
code.append(method.getIdentifier());
code.append("(");
walk(node.getArgsList());
code.append(")");
break;
case FUNCTION_CALL:
IdNode functionIdNode = node.getFunctionName();
// check if this is our special mapReduce() call:
if (functionIdNode.getIdentifier().equals("mapReduce")) {
writeMapReduce();
return;
}
// check if this is a cast function:
String functionName = functionIdNode.getIdentifier();
if (functionName.equals("text2int")) {
code.append("Integer.parseInt(");
walk(node.getArgsList());
```

```
code.append(")");
return;
} else if (functionName.equals("int2text")) {
code.append("Integer.toString(");
walk(node.getArgsList());
code.append(")");
return;
} else if (functionName.equals("text2real")) {
code.append("Double.parseDouble(");
walk(node.getArgsList());
code.append(")");
return;
} else if (functionName.equals("real2text")) {
code.append("Double.toString(");
walk(node.getArgsList());
code.append(")");
return;
} else if (functionName.equals("real2int")) {
code.append("(int) ");
walk(node.getArgsList());
return;
} else if (functionName.equals("int2real")) {
code.append("(double) ");
walk(node.getArgsList());
return;
}
// check if this is a print statement:
if (functionName.equals("print")) {
code.append("System.out.println(");
walk(node.getArgsList());
code.append(")");
return;
}

if (!node.getFunctionName().getIdentifier().equals("emit")) {
code.append("Functions.");
walk(node.getFunctionName());
} else {
code.append("output.collect");
emit = true;
}

code.append("(");

// check for arguments
if (node.hasArguments())
```

```
walk(node.getArgsList());

code.append(")");

// unset emit flag which may have been set:
emit = false;

}
}

@Override
public void visit(PrimaryExpressionNode node) {
LOGGER.finer("visit(PrimaryExpressionNode node) called on " + node);
code.append(node.toSource());

}

@Override
public void visit(PrimitiveTypeNode node) {
LOGGER.finer("visit(PrimitiveTypeNode node) called on " + node);

if (emit) {
code.append("new ");
code.append(Types.getHadoopType(node));
} else {
code.append(Types.getJavaType(node));
}

}

@Override
public void visit(ProgramNode node) {
LOGGER.finer("visit(ProgramNode node) called on " + node);

}

@Override
public void visit(RelationalExpressionNode node) {
LOGGER.finer("visit(RelationalNode node) called on " + node);
throw new UnsupportedOperationException(
"I should never see a relational expression node!");
}

@Override
public void visit(ReservedWordTypeNode node) {
LOGGER.finer("visit(ReservedWordTypeNode node) called on " + node);
```

```
throw new UnsupportedOperationException(
"I shouldn't be seeing a ReservedWordTypeNode!");
}

@Override
public void visit(SectionNode node) {
LOGGER.finer("visit(SectionNode node) called on " + node);

SectionNode.SectionName sectionKind = node.getSectionName();

switch (sectionKind) {
case FUNCTIONS:
code.append("public static class Functions {");
break;
case MAP:
code
.append("public static class Map extends MapReduceBase implements Mapper");
walk(node.getSectionTypeNode());
break;
case REDUCE:
code
.append("public static class Reduce extends MapReduceBase implements Reducer");
walk(node.getSectionTypeNode());

break;
case MAIN:
code
.append("public static void main(String[] args) throws Exception {");
break;
}
walk(node.getBlock());
code.append("}");

// need to write an additional block for inner methods in reduce and
// map:

switch (sectionKind) {
case MAP:
case REDUCE:
code.append("}");
}

}

@Override
public void visit(SectionTypeNode node) {
```

```
LOGGER.finer("visit(SectionTypeNode node) called on " + node);
// if we're at @Reduce, need to see output types for main
if (node.getSectionParent().getSectionName() == SectionNode.SectionName.REDUCE) {
outputKeyClass = Types.getHadoopType((PrimitiveTypeNode) node
.getReturnKey());
outputValueClass = Types.getHadoopType((PrimitiveTypeNode) node
.getReturnValue());
}

code.append("<");
if (node.getSectionParent().getSectionName() == SectionNode.SectionName.MAP) {
code.append("LongWritable");
} else
code
.append(Types.getHadoopType(node.getInputKeyIdNode()
.getType()));
code.append(", ");
code.append(Types.getHadoopType(node.getInputValueIdNode().getType()));
code.append(", ");
code.append(Types.getHadoopType(node.getReturnKey()));
code.append(", ");
code.append(Types.getHadoopType(node.getReturnValue()));
code.append("> {");

if (node.getSectionParent().getSectionName() == SectionNode.SectionName.REDUCE) {
code.append("public void reduce(");
code
.append(Types.getHadoopType(node.getInputKeyIdNode()
.getType()));
code.append(" ");
code.append(node.getInputKeyIdNode().getIdentifier());
code.append(", Iterator<");
code.append(Types.getHadoopType(node.getInputValueIdNode()
.getType()));
code.append("> ");
code.append(node.getInputValueIdNode().getIdentifier());
code.append(",  OutputCollector<");
code.append(Types.getHadoopType(node.getReturnKey()));
code.append(", ");
code.append(Types.getHadoopType(node.getReturnValue()));
code.append("> output, Reporter reporter) throws IOException {");
} else {
code.append("public void map(");
code.append("LongWritable ");
code.append(node.getInputKeyIdNode().getIdentifier());
code.append(", ");
```

```
code.append(Types.getHadoopType(node.getInputValueIdNode()
.getType()));
code.append(" value,  OutputCollector<");
code.append(Types.getHadoopType(node.getReturnKey()));
code.append(", ");
code.append(Types.getHadoopType(node.getReturnValue()));
code.append("> output, Reporter reporter) throws IOException {");
code.append("String line = value.toString();");
}

}

@Override
public void visit(SelectionStatementNode node) {
LOGGER.finer("visit(SelectionStatementNode node) called on " + node);
throw new UnsupportedOperationException(
"I should never see a SelectionStatementNode!");

}

@Override
public void visit(StatementListNode node) {
LOGGER.finer("visit(StatementListNode node) called on " + node);
for (Node child : node.getChildren()) {
walk(child);
writeStatement();
}

}

@Override
public void visit(StatementNode node) {
LOGGER.finer("visit(StatementNode node) called on " + node);
for (Node child : node.getChildren()) {
walk(child);
}
writeStatement();
}

@Override
public void visit(SwitchStatementNode node) {
LOGGER.finer("visit(SwitchStatementNode node) called on " + node);
throw new UnsupportedOperationException(
"Switch statements are not supported!");
}
```

```
@Override
public void visit(TypeNode node) {
LOGGER.finer("visit(TypeNode node) called on " + node);
// type node is too general, so call something more specific:
node.accept(this);
}

@Override
public void visit(UnOpNode node) {
LOGGER.finer("visit(UnOpNode node) called on " + node);

switch (node.getOpType()) {
case UMINUS:
code.append("-");
walk(node.getChildNode());
break;
case NOT:
code.append("!");
walk(node.getChildNode());
break;
case INCR:
walk(node.getChildNode());
code.append("++");
break;
case DECR:
walk(node.getChildNode());
code.append("--");
break;
case CAST:
throw new UnsupportedOperationException(
"Cast statements are NOT supported yet!");
case NONE:
// none means no unary operator applied.
walk(node.getChildNode());
break;
}

}

}
```

B.1.2 ErrorCheckingVisitor.java

```
/**
 *
 */
```

```
package back_end;

import java.util.ArrayList;
import java.util.List;
import java.util.logging.Logger;

import util.type.Types;

import util.ast.AbstractSyntaxTree;
import util.ast.node.ArgumentsNode;
import util.ast.node.BiOpNode;
import util.ast.node.CatchesNode;
import util.ast.node.ConstantNode;
import util.ast.node.DerivedTypeNode;
import util.ast.node.ElseIfStatementNode;
import util.ast.node.ElseStatementNode;
import util.ast.node.ExceptionTypeNode;
import util.ast.node.ExpressionNode;
import util.ast.node.FunctionNode;
import util.ast.node.GuardingStatementNode;
import util.ast.node.IdNode;
import util.ast.node.IfElseStatementNode;
import util.ast.node.IterationStatementNode;
import util.ast.node.JumpStatementNode;
import util.ast.node.JumpStatementNode.JumpType;
import util.ast.node.MockExpressionNode;
import util.ast.node.MockNode;
import util.ast.node.Node;
import util.ast.node.ParametersNode;
import util.ast.node.PostfixExpressionNode;
import util.ast.node.PrimaryExpressionNode;
import util.ast.node.PrimitiveTypeNode;
import util.ast.node.ProgramNode;
import util.ast.node.RelationalExpressionNode;
import util.ast.node.ReservedWordTypeNode;
import util.ast.node.SectionNode;
import util.ast.node.SectionTypeNode;
import util.ast.node.SelectionStatementNode;
import util.ast.node.StatementListNode;
import util.ast.node.StatementNode;
import util.ast.node.SwitchStatementNode;
import util.ast.node.TypeNode;
import util.ast.node.UnOpNode;
import util.error.MissingReturnError;

/**
```

```
 * Visitor class for error checking.
 *
 * This is the second walk performed after construction of the AST from source.
 *
 * Performs the following validations: - no dead code (statements after a return
 * statement in the same basic block) - no break/continue statements outside of
 * iteration loops - non-void functions have adequate number of return
 * statements - no case/default statements outside of immediate switch statement
 * - can't catch same exception type more than once
 *
 *
 * @author Paul Tylkin
 *
 */

public class ErrorCheckingVisitor implements Visitor {

protected AbstractSyntaxTree tree;

protected final static Logger LOGGER = Logger
.getLogger(ErrorCheckingVisitor.class.getName());

private static List<Boolean> returnFlagStack = new ArrayList<Boolean>();

private static void pushReturnStack() {
returnFlagStack.add(returnFlagStack.get(returnFlagStack.size() - 1));
}

private static void popReturnStack() {
returnFlagStack.remove(returnFlagStack.size() - 1);
}

public ErrorCheckingVisitor(AbstractSyntaxTree tree) {
this.tree = tree;
}

public void walk() {
returnFlagStack.add(false);
ProgramNode treeRoot = (ProgramNode) this.tree.getRoot();
visitReturnChildren(treeRoot);
visitFunctionReturns(treeRoot);
}

private void visitReturnChildren(Node node) {
/*
 * Constructs of the form
```

```
 *
 * int doubleint (int x){ return 2*x;a x = 2*x; }
 *
 * will not compile into Java, and so if a Hog program has this
 * construct, Hog will throw an unreachable code exception. Similarly,
 *
 * int max(int a, int b){ if (a > b){ return a; } else{ return b; }
 * return 0; }
 *
 * will also throw an error in Java.
 *
 * This pass through the program will check that there is no unreachable
 * code in a Hog program.
 */
if (node.isNewScope()) {
LOGGER.finer("We are in a new scope now in node " + node);
this.pushReturnStack();
}
if (returnFlagStack.get(returnFlagStack.size() - 1)) {
// throw new UnreachableCodeError(
// "The following statement is unreachable: "+ node.toSource());
}
if (node instanceof JumpStatementNode
&& ((JumpStatementNode) node).getJumpType() == JumpType.RETURN) { // TODO
// check
// that
// is
// a
// return
// node
if (returnFlagStack.get(returnFlagStack.size() - 1)) {
// throw new UnreachableCodeError(
// "The following statement is unreachable: "+ node.toSource());
} else {
returnFlagStack.set(returnFlagStack.size() - 1, true);
}
} else {
List<Node> children = node.getChildren();
for (Node n : children) {
visitReturnChildren(n);
}
}

if (node.isNewScope()) {
this.popReturnStack();
}
```

```
}

private boolean nonVoidFunctionFlag;

private void visitFunctionReturns(Node node) {
/*
 * Constructs of the form int max(int a, int b){ if (a > b){ return a; }
 * if (b >= a){ return b; } } will not compile into Java, saying that
 * there is an error in the return type, even though the function does
 * return the correct value. This pass through the Hog program will
 * throw Hog errors on this type of input.
 */
if (node instanceof FunctionNode) {
List<Node> children = node.getChildren();

// if return type is not void
if (!Types.isVoidType(((FunctionNode) node).getType())) {
this.nonVoidFunctionFlag = true;
for (Node n : children) {
if (n instanceof JumpStatementNode
&& ((JumpStatementNode) n).getJumpType() == JumpType.RETURN) {
this.nonVoidFunctionFlag = false;
}
}

/*if (this.nonVoidFunctionFlag) { //
throw new MissingReturnError(
"The following function is missing a return statement: "
+ node.toSource());
}*/
}

}
// System.out.println(nonVoidFunctionFlag);
List<Node> children = node.getChildren();

for (Node n : children) {
visitFunctionReturns(n);
}
}

@Override
public void visit(ArgumentsNode node) {
LOGGER.finer("visit(ArgumentsNode node) called on " + node);

}
```

```
@Override
public void visit(BiOpNode node) {
LOGGER.finer("visit(BiOpNode node) called on " + node);

}

@Override
public void visit(CatchesNode node) {
LOGGER.finer("visit(CatchesNode node) called on " + node);
}

@Override
public void visit(ConstantNode node) {
LOGGER.finer("visit(ConstantNode node) called on " + node);

}

@Override
public void visit(DerivedTypeNode node) {
LOGGER.finer("visit(DerivedTypeNode node) called on " + node);
}

@Override
public void visit(ElseIfStatementNode node) {
LOGGER.finer("visit(ElseIfStatementNode node) called on " + node);
}

@Override
public void visit(ElseStatementNode node) {
LOGGER.finer("visit(ElseStatementNode node) called on " + node);
}

@Override
public void visit(ExceptionTypeNode node) {
LOGGER.finer("visit(ExceptionTypeNode node) called on " + node);
}

public void visit(ExpressionNode node) {
LOGGER.finer("visit(ExpressionNode node) called on " + node);
}

public void visit(FunctionNode node) {
LOGGER.finer("visit(FunctionNode node) called on " + node);
}
```

```
@Override
public void visit(GuardingStatementNode node) {
LOGGER.finer("visit(GuardingStatementNode node) called on " + node);
}

@Override
public void visit(IdNode node) {
LOGGER.finer("visit(IdNode node) called on " + node);
}

@Override
public void visit(IfElseStatementNode node) {
LOGGER.finer("visit(IfElseStatementNode node) called on " + node);

}

@Override
public void visit(IterationStatementNode node) {
LOGGER.finer("visit(IterationStatementNode node) called on " + node);

}

@Override
public void visit(JumpStatementNode node) {
LOGGER.finer("visit(JumpStatementNode node) called on " + node);

}

@Override
public void visit(MockExpressionNode node) {
LOGGER.finer("visit(MockExpressionNode node) called on " + node);
}

@Override
public void visit(MockNode node) {
LOGGER.finer("visit(MockNode node) called on " + node);
}

@Override
public void visit(Node node) {
LOGGER.finer("visit(Node node) called on " + node);
}

@Override
public void visit(ParametersNode node) {
LOGGER.finer("visit(ParametersNode node) called on " + node);
```

```
}

@Override
public void visit(PostfixExpressionNode node) {
LOGGER.finer("visit(PostfixExpressionNode node) called on " + node);
}

@Override
public void visit(PrimaryExpressionNode node) {
LOGGER.finer("visit(PrimaryExpressionNode node) called on " + node);
}

@Override
public void visit(PrimitiveTypeNode node) {
LOGGER.finer("visit(PrimitiveTypeNode node) called on " + node);
}

@Override
public void visit(ProgramNode node) {
LOGGER.finer("visit(ProgramNode node) called on " + node);
}

@Override
public void visit(RelationalExpressionNode node) {
LOGGER.finer("visit(RelationalExpressionNode node) called on " + node);
}

@Override
public void visit(SectionNode node) {
LOGGER.finer("visit(SectionNode node) called on " + node);
}

@Override
public void visit(SectionTypeNode node) {
LOGGER.finer("visit(SectionTypeNode node) called on " + node);
}

@Override
public void visit(SelectionStatementNode node) {
LOGGER.finer("visit(SelectionStatementNode node) called on " + node);
}

@Override
public void visit(StatementListNode node) {
LOGGER.finer("visit(StatementListNode node) called on " + node);
}
```

```
@Override
public void visit(StatementNode node) {
LOGGER.finer("visit(StatementNode node) called on " + node);
}

@Override
public void visit(SwitchStatementNode node) {
LOGGER.finer("visit(SwitchStatementNode node) called on " + node);
}

@Override
public void visit(TypeNode node) {
LOGGER.finer("visit(TypeNode node) called on " + node);
}

@Override
public void visit(UnOpNode node) {
LOGGER.finer("visit(UnOpNode node) called on " + node);
}

@Override
public void visit(ReservedWordTypeNode node) {
LOGGER.finer("visit(ReservedWordTypeNode node) called on " + node);
}
}
```

B.1.3 SymbolTableVisitor.java

```
package back_end;

import java.util.ArrayList;
import java.util.List;
import java.util.logging.Logger;

import util.ast.AbstractSyntaxTree;
import util.ast.node.*;
import util.error.VariableRedefinedError;
import util.error.VariableUndeclaredError;
import util.symbol_table.FunctionSymbol;
import util.symbol_table.Symbol;
import util.symbol_table.SymbolTable;
import util.symbol_table.VariableSymbol;
import util.type.Types;
```

```
/**
 *
 * @author Benjamin Rapaport
 * @author Jason Halpern
 *
 */

public class SymbolTableVisitor implements Visitor {

protected AbstractSyntaxTree tree;

public SymbolTableVisitor(AbstractSyntaxTree tree) {
this.tree = tree;
}

protected final static Logger LOGGER = Logger
.getLogger(SymbolTableVisitor.class.getName());

public void walk() {

ProgramNode treeRoot = (ProgramNode) this.tree.getRoot();
treeRoot.accept(this);
}

private void openScope(Node node) {

// LOGGER.warning("Calling openScope for " + node.getName());
// push new scope
if(node.isNewScope()) {
// LOGGER.warning("Scope opening for " + node.getName());
SymbolTable.push();

// map this as representative node
try {
SymbolTable.mapNode(node);
} catch (Exception e) {
e.printStackTrace();
System.exit(1);
}
}

}

private void closeScope(Node node) {
```

```
// pop if this was a new scope
if(node.isNewScope()) {
SymbolTable.pop();
}

}

private void visitAllChildrenStandard(Node node) {
// visit all children
List<Node> children = node.getChildren();
for (Node n : children) {
n.accept(this);
}
}

@Override
public void visit(ArgumentsNode node) {
LOGGER.finer("visit(ArgumentsNode node) called on " + node.getName());
openScope(node);
visitAllChildrenStandard(node);
closeScope(node);
}

@Override
public void visit(BiOpNode node) {
LOGGER.finer("visit(BiOpNode node) called on " + node.getName());
openScope(node);
LOGGER.finer("The children of this BiNoNode are: " + node.getChildrenString());
visitAllChildrenStandard(node);
closeScope(node);
}

@Override
public void visit(CatchesNode node) {
LOGGER.finer("visit(CatchesNode node) called on " + node.getName());

openScope(node);

node.getHeader().accept(this);
if (node.hasBlock())
node.getBlock().accept(this);
closeScope(node);
if (node.hasNext())
node.getNext().accept(this);

}
```

```
@Override
public void visit(ConstantNode node) {
LOGGER.finer("visit(Constant node) called on " + node.getName());
openScope(node);
visitAllChildrenStandard(node);
closeScope(node);

}

@Override
public void visit(DerivedTypeNode node) {
LOGGER.finer("visit(DerivedType node) called on " + node.getName());
openScope(node);
visitAllChildrenStandard(node);
closeScope(node);

}

@Override
public void visit(ElseIfStatementNode node) {
LOGGER.finer("visit(ElseIfStatementNodt node) called on " + node.getName());
openScope(node);
visitAllChildrenStandard(node);
closeScope(node);
}

@Override
public void visit(ElseStatementNode node) {
LOGGER.finer("visit(ElseStatementNode node) called on " + node.getName());
openScope(node);
visitAllChildrenStandard(node);
closeScope(node);

}

@Override
public void visit(ExceptionTypeNode node) {
LOGGER.finer("visit(ExceptionTypeNode node) called on " + node.getName());
openScope(node);
visitAllChildrenStandard(node);
closeScope(node);
}

@Override
public void visit(ExpressionNode node) {
```

```
LOGGER.finer("visit(ExpressionNode node) called on " + node.getName());
openScope(node);
visitAllChildrenStandard(node);
closeScope(node);

}

@Override
public void visit(FunctionNode node) {
LOGGER.finer("visit(FunctionNode node) called on " + node.getName());

// add function to symbol table - these need to be visible to entire program
FunctionSymbol funSym = new FunctionSymbol(node.getType(), node.getParametersNode());
SymbolTable.putToRootSymbolTable(node.getIdentifier(), funSym);

// open new scope - these are specific to within the function
openScope(node);

// add variables and types in the params list to the symbol table
ParametersNode currParamNode = node.getParametersNode();
if( currParamNode != null) {
TypeNode paramType = currParamNode.getType();
String paramName = currParamNode.getIdentifier();
try {
SymbolTable.put(paramName, new VariableSymbol(paramType));
} catch (VariableRedefinedError e) {
// TODO Auto-generated catch block
e.printStackTrace();
}

// recurse through children, adding variables to the symbol table
while(currParamNode.hasChildren()) {
currParamNode = (ParametersNode) currParamNode.getChildren().get(0);
paramType = currParamNode.getType();
paramName = currParamNode.getIdentifier();
try {
SymbolTable.put(paramName, new VariableSymbol(paramType));
} catch (VariableRedefinedError e) {
// TODO Auto-generated catch block
e.printStackTrace();
System.exit(1);
}
}
}

visitAllChildrenStandard(node);
```

```
// close scope
closeScope(node);

}

@Override
public void visit(GuardingStatementNode node) {
LOGGER.finer("visit(GuardingStatementNode node) called on " + node.getName());
openScope(node);
visitAllChildrenStandard(node);
closeScope(node);
}

@Override
public void visit(IdNode node) {
LOGGER.finer("visit(IdNode node) called on " + node.getName());
openScope(node);
// if it has a type, it is a declaration. Put it in the symbol table
if(node.getType() != null) {

LOGGER.finer("IdNode has a non null type which is: " + node.getTypeName());

try {
SymbolTable.put(node.getIdentifier(), new VariableSymbol(node.getType()));
} catch (VariableRedefinedError e) {
// TODO Auto-generated catch block
e.printStackTrace();
System.exit(1);
}
}

//else, it does not have a type, so we ensure it is already declared
else {
Symbol nodeSymbol = SymbolTable.getSymbolForIdNode(node);
// if there is no nodeSymbol, this is being used before delcartion so throw an error
if( nodeSymbol == null ) {
LOGGER.finer("IdNode was used before it was declared. Throw an error.");
throw new VariableUndeclaredError("Use of " + node.getIdentifier() + " undefined.");
}

LOGGER.finer("IdNode has a null type. Symbol is  " + nodeSymbol.toString());
// it has been declared. Now we decorate it with its type
node.setType(nodeSymbol.getType());
LOGGER.finer("We have set the IdNode type to" + nodeSymbol.getType().getName());
}
```

```
closeScope(node);
}

@Override
public void visit(IfElseStatementNode node) {
LOGGER.finer("visit(IfElseStatement node) called on " + node.getName());

// open scope
openScope(node);

// first visit the condition
if(node.getCondition() != null)
node.getCondition().accept(this);


if(node.getIfCondTrue() != null) {
LOGGER.finer("we are in the getIfCondTrue");
node.getIfCondTrue().accept(this);
}

// then close scope, then visit any remaining children
closeScope(node);

if(node.getCheckNext() != null) {

node.getCheckNext().accept(this);
}

if(node.getIfCondFalse() != null) {
node.getIfCondFalse().accept(this);
}
}

@Override
public void visit(IterationStatementNode node) {
LOGGER.finer("visit(IterationStatementNode node) called on " + node.getName());
openScope(node);
visitAllChildrenStandard(node);
closeScope(node);
}

@Override
public void visit(JumpStatementNode node) {
LOGGER.finer("visit(JumpStatementNode node) called on " + node.getName());
openScope(node);
```

```
visitAllChildrenStandard(node);
closeScope(node);
}

@Override
public void visit(MockExpressionNode node) {
LOGGER.finer("visit(MockExpressionNode node) called on " + node.getName());
openScope(node);
visitAllChildrenStandard(node);
closeScope(node);
}

@Override
public void visit(MockNode node) {
LOGGER.finer("visit(MockNode node) called on " + node.getName());
openScope(node);
visitAllChildrenStandard(node);
closeScope(node);
}

@Override
public void visit(Node node) {
LOGGER.finer("visit(Node node) called on " + node.getName());
openScope(node);
visitAllChildrenStandard(node);
closeScope(node);
}

@Override
public void visit(ParametersNode node) {
LOGGER.finer("visit(ParametersNode node) called on " + node.getName());
openScope(node);
// we have already recursed through each paramater when visiting the function node
//visitAllChildrenStandard(node);
closeScope(node);
}

@Override
public void visit(PostfixExpressionNode node) {
LOGGER.finer("visit(PostfixExpressionNode node) called on " + node.getName());
openScope(node);
visitAllChildrenStandard(node);
closeScope(node);
}

@Override
```

```
public void visit(PrimaryExpressionNode node) {
LOGGER.finer("visit(PrimaryExpressionNode node) called on " + node.getName());
openScope(node);
visitAllChildrenStandard(node);
closeScope(node);
}

@Override
public void visit(PrimitiveTypeNode node) {
LOGGER.finer("visit(PrimitiveTypeNode node) called on " + node.getName());
openScope(node);
visitAllChildrenStandard(node);
closeScope(node);
}

@Override
public void visit(ProgramNode node) {
LOGGER.finer("visit(ProgramNode node) called on " + node.getName());
openScope(node);
visitAllChildrenStandard(node);
closeScope(node);
}

@Override
public void visit(RelationalExpressionNode node) {
LOGGER.finer("visit(RelationalExpressionNode node) called on " + node.getName());
openScope(node);
visitAllChildrenStandard(node);
closeScope(node);
}

@Override
public void visit(ReservedWordTypeNode node) {
LOGGER.finer("visit(ReservedWordTypeNode node) called on " + node.getName());
openScope(node);
visitAllChildrenStandard(node);
closeScope(node);
}

@Override
public void visit(SectionNode node) {
LOGGER.finer("visit(SectionNode node) called on " + node.getName());
openScope(node);
visitAllChildrenStandard(node);
//if(node.getSectionName() != SectionNode.SectionName.FUNCTIONS){
closeScope(node);
```

```
//}
}

@Override
public void visit(SectionTypeNode node) {
LOGGER.finer("visit(SectionTypeNode node) called on " + node.getName());

openScope(node);

// add emit() with the parameters as specified by the SectionTypeNode
List<TypeNode> emitParams = new ArrayList<TypeNode>();
emitParams.add(node.getReturnKey());
emitParams.add(node.getReturnValue());
FunctionSymbol funSym = new FunctionSymbol(new PrimitiveTypeNode(Types.Primitive.VOID), emit

// add input key and value
VariableSymbol inputKeySym = new VariableSymbol(node.getInputKeyIdNode().getType());
String inputKeyStr = node.getInputKeyIdNode().getIdentifier();

VariableSymbol inputValueSym = new VariableSymbol(node.getInputValueIdNode().getType());
String inputValStr = node.getInputValueIdNode().getIdentifier();


try {
SymbolTable.put(inputKeyStr, inputKeySym);
SymbolTable.put(inputValStr, inputValueSym);
SymbolTable.put("emit", funSym);
} catch (VariableRedefinedError e) {
// TODO Auto-generated catch block
e.printStackTrace();
System.exit(1);
}




closeScope(node);
}

@Override
public void visit(SelectionStatementNode node) {
LOGGER.finer("visit(SeletionStatementNode node) called on " + node.getName());
openScope(node);
visitAllChildrenStandard(node);
closeScope(node);
}
```

```
@Override
public void visit(StatementListNode node) {
LOGGER.finer("visit(StatementListNode node) called on " + node.getName());
openScope(node);
visitAllChildrenStandard(node);
closeScope(node);
}

@Override
public void visit(StatementNode node) {
LOGGER.finer("visit(StatementNode node) called on " + node.getName());
openScope(node);
visitAllChildrenStandard(node);
closeScope(node);
}

@Override
public void visit(SwitchStatementNode node) {
LOGGER.finer("visit(SwitchStatementNode node) called on " + node.getName());
openScope(node);
visitAllChildrenStandard(node);
closeScope(node);
}

@Override
public void visit(TypeNode node) {
LOGGER.finer("visit(TypeNode node) called on " + node.getName());
openScope(node);
visitAllChildrenStandard(node);
closeScope(node);
}

@Override
public void visit(UnOpNode node) {
LOGGER.finer("visit(UnOpNode node) called on " + node.getName());
openScope(node);
visitAllChildrenStandard(node);
closeScope(node);
}
}
```

B.1.4 TypeCheckingVisitor.java

```
package back_end;
```

```
import java.util.List;
import java.util.logging.Logger;

import util.ast.AbstractSyntaxTree;
import util.ast.node.*;
import util.error.TypeMismatchError;
import util.symbol_table.FunctionSymbol;
import util.symbol_table.SymbolTable;
import util.type.Types;

/**
 * Visitor class for type checking.
 *
 * This is the first walk performed after construction of the AST from source.
 *
 * Performs the following actions:
 * - populates & propagate types through AST.
 *
 * Performs the following validations:
 * - all operands are of right type for given operator.
 * - variables are declared before they are used.
 * - return statements of functions have right type.
 * - section types of map/reduce are appropriate.
 * - actual parameters to a procedure match formal parameters.
 *
 * @author Jason Halpern
 * @author Benjamin Rapaport
 *
 */
public class TypeCheckingVisitor implements Visitor {

protected final static Logger LOGGER = Logger
.getLogger(TypeCheckingVisitor.class.getName());

protected AbstractSyntaxTree tree;

public TypeCheckingVisitor(AbstractSyntaxTree tree) {
this.tree = tree;
}

@Override
public void walk() {
// TODO Auto-generated method stub
ProgramNode treeRoot = (ProgramNode) this.tree.getRoot();
treeRoot.accept(this);
```

```
}

public TypeNode getExpectedReturnType(Node node) {
// TODO Auto-generated method stub

Node tempNode;

while(node.hasParent()){
tempNode = node.getParent();
if(tempNode instanceof FunctionNode){
return ((FunctionNode) tempNode).getType();
}
node = tempNode;
}

return null;
}

private void visitAllChildrenStandard(Node node) {
// visit all children
List<Node> children = node.getChildren();
for (Node n : children) {
n.accept(this);
}
}

@Override
public void visit(ArgumentsNode node) {
LOGGER.finer("Type Check visit(ArgumentsNode node) called on "
+ node.getName());
node.getExpressionNode().accept(this);
// set the type to the type of its child expressionNode
node.setType(node.getExpressionNode().getType());
if (node.getArgumentsNode() != null) {
node.getArgumentsNode().accept(this);
}

}

@Override
public void visit(BiOpNode node) {
LOGGER.finer("Type Check visit(BiOpNode node) called on "
+ node.getName());

// call on left and right node
ExpressionNode leftNode = node.getLeftNode();
```

```
ExpressionNode rightNode = node.getRightNode();
leftNode.accept(this);
rightNode.accept(this);

// check if they are compatible
Types.isCompatible(node.getOpType(), leftNode.getType(),
rightNode.getType());

// get return type
node.setType(Types.getResult(node.getOpType(), leftNode.getType(),
rightNode.getType()));

}

@Override
public void visit(CatchesNode node) {
LOGGER.finer("Type Check visit(CatchesNode node) called on "
+ node.getName());
visitAllChildrenStandard(node);

}

@Override
public void visit(ConstantNode node) {
LOGGER.finer("Type Check visit(ConstantNode node) called on "
+ node.getName());
visitAllChildrenStandard(node);

}

@Override
public void visit(DerivedTypeNode node) {
LOGGER.finer("Type Check visit(DerivedTypeNode node) called on "
+ node.getName());
visitAllChildrenStandard(node);

}

@Override
public void visit(ElseIfStatementNode node) {
LOGGER.finer("Type Check visit(ElseIfStatementNode node) called on "
+ node.getName());
visitAllChildrenStandard(node);

}
```

```
@Override
public void visit(ElseStatementNode node) {
LOGGER.finer("Type Check visit(ElseStatementNode node) called on "
+ node.getName());
visitAllChildrenStandard(node);

}

@Override
public void visit(ExceptionTypeNode node) {
LOGGER.finer("Type Check visit(ExceptionTypeNode node) called on "
+ node.getName());
visitAllChildrenStandard(node);

}

@Override
public void visit(ExpressionNode node) {
LOGGER.finer("Type Check visit(ExpressionNode node) called on "
+ node.getName());
visitAllChildrenStandard(node);

}

@Override
public void visit(FunctionNode node) {
LOGGER.finer("Type Check visit(FunctionNode node) called on "
+ node.getName());
visitAllChildrenStandard(node);

}

@Override
public void visit(GuardingStatementNode node) {
LOGGER.finer("Type Check visit(GuardingStatementNode node) called on "
+ node.getName());
visitAllChildrenStandard(node);

}

@Override
public void visit(IdNode node) {
LOGGER.finer("Type Check visit(IdNode node) called on "
+ node.getName());
visitAllChildrenStandard(node);
```

```
}

@Override
public void visit(IfElseStatementNode node) {
LOGGER.finer("Type Check visit(IfElseStatementNode node) called on "
+ node.getName());
visitAllChildrenStandard(node);

}

@Override
public void visit(IterationStatementNode node) {
LOGGER.finer("Type Check visit(IterationStatementNode node) called on "
+ node.getName());
visitAllChildrenStandard(node);

}

@Override
public void visit(JumpStatementNode node) {
LOGGER.finer("Type Check visit(JumpStatementNode node) called on "
+ node.getName());

//System.out.println("type check jump statement node");
visitAllChildrenStandard(node);

TypeNode expectedReturnType;
if(node.getJumpType() == JumpStatementNode.JumpType.RETURN){
//System.out.println("cehcking return");
expectedReturnType = this.getExpectedReturnType(node);
//System.out.println("EXPECTED: " + expectedReturnType + " ACTUAL:" + node.getExpressionNode
if(!Types.isSameType(node.getExpressionNode().getType(), expectedReturnType)){
throw new TypeMismatchError("Return statement " + node.getExpressionNode().getType() + "does
expectedReturnType);
}
}
}

@Override
public void visit(MockExpressionNode node) {
LOGGER.finer("" +
" visit(MockExpressionNode node) called on "
+ node.getName());
visitAllChildrenStandard(node);

}
```

```
@Override
public void visit(MockNode node) {
LOGGER.finer("Type Check visit(MockNode node) called on "
+ node.getName());
visitAllChildrenStandard(node);

}

@Override
public void visit(Node node) {
LOGGER.finer("Type Check visit(Node node) called on " + node.getName());

visitAllChildrenStandard(node);
}

@Override
public void visit(ParametersNode node) {
LOGGER.finer("Type Check visit(ParametersNode node) called on "
+ node.getName());
visitAllChildrenStandard(node);

}

@Override
public void visit(PostfixExpressionNode node) {


LOGGER.finer("Type Check visit(PostfixExpressionNode node) called on "
+ node.getName());


// visit all children
visitAllChildrenStandard(node);

// make all checks on function call
Types.checkValidFunctionCall(node);

// get the symbol for this function
FunctionSymbol funSym = SymbolTable.getSymbolForPostFixExpressionNode(node);

// set type to return type given in symbol table
// if it is a reservedWordTypeNode
if( funSym.getType() instanceof ReservedWordTypeNode) {

// if it is check inner type, set type accordingly
```

```
if(((ReservedWordTypeNode) funSym.getType()).getType() == Types.Flags.CHECK_INNER_TYPE) {
TypeNode innerType = (((DerivedTypeNode) node.getObjectOfMethod().getType()).getInnerTypeNoc
node.setType(innerType);
}

// if it is a check entire type, the return type is the typenode of the object the method is
else if(((ReservedWordTypeNode) funSym.getType()).getType() == Types.Flags.CHECK_ENTIRE_TYPE
node.setType(node.getObjectOfMethod().getType());
}

else {
node.setType(funSym.getType());
}
}

// otherwise, set the type to what is indicated in the symbol table
else {
node.setType(funSym.getType());
}
}

@Override
public void visit(PrimaryExpressionNode node) {
LOGGER.finer("Type Check visit(PrimaryExpressionNode node) called on "
+ node.getName());
visitAllChildrenStandard(node);

}

@Override
public void visit(PrimitiveTypeNode node) {
LOGGER.finer("Type Check visit(PrimitiveTypeNode node) called on "
+ node.getName());
visitAllChildrenStandard(node);

}

@Override
public void visit(ProgramNode node) {
LOGGER.finer("Type Check visit(ProgramNode node) called on "
+ node.getName());
visitAllChildrenStandard(node);

}

@Override
```

```
public void visit(RelationalExpressionNode node) {
LOGGER.finer("Type Check visit(RelationalExpressionNode node) called on "
+ node.getName());
visitAllChildrenStandard(node);
node.setType(Types.getResult(node.getOpType(), node.getLeftNode().getType(), node.getRightNo


}

@Override
public void visit(ReservedWordTypeNode node) {
LOGGER.finer("Type Check visit(ReservedWordTypeNode node) called on "
+ node.getName());
visitAllChildrenStandard(node);

}

@Override
public void visit(SectionNode node) {
LOGGER.finer("Type Check visit(SectionNode node) called on "
+ node.getName());
visitAllChildrenStandard(node);

}

@Override
public void visit(SectionTypeNode node) {
LOGGER.finer("Type Check visit(SectionTypeNode node) called on "
+ node.getName());
visitAllChildrenStandard(node);

}

@Override
public void visit(SelectionStatementNode node) {
LOGGER.finer("Type Check visit(SelectionStatementNode node) called on "
+ node.getName());
visitAllChildrenStandard(node);

}

@Override
public void visit(StatementListNode node) {
LOGGER.finer("Type Check visit(StatementListNode node) called on "
+ node.getName());
visitAllChildrenStandard(node);
```

```
}

@Override
public void visit(StatementNode node) {
LOGGER.finer("Type Check visit(StatementNode node) called on "
+ node.getName());
visitAllChildrenStandard(node);

}

@Override
public void visit(SwitchStatementNode node) {
LOGGER.finer("Type Check visit(SwitchStatementNode node) called on "
+ node.getName());
visitAllChildrenStandard(node);

}

@Override
public void visit(TypeNode node) {
LOGGER.finer("Type Check visit(TypeNode node) called on "
+ node.getName());
visitAllChildrenStandard(node);

}

@Override
/**
 * This will visit the UnOpNode's child first, and then set the type of this node
 * based on the opType and the type of the child
 */
public void visit(UnOpNode node) {
LOGGER.finer("Type Check visit(UnOpNode node) called on "
+ node.getName());

// call on children - only one child
visitAllChildrenStandard(node);

// check if they are compatible
Types.isCompatible(node.getOpType(), node.getChild().getType());

// set type of this UnOpNode
node.setType(Types.getResult(node.getOpType(), node.getChild()
.getType()));
```

```
}

}
```

B.1.5 Visitor.java

```
package back_end;

import util.ast.node.*;

/**
 * Visitor interface.
 *
 * All visitors implement this interface.
 *
 * Structured to follow the Visitor Pattern found in Design Patterns:
 * Elements of Reusable Object-Oriented Code, by Erich Gamma, Richard Helm,
 * Ralph Johnson, and John Vlissides.
 *
 * @author Jason Halpern
 *
 */
public interface Visitor {

public abstract void walk();
public abstract void visit(ArgumentsNode node);
public abstract void visit(BiOpNode node);
public abstract void visit(CatchesNode node);
public abstract void visit(ConstantNode node);
public abstract void visit(DerivedTypeNode node);
public abstract void visit(ElseIfStatementNode node);
public abstract void visit(ElseStatementNode node);
public abstract void visit(ExceptionTypeNode node);
public abstract void visit(ExpressionNode node);
public abstract void visit(FunctionNode node);
public abstract void visit(GuardingStatementNode node);
public abstract void visit(IdNode node);
public abstract void visit(IfElseStatementNode node);
public abstract void visit(IterationStatementNode node);
public abstract void visit(JumpStatementNode node);
public abstract void visit(MockExpressionNode node);
public abstract void visit(MockNode node);
public abstract void visit(Node node);
public abstract void visit(ParametersNode node);
public abstract void visit(PostfixExpressionNode node);
public abstract void visit(PrimaryExpressionNode node);
```

```
public abstract void visit(PrimitiveTypeNode node);
public abstract void visit(ProgramNode node);
public abstract void visit(RelationalExpressionNode node);
public abstract void visit(ReservedWordTypeNode node);
public abstract void visit(SectionNode node);
public abstract void visit(SectionTypeNode node);
public abstract void visit(SelectionStatementNode node);
public abstract void visit(StatementListNode node);
public abstract void visit(StatementNode node);
public abstract void visit(SwitchStatementNode node);
public abstract void visit(TypeNode node);
public abstract void visit(UnOpNode node);

}
```

B.2 front_end package

B.2.1 ConsoleLexer.java

```
package front_end;

import java.io.BufferedWriter;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

import util.ast.AbstractSyntaxTree;
import util.ast.node.BiOpNode;
import util.ast.node.ExpressionNode;
import util.ast.node.IdNode;
import util.ast.node.MockExpressionNode;
import util.ast.node.MockNode;
import util.ast.node.Node;
import util.ast.node.ProgramNode;
import util.symbol_table.SymbolTable;
import util.type.Types;

import java_cup.parser;
import java_cup.runtime.Scanner;
import java_cup.runtime.Symbol;
import front_end.*;

import java.util.Iterator;
```

```
import java.util.logging.*;

import back_end.SymbolTableVisitor;
import back_end.TypeCheckingVisitor;

/**
 * A console front-end to the Lexer class for dynamically testing the Lexer,
 * and development. Not intended to be used/accessible by/to users.
 *
 * @author Samuel Messing
 *
 */
@SuppressWarnings("unused")
public class ConsoleLexer {

private final static Logger LOGGER = Logger.getLogger(ConsoleLexer.class.getName());

/**
 * @param args
 * */
public static void main(String[] args) throws IOException {


LOGGER.info("Entering ConsoleLexer main()");
String filename = "WordCount.hog";
ProgramNode root = null;
FileReader fileReader = new FileReader(new File(filename));
    try {
        // Parser p = new Parser(new Lexer(System.in));
     Parser p = new Parser(new Lexer(fileReader));
     root = (ProgramNode) p.parse().value;

      }
      catch (FileNotFoundException e) {
       System.out.println("file not found.");
      }
      catch (Exception ex) {
       ex.printStackTrace();
      }

    AbstractSyntaxTree ast = new AbstractSyntaxTree(root);
    root.print();

    SymbolTableVisitor symTabVisitor = new SymbolTableVisitor(ast);
    symTabVisitor.walk();
    SymbolTable.printSymbolTable();
```

```
    System.out.println("\n\n\n");
    SymbolTable.print();

    root.print();

    TypeCheckingVisitor typeCheckVisitor = new TypeCheckingVisitor(ast);
    typeCheckVisitor.walk();

    root.print();

}
}
```

B.2.2 Hog.java

```
/**
 *
 */
package front_end;

import java.io.BufferedWriter;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.util.logging.Logger;

import back_end.CodeGeneratingVisitor;
import back_end.SymbolTableVisitor;
import back_end.TypeCheckingVisitor;

import util.ast.AbstractSyntaxTree;
import util.ast.node.ProgramNode;

/**
 * The Hog compiler.
 *
 * Usage:<br/>
 * <br/>
 *
 * <code>hog [--hdfs|--local] source [arg1, arg2, ...]<br/><br/></code>
 *
 * For example,<br/>
 * <br/>
```

```
 *
 * <code>hog --local WordCount.hog --input war_and_peace.txt --output
 * war_and_peace_counts.txt</code><br/>
 * <br/>
 *
 * Compiles the program <code>WordCount.hog</code>, and runs the program on a
 * local Hadoop cluster on the file <code>war_and_peace.txt</code>. This class
 * handles copying <code>war_and_peace.txt</code> to HDFS (Hadoop's Distributed
 * File System). The class also handles downloading the output to the file
 * <code>war_and_peace_counts.txt</code>.
 *
 * @author Samuel Messing
 * @author Kurry Tran
 *
 */
public class Hog {

// logger used for all nodes
protected final static Logger LOGGER = Logger
.getLogger(Hog.class.getName());
protected static boolean local = true;
protected static boolean hasInput;
protected static boolean hasOutput;
protected static FileReader source;
protected static String sourceName;
protected static String input;
protected static String output;

/**
 * @param args
 */
public static void main(String[] args) {
usage(args);

System.out.println("Loading source file " + sourceName);
System.out.println("Parsing file...");
Parser parser = new Parser(new Lexer(source));
ProgramNode root = null;

try {
root = (ProgramNode) parser.parse().value;
} catch (FileNotFoundException e) {
LOGGER.severe("Hog program " + source + " not found!");
System.exit(1);
} catch (Exception ex) {
ex.printStackTrace();
```

```
}


AbstractSyntaxTree tree = new AbstractSyntaxTree(root);
System.out.println("Generating symbol tables...");
// generate/populate symbol tables
SymbolTableVisitor symbolVisitor = new SymbolTableVisitor(tree);
symbolVisitor.walk();
System.out.println("Populating types...");
// populate/propagate/check types
TypeCheckingVisitor typeVisitor = new TypeCheckingVisitor(tree);
typeVisitor.walk();
System.out.println("Generating Java source...");
// generate source code:
CodeGeneratingVisitor codeGenerator = new CodeGeneratingVisitor(tree);
codeGenerator.walk();
System.out.println("Hog.java written...");

FileWriter fstream = null;
try {
fstream = new FileWriter("Hog.java");
BufferedWriter out = new BufferedWriter(fstream);
out.write(codeGenerator.getCode());
out.close();
} catch (IOException e) {
e.printStackTrace();
}

}

/**
 * Processes the input to this classes main function.
 *
 * @param args
 *            the passed in command-line arguments.
 */
private static void usage(String[] args) {

if (args.length < 2) {
die();
}

if (args[0].equals("--help")) {
printUsage();
}
```

```
if (args.length % 2 != 0) {
LOGGER.severe("Invalid arguments");
die();
}

local = args[0].equals("--hdfs") ? false : true;

try {
source = new FileReader(args[1]);
sourceName = args[1];
} catch (FileNotFoundException e) {
// TODO Auto-generated catch block
e.printStackTrace();
}

if (!args[1].endsWith(".java") && (!args[1].endsWith(".hog"))) {
die();
}

if (args.length == 6) {

input = args[2].equals("--input") ? args[3] : args[5];
output = args[2].equals("--output") ? args[3] : args[5];
hasInput = hasOutput = true;

} else if (args.length == 4) {

input = args[2].equals("--input") ? args[3] : "";
output = args[2].equals("--output") ? args[3] : "";
hasInput = input.equals("") ? false : true;
hasOutput = output.equals("") ? false : true;
}

}

/**
 * Prints usage information and exits.
 */
private static void die() {
printUsage();
System.exit(1);
}

/**
 * Prints the usage information for the main method in this class to
 * standard out.
```

```
 */
private static void printUsage() {
System.out.println("Hog --- a scripting MapReduce language");
System.out
.println("Usage: Hog [--hdfs|--local] source [--input file] [--output file]");
}

}
```

B.2.3 Lexer.jflex

```
package front_end;

import java_cup.runtime.*;
import util.ast.node.ExceptionTypeNode;
import util.type.Types;

/**
 * The lexer used by Parser.java to generate a token stream for a particular input file.
 *
 * @author Samuel Messing
 */

%%
/**
 * LEXICAL FUNCTIONS:
 */

%cup
%line
%column
%unicode
%class Lexer
%public

%{

/**
 * Return a new Symbol with the given token id, and with the current line and
 * column numbers.
 */
Symbol newSym(int tokenId) {
    return new Symbol(tokenId, yyline, yycolumn);
}

/**
```

```
 * Return a new Symbol with the given token id, the current line and column
 * numbers, and the given token value.  The value is used for tokens such as
 * identifiers and numbers.
 */
Symbol newSym(int tokenId, Object value) {
    return new Symbol(tokenId, yyline, yycolumn, value);
}

%}


/**
 * PATTERN DEFINITIONS:
 */
letter          = [A-Za-z]
digit           = [0-9]
alphanumeric    = {letter}|{digit}
other_id_char   = [_]
text_literal_old    = [a-zA-Z_ ]?\"(\\.|[^\\’])*\"
text_literal       = \"[^\"]*\"
identifier      = {letter}({alphanumeric}|{other_id_char})*
integer         = {digit}*
real            = {integer}\.{integer}
commentlbrace   = \#\{
commentrbrace   = \}\#
nonrightbrace   = [^}]
comment_body    = {nonrightbrace}*
comment         = ({commentlbrace}{comment_body}{commentrbrace}|\#.*\n)
newline         = [\n]+
whitespace      = [ \t]


%%
/**
 * LEXICAL RULES:
 */
and             { return newSym(sym.AND); }
else            { return newSym(sym.ELSE); }
elseif          { return newSym(sym.ELSEIF); }
if              { return newSym(sym.IF); }
or              { return newSym(sym.OR); }
foreach         { return newSym(sym.FOREACH); }
for             { return newSym(sym.FOR); }
while           { return newSym(sym.WHILE); }
in              { return newSym(sym.IN); }
text            { return newSym(sym.TEXT); }
```

```
bool            { return newSym(sym.BOOL); }
int             { return newSym(sym.INT); }
real            { return newSym(sym.REAL); }
list            { return newSym(sym.LIST); }
set             { return newSym(sym.SET); }
iter            { return newSym(sym.ITER); }
void            { return newSym(sym.VOID); }
not             { return newSym(sym.NOT); }
switch          { return newSym(sym.SWITCH); }
case            { return newSym(sym.CASE); }
default         { return newSym(sym.DEFAULT); }
continue        { return newSym(sym.CONTINUE); }
return          { return newSym(sym.RETURN); }
iter            { return newSym(sym.ITER); }
try             { return newSym(sym.TRY); }
catch           { return newSym(sym.CATCH); }
finally         { return newSym(sym.FINALLY); }
@Functions      { return newSym(sym.FUNCTIONS); }
@Map            { return newSym(sym.MAP); }
@Reduce         { return newSym(sym.REDUCE); }
@Main           { return newSym(sym.MAIN); }
"*"             { return newSym(sym.TIMES); }
"+"             { return newSym(sym.PLUS); }
"-"             { return newSym(sym.MINUS); }
"/"             { return newSym(sym.DIVIDE); }
"%"             { return newSym(sym.MOD); }
";"             { return newSym(sym.SEMICOL); }
","             { return newSym(sym.COMMA); }
"("             { return newSym(sym.L_PAREN); }
")"             { return newSym(sym.R_PAREN); }
"{"             { return newSym(sym.L_BRACE); }
"}"             { return newSym(sym.R_BRACE); }
"["             { return newSym(sym.L_BRKT); }
"]"             { return newSym(sym.R_BRKT); }
"->"            { return newSym(sym.ARROW); }
"--"            { return newSym(sym.DECR); }
"++"            { return newSym(sym.INCR); }
"="             { return newSym(sym.ASSIGN); }
"<"             { return newSym(sym.LESS); }
">"             { return newSym(sym.GRTR); }
"<="            { return newSym(sym.LESS_EQL); }
">="            { return newSym(sym.GRTR_EQL); }
"!="            { return newSym(sym.NOT_EQLS); }
":"             { return newSym(sym.COL); }
"=="            { return newSym(sym.DBL_EQLS); }
"."             { return newSym(sym.DOT); }
```

```
{text_literal}  { return newSym(sym.TEXT_LITERAL, new String(yytext())); }
true            { return newSym(sym.BOOL_CONST, new String("true")); }
false           { return newSym(sym.BOOL_CONST, new String("false")); }
return          { return newSym(sym.RETURN); }
FileNotFoundException      { return newSym(sym.EXCEPTION, new ExceptionTypeNode(Types.Except
FileLoadException          { return newSym(sym.EXCEPTION, new ExceptionTypeNode(Types.Except
ArrayOutOfBoundsException  { return newSym(sym.EXCEPTION, new ExceptionTypeNode(Types.Except
IncorrectArgumentException { return newSym(sym.EXCEPTION, new ExceptionTypeNode(Types.Except
TypeMismatchException      { return newSym(sym.EXCEPTION, new ExceptionTypeNode(Types.Except
NullReferenceException     { return newSym(sym.EXCEPTION, new ExceptionTypeNode(Types.Except
ArithmeticException        { return newSym(sym.EXCEPTION, new ExceptionTypeNode(Types.Except
{integer}       { return newSym(sym.INT_CONST, new String(yytext())); }
{real}          { return newSym(sym.REAL_CONST, new String(yytext())); }
{comment}       { /* For this stand-alone lexer, print out comments. */
                  System.out.println("Recognized comment: " + yytext()); }
{newline}       { /* Ignore newlines. */ }
{whitespace}    { /* Ignore whitespace. */ }
{identifier}    { return newSym(sym.ID, new String(yytext())); }
.               { System.out.println("Illegal char, '" + yytext() +
                    "' line: " + yyline + ", column: " + yychar); }
```

B.2.4 Parser.cup

```
package front_end;
import java.util.*;
import java.io.*;
import java_cup.runtime.*;
import util.ast.node.*;
import util.type.*;
import java.util.logging.*;

/**
 * Parse a source program based on the Hog language syntax.
 *
 * @author Samuel Messing
 * @author Benjamin Rapaport
 * @author Paul Tylkin
 */

action code
  {:
    Hashtable table = new Hashtable();
    protected final static Logger LOGGER = Logger.getLogger(Parser.class
            .getName());
  :};
```

```
parser code
  {:
    Lexer lexer;
    private File file;
    public Parser( File file ) {
      this();
      this.file = file;
      try {
        lexer = new Lexer( new FileReader( file ) );
      }
      catch ( IOException exception ) {
        throw new Error( "Unable to open file \"" + file + "\"" );
      }
    }

    public Parser(Lexer l) {
      super(l);
      lexer=l;
    }
  :};

scan with
  {:
    return lexer.next_token();
  :};

terminal DECR, INCR, RETURN, CONTINUE;
terminal TIMES, DIVIDE, MOD;
terminal LESS, GRTR, LESS_EQL, GRTR_EQL, DBL_EQLS, NOT_EQLS, ASSIGN;
terminal TEXT, BOOL, INT, REAL, VOID;
terminal MINUS, UMINUS, PLUS;
terminal ARROW, DOT;
terminal String TEXT_LITERAL;
terminal String ID;
terminal String INT_CONST;
terminal String REAL_CONST;
terminal String BOOL_CONST;
terminal String CASE;
terminal BREAK, DEFAULT;
terminal AND, OR, NOT;
terminal WHILE, FOR, FOREACH, IN, IF, ELSE, ELSEIF, SWITCH;
terminal FUNCTIONS, MAIN, MAP, REDUCE;
terminal L_BRACE, R_BRACE, L_BRKT, R_BRKT, L_PAREN, R_PAREN, SEMICOL, COL, COMMA;
terminal LIST, ITER, SET;
terminal TRY, CATCH, FINALLY;
terminal ExceptionTypeNode EXCEPTION;
```

```
nonterminal GuardingStatementNode GuardingStatement;
nonterminal CatchesNode Catches;
nonterminal IdNode CatchHeader;
nonterminal StatementListNode Finally;
nonterminal StatementListNode Block;
nonterminal StatementListNode ExpressionStatements;
nonterminal ExpressionNode ForExpr;
nonterminal StatementListNode ForInit;
nonterminal StatementListNode ForIncr;
nonterminal DerivedTypeNode DictType;

nonterminal ProgramNode Program;
nonterminal SectionNode Functions;
nonterminal SectionNode Main;
nonterminal SectionNode Map;
nonterminal SectionNode Reduce;
nonterminal SectionTypeNode SectionType;
nonterminal StatementNode Statement;
nonterminal ExpressionNode ExpressionStatement;
nonterminal StatementNode FunctionList;
nonterminal StatementNode IterationStatement;
nonterminal StatementNode LabeledStatement;
nonterminal SelectionStatementNode SelectionStatement;
nonterminal StatementNode DeclarationStatement;
nonterminal StatementListNode StatementList;
nonterminal ElseIfStatementNode ElseIfStatement;
nonterminal ElseStatementNode ElseStatement;
nonterminal JumpStatementNode JumpStatement;
nonterminal ExpressionNode EqualityExpression;
nonterminal ExpressionNode LogicalExpression;
nonterminal ExpressionNode LogicalTerm;
nonterminal ExpressionNode RelationalExpression;
nonterminal ExpressionNode Expression;
nonterminal ExpressionNode AdditiveExpression;
nonterminal ExpressionNode MultiplicativeExpression;
nonterminal ExpressionNode CastExpression;
nonterminal ExpressionNode UnaryExpression;
nonterminal ExpressionNode PostfixExpression;
nonterminal ExpressionNode PrimaryExpression;
nonterminal ExpressionNode Constant;
nonterminal ExpressionNode ArgumentExpressionList;
nonterminal FunctionNode Function;
nonterminal ParametersNode ParameterList;
nonterminal TypeNode Type;
nonterminal UnOpNode.OpType UnaryOperator;
```

```
nonterminal Types.Derived DerivedType;

precedence left MINUS, PLUS;
precedence right UMINUS;
precedence right ELSE;
precedence right ELSEIF;
precedence right L_PAREN;

start with Program;

Program::=
    Functions:f Map:m Reduce:r Main:n
    {:
      RESULT = new ProgramNode(f, m, r, n);
      RESULT.setNewScope();
      CUP$Parser$actions.LOGGER.fine("Functions Map Reduce Main -> Program");
    :}
  ;

Functions::=
    FUNCTIONS L_BRACE FunctionList:fl R_BRACE
    {:
      RESULT = new SectionNode(fl, SectionNode.SectionName.FUNCTIONS);
      RESULT.setNewScope();
      CUP$Parser$actions.LOGGER.fine("FUNCTIONS L_BRACE FunctionList R_BRACE -> Functions");
    :}
  |
    /* epsilon */
    {:
      CUP$Parser$actions.LOGGER.fine("epsilon -> Functions");
    :}
  ;

FunctionList::=
    Function:f
    {:
      RESULT = f;
      CUP$Parser$actions.LOGGER.fine("Function -> FunctionList");
    :}
  |
    FunctionList:fl Function:f
    {:
      RESULT = new StatementNode(fl, f);
      CUP$Parser$actions.LOGGER.fine("FunctionList Function -> FunctionList");
    :}
  ;
```

```
Function::=
    Type:t ID:i L_PAREN ParameterList:pl R_PAREN L_BRACE StatementList:sl R_BRACE
    {:
      RESULT = new FunctionNode(t, i, pl, sl);
      RESULT.setNewScope();
      CUP$Parser$actions.LOGGER.fine("Type ID L_PAREN ParameterList R_PAREN L_BRACE Statemer
    :}
  ;

ParameterList::=
    ParameterList:pl COMMA Type:t ID:i
    {:
      RESULT = new ParametersNode(t, i, pl);
      CUP$Parser$actions.LOGGER.fine("ParameterList COMMA Type ID: " + i + " -> ParameterLis
    :}
  |
    Type:t ID:i
    {:
      RESULT = new ParametersNode(t, i);
      CUP$Parser$actions.LOGGER.fine("Type ID: " + i + " -> ParameterList");
    :}
  |
    /* epsilon */
  ;

Map::=
    MAP SectionType:st L_BRACE StatementList:sl R_BRACE
    {:
      RESULT = new SectionNode(st, sl, SectionNode.SectionName.MAP);
      RESULT.setNewScope();
      CUP$Parser$actions.LOGGER.fine("MAP SectionType L_BRACE StatementList R_BRACE -> Map")
    :}
  ;

Reduce::=
    REDUCE SectionType:st L_BRACE StatementList:sl R_BRACE
    {:
      RESULT = new SectionNode(st, sl, SectionNode.SectionName.REDUCE);
      RESULT.setNewScope();
      CUP$Parser$actions.LOGGER.fine("REDUCE SectionType L_BRACE StatementList R_BRACE -> Re
    :}
  ;

SectionType::=
    L_PAREN Type:t1 ID:i1 COMMA Type:t2 ID:i2 R_PAREN ARROW L_PAREN Type:t3 COMMA Type:t4 R_
```

```
    {:
     RESULT = new SectionTypeNode(new IdNode(t1, i1), new IdNode(t2, i2), t3, t4);
     CUP$Parser$actions.LOGGER.fine("L_PAREN Type ID COMMA Type ID R_PAREN ARROW L_PAREN Typ
    :}
  ;

Main::=
    MAIN L_BRACE StatementList:sl R_BRACE
    {:
      RESULT = new SectionNode(sl, SectionNode.SectionName.MAIN);
      RESULT.setNewScope();
      CUP$Parser$actions.LOGGER.fine("MAIN L_BRACE StatementList R_BRACE -> Main");
    :}
  ;

StatementList::=
    Statement:s
    {:
      RESULT = s;
      CUP$Parser$actions.LOGGER.fine("Statement -> StatementList");
    :}
  |
    StatementList:sl Statement:s
    {:
      RESULT = new StatementListNode(sl, s);
      CUP$Parser$actions.LOGGER.fine("StatementList Statement -> StatementList");
    :}
  ;

Statement::=
    ExpressionStatement:es
    {:
      if (es != null) {
        es.setEndOfLine();
      }
      RESULT = es;
      CUP$Parser$actions.LOGGER.fine("ExpressionStatement -> Statement");
    :}
  |
   SelectionStatement:ss
    {:
      ss.setEndOfLine();
      RESULT = ss;
      CUP$Parser$actions.LOGGER.fine("SelectionStatement -> Statement");
    :}
  |
```

```
    IterationStatement:is
    {:
      is.setEndOfLine();
      RESULT = is;
      CUP$Parser$actions.LOGGER.fine("IterationStatement -> Statement");
    :}
  |
    LabeledStatement:ls
    {:
     ls.setEndOfLine();
     RESULT = ls;
     CUP$Parser$actions.LOGGER.fine("LabeledStatement -> Statement");
    :}
  |
    JumpStatement:js
    {:
      js.setEndOfLine();
      RESULT = js;
      CUP$Parser$actions.LOGGER.fine("JumpStatement -> Statement");
    :}
  | DeclarationStatement:ds
    {:
      ds.setEndOfLine();
      RESULT = ds;
      CUP$Parser$actions.LOGGER.fine("DeclarationStatement -> Statement");
    :}
  | GuardingStatement:gs
    {:
      gs.setEndOfLine();
      RESULT = gs;
      CUP$Parser$actions.LOGGER.fine("GuardingStatement -> Statement");
    :}
  | Block:b
    {:
      //RESULT = b;
      CUP$Parser$actions.LOGGER.fine("Block -> Statement");
    :}
  ;

GuardingStatement::=
    TRY Block:b Finally:f
    {:
      RESULT = new GuardingStatementNode(b, f);
      CUP$Parser$actions.LOGGER.fine("TRY Block Finally -> GuardingStatment");
    :}
   |
```

```
    TRY Block:b Catches:c
    {:
      RESULT = new GuardingStatementNode(b, c);
      CUP$Parser$actions.LOGGER.fine("TRY Block Catches -> GuardingStatment");
    :}
   |
    TRY Block:b Catches:c Finally:f
    {:
      RESULT = new GuardingStatementNode(b, c, f);
      CUP$Parser$actions.LOGGER.fine("TRY Block Catches Finally -> GuardingStatment");
    :}
;

Block::=
    L_BRACE StatementList:sl R_BRACE
    {:
      RESULT = sl;
      RESULT.setNewScope();
      CUP$Parser$actions.LOGGER.fine("L_BRACE StatementList R_BRACE -> Block");
    :}
  |
    L_BRACE R_BRACE
    {:
      CUP$Parser$actions.LOGGER.fine("L_BRACE R_BRACE -> Block");
    :}
;

Finally::=
    FINALLY Block:b
    {:
      RESULT = b;
      CUP$Parser$actions.LOGGER.fine("FINALLY Block -> Finally");
    :}
;

Catches::=
    CatchHeader:c Block:b
    {:
      RESULT = new CatchesNode(c, b);
      RESULT.setNewScope();
      CUP$Parser$actions.LOGGER.fine("CatchHeader Block -> Catch");
    :}
   |
    Catches:cs CatchHeader:c Block:b
    {:
      RESULT = new CatchesNode(c, b, cs);
```

```
      RESULT.setNewScope();
      CUP$Parser$actions.LOGGER.fine("Catches CatchHeader Block -> Catches");
    :}
    ;

CatchHeader::=
   CATCH L_PAREN EXCEPTION:ex ID:i R_PAREN
   {:
      IdNode catchNode = new IdNode(ex, i);
      catchNode.setDeclaration();
      RESULT = catchNode;
      CUP$Parser$actions.LOGGER.fine("CATCH L_PAREN Type ID R_PAREN -> CatchHeader");
   :}
   ;

DeclarationStatement::=
    Type:t ID:i
    {:
      IdNode declaration = new IdNode(t, i);
      declaration.setDeclaration();
      RESULT = declaration;
      CUP$Parser$actions.LOGGER.fine("Type ID -> DeclarationStatement");
    :}
  |
    Type:t ID:i ASSIGN Expression:e
    {:
      IdNode declaration = new IdNode(t, i);
      declaration.setDeclaration();
      RESULT = new BiOpNode(BiOpNode.OpType.ASSIGN, declaration, e);
      CUP$Parser$actions.LOGGER.fine("Type ID ASSIGN Expression -> DeclarationStatement");
    :}
  ;

JumpStatement::=
    CONTINUE
    {:
      RESULT = new JumpStatementNode(JumpStatementNode.JumpType.CONTINUE);
      CUP$Parser$actions.LOGGER.fine("CONTINUE -> JumpStatement");
    :}
  |
    BREAK
    {:
      RESULT = new JumpStatementNode(JumpStatementNode.JumpType.BREAK);
      CUP$Parser$actions.LOGGER.fine("BREAK -> JumpStatement");
    :}
  |
```

```
    RETURN ExpressionStatement:es
    {:
      RESULT = new JumpStatementNode(JumpStatementNode.JumpType.RETURN, es);
      CUP$Parser$actions.LOGGER.fine("RETURN Expression -> JumpStatement");
    :}
  ;

ExpressionStatement::=
    SEMICOL
    {:
      CUP$Parser$actions.LOGGER.fine("SEMICOL -> ExpressionStatement: NO RESULT STATEMENT");
    :}
  |
    Expression:e SEMICOL
    {:
      RESULT = e;
      CUP$Parser$actions.LOGGER.fine("Expression SEMICOL -> ExpressionStatement");
    :}
  ;

Expression::=
    LogicalExpression:le
    {:
     RESULT = le;
     CUP$Parser$actions.LOGGER.fine("LogicalExpression -> Expression");
    :}
  |
    UnaryExpression:ue ASSIGN Expression:e
    {:
      RESULT = new BiOpNode(BiOpNode.OpType.ASSIGN, ue, e);
      CUP$Parser$actions.LOGGER.fine("UnaryExpression ASSIGN Expression -> Expression");
    :}
  ;

LogicalExpression::=
    LogicalExpression:le OR LogicalTerm:lt
    {:
      RESULT = new BiOpNode(BiOpNode.OpType.OR, le, lt);
      CUP$Parser$actions.LOGGER.fine("LogicalExpression OR LogicalTerm -> LogicalExpression'
    :}
  |
    LogicalTerm:l
    {:
      RESULT = l;
      CUP$Parser$actions.LOGGER.fine("LogicalTerm -> LogicalExpression");
    :}
```

```
  ;

LogicalTerm::=
    LogicalTerm:l AND EqualityExpression:e
    {:
      RESULT = new BiOpNode(BiOpNode.OpType.AND, l, e);
      CUP$Parser$actions.LOGGER.fine("LogicalTerm AND EqualityExpression -> LogicalTerm");
    :}
  |
    EqualityExpression:e
    {:
      RESULT = e;
      CUP$Parser$actions.LOGGER.fine("EqualityExpression -> LogicalTerm");
    :}
  ;

EqualityExpression::=
    RelationalExpression:r
    {:
      RESULT = r;
      CUP$Parser$actions.LOGGER.fine("RelationalExpression -> EqualityExpression");
    :}
  |
    EqualityExpression:e DBL_EQLS RelationalExpression:r
    {:
      RESULT = new BiOpNode(BiOpNode.OpType.DBL_EQLS, e, r);
      CUP$Parser$actions.LOGGER.fine("EqualityExpression DBL_EQLS RelationalExpression -> Eq
    :}
  |
    EqualityExpression:e NOT_EQLS RelationalExpression:r
    {:
      RESULT = new BiOpNode(BiOpNode.OpType.NOT_EQLS, e, r);
      CUP$Parser$actions.LOGGER.fine("EqualityExpression NOT_EQLS RelationalExpression -> Eq
    :}
  ;

RelationalExpression::=
    AdditiveExpression:a
    {:
      RESULT = a;
      CUP$Parser$actions.LOGGER.fine("AdditiveExpression -> RelationalExpression");
    :}
  |
    RelationalExpression:r LESS AdditiveExpression:a
    {:
      RESULT = new BiOpNode(BiOpNode.OpType.LESS, r, a);
```

```
      CUP$Parser$actions.LOGGER.fine("RelationalExpression LESS AdditiveExpression -> Relati
    :}
  |
    RelationalExpression:r GRTR AdditiveExpression:a
    {:
      RESULT = new BiOpNode(BiOpNode.OpType.GRTR, r, a);
      CUP$Parser$actions.LOGGER.fine("RelationalExpression GRTR AdditiveExpression -> Relati
    :}
  |
    RelationalExpression:r LESS_EQL AdditiveExpression:a
    {:
      RESULT = new BiOpNode(BiOpNode.OpType.LESS_EQL, r, a);
      CUP$Parser$actions.LOGGER.fine("RelationalExpression LESS_EQL AdditiveExpression -> Re
    :}
  |
    RelationalExpression:r GRTR_EQL AdditiveExpression:a
    {:
      RESULT = new BiOpNode(BiOpNode.OpType.GRTR_EQL, r, a);
      CUP$Parser$actions.LOGGER.fine("RelationalExpression GRTR_EQL AdditiveExpression -> Re
    :}
  ;

AdditiveExpression::=
    MultiplicativeExpression:m
    {:
      RESULT = m;
      CUP$Parser$actions.LOGGER.fine("MultiplicativeExpreesion -> AdditiveExpression");
    :}
  |
    AdditiveExpression:a PLUS MultiplicativeExpression:m
    {:
      RESULT = new BiOpNode(BiOpNode.OpType.PLUS, a, m);
      CUP$Parser$actions.LOGGER.fine("AdditiveExpression PLUS MultiplicativeExpression -> Ad
    :}
  |
    AdditiveExpression:a MINUS MultiplicativeExpression:m
    {:
      RESULT = new BiOpNode(BiOpNode.OpType.MINUS, a, m);
      CUP$Parser$actions.LOGGER.fine("AdditiveExpression MINUS MultiplicativeExpression -> A
    :}
  ;

MultiplicativeExpression::=
    CastExpression:c
    {:
      RESULT = c;
```

```
      CUP$Parser$actions.LOGGER.fine("CastExpression -> MultiplicativeExpression");
    :}
  |
    MultiplicativeExpression:m TIMES CastExpression:c
    {:
      RESULT = new BiOpNode(BiOpNode.OpType.TIMES, m, c);
      CUP$Parser$actions.LOGGER.fine("MultipliativeExpression TIMES CastExpression -> Multip
    :}
  |
    MultiplicativeExpression:m DIVIDE CastExpression:c
    {:
      RESULT = new BiOpNode(BiOpNode.OpType.DIVIDE, m, c);
      CUP$Parser$actions.LOGGER.fine("MultipliativeExpression DIVIDE CastExpression -> Multi
    :}
  |
    MultiplicativeExpression:m MOD CastExpression:c
    {:
      RESULT = new BiOpNode(BiOpNode.OpType.MOD, m, c);
      CUP$Parser$actions.LOGGER.fine("MultipliativeExpression MOD CastExpression -> Multipli
    :}
  ;

CastExpression::=
    UnaryExpression:u
    {:
      RESULT = u;
      CUP$Parser$actions.LOGGER.fine("UnaryExpression -> CastExpression");
    :}
  |
    L_PAREN Type:t R_PAREN CastExpression:c
    {:
      RESULT = new UnOpNode(t, UnOpNode.OpType.CAST, c);
      CUP$Parser$actions.LOGGER.fine("L_PAREN Type R_PAREN CastExpression -> CastExpression'
    :}
  ;

UnaryExpression::=
    UnaryOperator:u CastExpression:c
    {:
      RESULT = new UnOpNode(u, c);
      CUP$Parser$actions.LOGGER.fine("UnaryOperator CastExpression -> UnaryExpression");
    :}
  |
    PostfixExpression:p
    {:
      RESULT = p;
```

```
      CUP$Parser$actions.LOGGER.fine("PostfixExpression -> UnaryExpression");
    :}
  ;

UnaryOperator::=
    MINUS
    {:
      RESULT = UnOpNode.OpType.UMINUS;
      CUP$Parser$actions.LOGGER.fine("UMINUS -> UnaryOperator");
    :}
    %prec UMINUS
  |
    NOT
    {:
      RESULT = UnOpNode.OpType.NOT;
      CUP$Parser$actions.LOGGER.fine("NOT -> UnaryOperator");
    :}
  ;

PostfixExpression::=
    PrimaryExpression:p
    {:
      RESULT = p;
      CUP$Parser$actions.LOGGER.fine("PrimaryExpression -> PostFixExpression");
    :}
  |
    ID:objectName DOT ID:methodName
    {:
      // object method call no params
      IdNode obj = new IdNode(objectName);
      IdNode method = new IdNode(methodName);
      RESULT = new PostfixExpressionNode(obj, method);
      CUP$Parser$actions.LOGGER.fine("ID: " + objectName + " DOT ID: " + methodName + " -> P
    :}
  |
    ID:objectName DOT ID:methodName L_PAREN ArgumentExpressionList:a R_PAREN
    {:
      // object method call with params
      IdNode object = new IdNode(objectName);
      IdNode method = new IdNode(methodName);
      // if argument list is empty, this is a method with no params
      if( a == null ) {
        RESULT = new PostfixExpressionNode(object, method);
      } else {
        RESULT = new PostfixExpressionNode(PostfixExpressionNode.PostfixType.METHOD_WITH_PAR
      }
```

```
      CUP$Parser$actions.LOGGER.fine("ID " + objectName + " DOT ID: " + methodName + "L_PARE
    :}
  |

    ID:i L_PAREN ArgumentExpressionList:a R_PAREN
    {:
      //function call
      IdNode functionName = new IdNode(i);
      RESULT = new PostfixExpressionNode(PostfixExpressionNode.PostfixType.FUNCTION_CALL, fu
      CUP$Parser$actions.LOGGER.fine("PostfixExpression L_PAREN ArgumentExpressionList R_PAR
    :}
  |
    PostfixExpression:p INCR
    {:
      RESULT = new UnOpNode(UnOpNode.OpType.INCR, p);
      CUP$Parser$actions.LOGGER.fine("PostfixExpression INCR -> PostFixExpression");
    :}
  |
    PostfixExpression:p DECR
    {:
      RESULT = new UnOpNode(UnOpNode.OpType.DECR, p);
      CUP$Parser$actions.LOGGER.fine("PostfixExpression DECR -> PostFixExpression");
    :}
  ;

ArgumentExpressionList::=
    Expression:e
    {:
      RESULT = new ArgumentsNode(null,e);
      CUP$Parser$actions.LOGGER.fine("Expression -> ArgumentExpressionList");
    :}
  |
    ArgumentExpressionList:a COMMA Expression:e
    {:
      RESULT = new ArgumentsNode(a,e);
      CUP$Parser$actions.LOGGER.fine("ArgumentExpressionList COMMA Expression -> ArgumentExp
    :}
  |
    /* epsilon */
    {:
      CUP$Parser$actions.LOGGER.fine("epsilon -> ArgumentExpressionList");
    :}

 ;

PrimaryExpression::=
```

```
    ID:i
    {:
      RESULT = new IdNode(i);
      CUP$Parser$actions.LOGGER.fine("ID: " + i + " -> PrimaryExpression");
    :}
  |
    Constant:c
    {:
      RESULT = c;
      CUP$Parser$actions.LOGGER.fine("Constant -> PrimaryExpression");
    :}
  |
    L_PAREN Expression:e R_PAREN
    {:
      RESULT = e;
      CUP$Parser$actions.LOGGER.fine("L_PAREN Expression R_PAREN -> PrimaryExpression");
    :}
  ;

Constant::=
    INT_CONST:i
    {:
      RESULT = new ConstantNode(Types.Primitive.INT, i);
      CUP$Parser$actions.LOGGER.fine("INT_CONST: " + i + " -> Constant");
    :}
  |
    REAL_CONST:d
    {:
      RESULT = new ConstantNode(Types.Primitive.REAL, d);
      CUP$Parser$actions.LOGGER.fine("REAL_CONST: " + d + " -> Constant");
    :}
  |
    BOOL_CONST:b
    {:
      RESULT = new ConstantNode(Types.Primitive.BOOL, b);
      CUP$Parser$actions.LOGGER.fine("BOOL_CONST: " + b + " -> Constant");
    :}
  |
    TEXT_LITERAL:t
    {:
      RESULT = new ConstantNode(Types.Primitive.TEXT, t);
      CUP$Parser$actions.LOGGER.fine("TEXT_LITERAL: " + t + " -> Constant");
    :}
  ;

SelectionStatement::=
```

```
    IF Expression:e Block:b ElseIfStatement:elif ElseStatement:el
    {:
      RESULT = new IfElseStatementNode(e, b, elif, el);
      RESULT.setNewScope();
      CUP$Parser$actions.LOGGER.fine("IF Expression Block " +
                                     "ElseIfStatement ElseStatment -> SelectionStatement");
    :}
  |
    SWITCH Expression:e L_BRACE StatementList:s R_BRACE
    {:
    // NOT THINKING ABOUT SCOPE. TOO TIRED.
      RESULT = new SwitchStatementNode(e, s);
      RESULT.setNewScope();
      CUP$Parser$actions.LOGGER.fine("SWITCH Expression L_BRACE " +
                                     "StatementList R_BRACE -> SelectionStatement");
    :}
  ;

ElseIfStatement::=
    ELSEIF Expression:e Block:b ElseIfStatement:elif
    {:
      RESULT = new ElseIfStatementNode(e, b, elif);
      RESULT.setNewScope();
      CUP$Parser$actions.LOGGER.fine("ELSEIF L_PAREN Expression R_PAREN ElseIfStatement -> '
                                     "ElseIfStatement");
    :}
  |
    /* epsilon */
    {:
      CUP$Parser$actions.LOGGER.fine("epsilon -> ElseIfStatement");
    :}
  ;

ElseStatement::=
    ELSE Block:b
    {:
      RESULT = new ElseStatementNode(b);
      RESULT.setNewScope();
      CUP$Parser$actions.LOGGER.fine("Block -> ElseStatement");
    :}
  |
    /* epsilon */
    {:
      CUP$Parser$actions.LOGGER.fine("epsilon -> ElseStatement");
    :}
  ;
```

```
IterationStatement::=
    WHILE L_PAREN Expression:e R_PAREN Block:b
    {:
      RESULT = new IterationStatementNode(e,b);
      RESULT.setNewScope();
      CUP$Parser$actions.LOGGER.fine("WHILE L_PAREN Expression R_PAREN L_BRACE " +
                                     "StatementList R_BRACE -> IterationStatement");
    :}
  |
    FOR L_PAREN ForInit:init ForExpr:test ForIncr:incr R_PAREN Block:b
    {:
      RESULT = new IterationStatementNode(init, test, incr, b);
      RESULT.setNewScope();
      CUP$Parser$actions.LOGGER.fine("FOR ForInit ForExpr ForInc L_BRACE StatementList " +
                                     "R_BRACE -> IterationStatement");
    :}
  |
    FOR L_PAREN ForInit ForExpr R_PAREN Block:b
    {:
      //RESULT = new IterationStatementNode(e1,e2,e3,b);
      //RESULT.setNewScope();
      CUP$Parser$actions.LOGGER.fine("FOR L_BRACEN ForInit ForExpr R_PAREN L_BRACE " +
                                     "StatementList R_BRACE -> IterationStatement");
    :}
  |
    FOREACH Type:t ID:i IN Expression:e2 Block:b
    {:
      IdNode idNode = new IdNode(t, i);
      RESULT = new IterationStatementNode(idNode,e2,b);
      RESULT.setNewScope();
      CUP$Parser$actions.LOGGER.fine("FOREACH Expression IN Expression L_BRACE StatementList
                                     "R_BRACE -> IterationStatement");
    :}
  ;

ForInit::=
    ExpressionStatements:es
    {:
      RESULT = es;
      CUP$Parser$actions.LOGGER.fine("ExpressionStatements -> ForInit");
    :}
  |
    DeclarationStatement:ds SEMICOL
    {:
      RESULT = ds;
```

```
      CUP$Parser$actions.LOGGER.fine("DeclarationStatement SEMICOL -> ForInit");
    :}
  ;

ForExpr::=
    ExpressionStatement:es
    {:
      RESULT = es;
      CUP$Parser$actions.LOGGER.fine("ExpressionStatement -> ForExpr");
    :}
  ;

ForIncr::=
    ExpressionStatements:es
    {:
      RESULT = es;
      CUP$Parser$actions.LOGGER.fine("ExpressionStatements -> ForIncr");
    :}
  ;

ExpressionStatements::=
    ExpressionStatement:es
    {:
      RESULT = es;
      CUP$Parser$actions.LOGGER.fine("ExpressionStatement -> ExpressionStatements");
    :}
  |
    ExpressionStatements:e COMMA ExpressionStatement:es
    {:
      List<Node> children = new ArrayList<Node>();
      children.add(e);
      children.add(es);
      RESULT = new StatementListNode(children);
      CUP$Parser$actions.LOGGER.fine("ExpressionStatements COMMA ExpressionStatement -> " +
                                     "ExpressionStatements");
    :}
  ;

LabeledStatement::=
    CASE LogicalExpression COL Statement:s
    {:
      CUP$Parser$actions.LOGGER.fine("CASE LogicalExpression COL Statement -> LabeledStateme
    :}
  |
    DEFAULT:d COL Statement:s
    {:
```

```
      CUP$Parser$actions.LOGGER.fine("DEFAULT COL Statement");
    :}
  ;

Type::=
    VOID
    {:
      RESULT = new PrimitiveTypeNode(Types.Primitive.VOID);
      CUP$Parser$actions.LOGGER.fine("VOID -> Type");
    :}
  |
    TEXT
    {:
      RESULT = new PrimitiveTypeNode(Types.Primitive.TEXT);
      CUP$Parser$actions.LOGGER.fine("TEXT -> Type");
    :}
  |
    BOOL
    {:
      RESULT = new PrimitiveTypeNode(Types.Primitive.BOOL);
      CUP$Parser$actions.LOGGER.fine("BOOL -> Type");
    :}
  |
    INT:i
    {:
      RESULT = new PrimitiveTypeNode(Types.Primitive.INT);
      CUP$Parser$actions.LOGGER.fine("INT -> Type");
    :}
  |
    REAL
    {:
      RESULT = new PrimitiveTypeNode(Types.Primitive.REAL);
      CUP$Parser$actions.LOGGER.fine("REAL -> Type");
    :}
  |
    DerivedType:d LESS Type:t GRTR
    {:
      RESULT = new DerivedTypeNode(d, t);
      CUP$Parser$actions.LOGGER.fine("DerivedType LESS Type GRTR -> Type");
    :}
  ;

DerivedType::=
    LIST
    {:
      RESULT = Types.Derived.LIST;
```

```
      CUP$Parser$actions.LOGGER.fine("LIST -> DerivedType");
    :}
  |
    ITER
    {:
      RESULT = Types.Derived.ITER;
      CUP$Parser$actions.LOGGER.fine("ITER -> DerivedType");
    :}
  |
    SET
    {:
      RESULT = Types.Derived.SET;
      CUP$Parser$actions.LOGGER.fine("SET -> DerivedType");
    :}
  ;
```

B.3 test package

B.3.1 AbstractSyntaxTreeTester.java

```
package test;

import static org.junit.Assert.*;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;

import back_end.TypeCheckingVisitor;
import util.ast.AbstractSyntaxTree;
import util.ast.node.BiOpNode;
import util.ast.node.ConstantNode;
import util.ast.node.ExpressionNode;
import util.ast.node.IdNode;
import util.ast.node.MockExpressionNode;
import util.ast.node.MockNode;
import util.ast.node.Node;
import util.ast.node.RelationalExpressionNode;
import util.ast.node.UnOpNode;
```

```
import util.type.Types;


/**
 *
 * Tests for the functionality provided by the AbstractSyntaxTree class.
 *
 * @author Samuel Messing
 * @author Jason Halpern
 *
 */
public class AbstractSyntaxTreeTester {

private ExpressionNode A;
private ExpressionNode B;
private ExpressionNode C;
private ExpressionNode D;
private ExpressionNode E;
private ExpressionNode F;
private ExpressionNode G;
private ExpressionNode H;
private ExpressionNode I;
private ExpressionNode J;
private AbstractSyntaxTree tree;

@BeforeClass
public static void setUpClass() throws Exception {
}

@AfterClass
public static void tearDownClass() throws Exception {
}

@Before
public void setUp() {
// A is root of tree, with children B, C
// B has children D, E
// C has children F, G
D = new MockExpressionNode();
E = new MockExpressionNode();
F = new MockExpressionNode();
G = new MockExpressionNode();
J = new MockExpressionNode();
H = new UnOpNode(UnOpNode.OpType.UMINUS, G);
I = new RelationalExpressionNode(BiOpNode.OpType.LESS, E, J);
B = new BiOpNode(BiOpNode.OpType.PLUS, D, I);
```

```
C = new BiOpNode(BiOpNode.OpType.MINUS, F, H);
A = new BiOpNode(BiOpNode.OpType.TIMES, B, C);
tree = new AbstractSyntaxTree(A);
}

@After
public void tearDown() {
}

/**
 * Tests for correct ordering of nodes when generating preOrderTraversal
 * iterators.
 */
@Test
public void preOrderTraversalTest() {

List<Node> correctPreOrderTraversal = new ArrayList<Node>();

correctPreOrderTraversal.add(A);
correctPreOrderTraversal.add(B);
correctPreOrderTraversal.add(D);
correctPreOrderTraversal.add(I);
correctPreOrderTraversal.add(E);
correctPreOrderTraversal.add(J);
correctPreOrderTraversal.add(C);
correctPreOrderTraversal.add(F);
correctPreOrderTraversal.add(H);
correctPreOrderTraversal.add(G);

Iterator<Node> preOrderTraversal = tree.preOrderTraversal();

int index = 0;
while (preOrderTraversal.hasNext()) {
Node nextNode = preOrderTraversal.next();
Node correctNextNode = correctPreOrderTraversal.get(index);
assertEquals(
"It should compute the preOrderTraversal in the correct order.",
nextNode, correctNextNode);
index++;
}

}

/**
 * Tests for correct ordering of nodes when generating postOrderTraversal
 * iterators.
```

```
 */
@Test
public void postOrderTraversalTest() {

List<Node> correctPostOrderTraversal = new ArrayList<Node>();

correctPostOrderTraversal.add(D);
correctPostOrderTraversal.add(E);
correctPostOrderTraversal.add(J);
correctPostOrderTraversal.add(I);
correctPostOrderTraversal.add(B);
correctPostOrderTraversal.add(F);
correctPostOrderTraversal.add(G);
correctPostOrderTraversal.add(H);
correctPostOrderTraversal.add(C);
correctPostOrderTraversal.add(A);

Iterator<Node> postOrderTraversal = tree.postOrderTraversal();

int index = 0;
while (postOrderTraversal.hasNext()) {
Node nextNode = postOrderTraversal.next();
Node correctNextNode = correctPostOrderTraversal.get(index);
assertEquals(
"It should compute the postOrderTraversal in the correct order.",
nextNode, correctNextNode);
index++;
}

}




}
```

B.3.2 CodeGeneratingTester.java

```
package test;

import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.StringReader;
import java.util.Iterator;
```

```
import org.junit.Test;

import util.ast.AbstractSyntaxTree;
import util.ast.node.IdNode;
import util.ast.node.Node;
import util.ast.node.ProgramNode;
import util.symbol_table.Symbol;
import util.symbol_table.SymbolTable;
import back_end.CodeGeneratingVisitor;
import back_end.SymbolTableVisitor;
import back_end.TypeCheckingVisitor;
import front_end.Lexer;
import front_end.Parser;

/**
 *
 * @author Jason Halpern
 *
 */
public class CodeGeneratingTester {


@Test
public void codeGeneratingTest() {
String filename = "src/test/WordCount.hog";
ProgramNode root = null;
FileReader fileReader;
    try {
     fileReader = new FileReader(new File(filename));
        // Parser p = new Parser(new Lexer(System.in));
     Parser p = new Parser(new Lexer(fileReader));
     root = (ProgramNode) p.parse().value;

      }
      catch (FileNotFoundException e) {
       System.out.println("file not found.");
      }
      catch (Exception ex) {
       ex.printStackTrace();
      }

    AbstractSyntaxTree ast = new AbstractSyntaxTree(root);
    root.print();

    SymbolTableVisitor symTabVisitor = new SymbolTableVisitor(ast);
```

```
    symTabVisitor.walk();
    SymbolTable.printSymbolTable();
    System.out.println("\n\n\n");
    SymbolTable.print();

    TypeCheckingVisitor typeCheckVisitor = new TypeCheckingVisitor(ast);
    typeCheckVisitor.walk();

    CodeGeneratingVisitor codeGenVisitor = new CodeGeneratingVisitor(ast);
    codeGenVisitor.walk();
    String temp = codeGenVisitor.getCode();

System.out.println(temp);

}

@Test
public void codeGeneratingTestTwo() {
String filename = "src/test/Factorial.hog";
ProgramNode root = null;
FileReader fileReader;
    try {
     fileReader = new FileReader(new File(filename));
        // Parser p = new Parser(new Lexer(System.in));
     Parser p = new Parser(new Lexer(fileReader));
     root = (ProgramNode) p.parse().value;

      }
      catch (FileNotFoundException e) {
       System.out.println("file not found.");
      }
      catch (Exception ex) {
       ex.printStackTrace();
      }

    AbstractSyntaxTree ast = new AbstractSyntaxTree(root);
    root.print();

    SymbolTableVisitor symTabVisitor = new SymbolTableVisitor(ast);
    symTabVisitor.walk();
    SymbolTable.printSymbolTable();
    System.out.println("\n\n\n");
    SymbolTable.print();

    TypeCheckingVisitor typeCheckVisitor = new TypeCheckingVisitor(ast);
    typeCheckVisitor.walk();
```

```
    CodeGeneratingVisitor codeGenVisitor = new CodeGeneratingVisitor(ast);
    codeGenVisitor.walk();
    String temp = codeGenVisitor.getCode();

System.out.println(temp);

}

@Test
public void codeGeneratingTestThree() {
String filename = "src/test/MergeSort.hog";
ProgramNode root = null;
FileReader fileReader;
    try {
     fileReader = new FileReader(new File(filename));
        // Parser p = new Parser(new Lexer(System.in));
     Parser p = new Parser(new Lexer(fileReader));
     root = (ProgramNode) p.parse().value;

      }
      catch (FileNotFoundException e) {
       System.out.println("file not found.");
      }
      catch (Exception ex) {
       ex.printStackTrace();
      }

    AbstractSyntaxTree ast = new AbstractSyntaxTree(root);
    root.print();

    SymbolTableVisitor symTabVisitor = new SymbolTableVisitor(ast);
    symTabVisitor.walk();
    SymbolTable.printSymbolTable();
    System.out.println("\n\n\n");
    SymbolTable.print();

    TypeCheckingVisitor typeCheckVisitor = new TypeCheckingVisitor(ast);
    typeCheckVisitor.walk();

    CodeGeneratingVisitor codeGenVisitor = new CodeGeneratingVisitor(ast);
    codeGenVisitor.walk();
    String temp = codeGenVisitor.getCode();

System.out.println(temp);
```

```
}
}
```

B.3.3 LexerTester.java

```
package test;

import static org.junit.Assert.*;

import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;

import front_end.Lexer;
import front_end.sym;

import java.io.IOException;
import java.io.StringReader;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

import java_cup.runtime.Symbol;

/**
 * Tests Lexer's performance on decomposing different inputs into the correct
 * sequence of tokens.
 *
 * @author Jason Halpern
 * @author Samuel Messing
 *
 */
public class LexerTester {

@BeforeClass
public static void setUpClass() throws Exception {
}

@AfterClass
public static void tearDownClass() throws Exception {
}

@Before
```

```
public void setUp() {

}

@After
public void tearDown() {
}

/**
 * Tests for correct parsing of addition operator and its operands.
 *
 * Specifically, ensures that Lexer produces token streams of ID PLUS ID for
 * strings like "a + b" and INT_CONST PLUS INT_CONST for strings like
 * "1 + 2".
 *
 * @throws IOException
 */
@Test
public void additionSymbolTest1() throws IOException {

String text = "2 + 5";
StringReader stringReader = new StringReader(text);
Lexer lexer = new Lexer(stringReader);
List<Integer> tokenList = new ArrayList<Integer>();
Symbol token = lexer.next_token();

while (token.sym != sym.EOF) {
tokenList.add(token.sym);
token = lexer.next_token();
}

assertEquals(
"It should produce 3 tokens for the string '" + text + "'", 3,
tokenList.size());
assertEquals("The first token should be an INT_CONST", sym.INT_CONST,
tokenList.get(0).intValue());
assertEquals("The second token should be a PLUS", sym.PLUS, tokenList
.get(1).intValue());
assertEquals("The third token should be an INT_CONST", sym.INT_CONST,
tokenList.get(2).intValue());

}

/**
 * Tests for correct parsing of addition operator and its operands.
 *
```

```
 * Specifically, ensures that Lexer produces token streams of ID PLUS ID for
 * strings like "a + b" and INT_CONST PLUS INT_CONST for strings like
 * "1 + 2".
 *
 * @throws IOException
 */
@Test
public void additionSymbolTest2() throws IOException {

String text = "a + b";
StringReader stringReader = new StringReader(text);
Lexer lexer = new Lexer(stringReader);
List<Integer> tokenList = new ArrayList<Integer>();
Symbol token = lexer.next_token();

while (token.sym != sym.EOF) {
tokenList.add(token.sym);
token = lexer.next_token();
}

assertEquals(
"It should produce 3 tokens for the string '" + text + "'", 3,
tokenList.size());
assertEquals("The first token should be an ID", sym.ID, tokenList
.get(0).intValue());
assertEquals("The second token should be a PLUS", sym.PLUS, tokenList
.get(1).intValue());
assertEquals("The third token should be an ID", sym.ID, tokenList
.get(2).intValue());

}

/**
 * Tests for correct parsing of subtraction operator and its operands.
 *
 * Specifically, ensures that Lexer produces token streams of ID MINUS ID
 * for strings like "a - b" and INT_CONST MINUS INT_CONST for strings like
 * "1 - 2".
 *
 * @throws IOException
 */
@Test
public void subtractionSymbolTest1() throws IOException {

String text = "a - b";
StringReader stringReader = new StringReader(text);
```

```
Lexer lexer = new Lexer(stringReader);
List<Integer> tokenList = new ArrayList<Integer>();
Symbol token = lexer.next_token();

while (token.sym != sym.EOF) {
tokenList.add(token.sym);
token = lexer.next_token();
}

assertEquals(
"It should produce 3 tokens for the string '" + text + "'", 3,
tokenList.size());
assertEquals("The first token should be an ID", sym.ID, tokenList
.get(0).intValue());
assertEquals("The second token should be a MINUS", sym.MINUS, tokenList
.get(1).intValue());
assertEquals("The third token should be an ID", sym.ID, tokenList
.get(2).intValue());

}

/**
 * Tests for correct parsing of subtraction operator and its operands.
 *
 * Specifically, ensures that Lexer produces token streams of ID MINUS ID
 * for strings like "a - b" and INT_CONST MINUS INT_CONST for strings like
 * "1 - 2".
 *
 * @throws IOException
 */
@Test
public void subtractionSymbolTest2() throws IOException {

String text = "1 - 2";
StringReader stringReader = new StringReader(text);
Lexer lexer = new Lexer(stringReader);
List<Integer> tokenList = new ArrayList<Integer>();
Symbol token = lexer.next_token();

while (token.sym != sym.EOF) {
tokenList.add(token.sym);
token = lexer.next_token();
}

assertEquals(
"It should produce 3 tokens for the string '" + text + "'", 3,
```

```
tokenList.size());
assertEquals("The first token should be an INT_CONST", sym.INT_CONST,
tokenList.get(0).intValue());
assertEquals("The second token should be a MINUS", sym.MINUS, tokenList
.get(1).intValue());
assertEquals("The third token should be an INT_CONST", sym.INT_CONST,
tokenList.get(2).intValue());

}

/**
 * Tests for correct parsing of division operator and its operands.
 *
 * Specifically, ensures that Lexer produces token streams of ID DIVIDE ID
 * for strings like "a / b" and INT_CONST DIVIDE INT_CONST for strings like
 * "1 / 2".
 *
 * @throws IOException
 */
@Test
public void divisionSymbolTest1() throws IOException {

String text = "1 / 2";
StringReader stringReader = new StringReader(text);
Lexer lexer = new Lexer(stringReader);
List<Integer> tokenList = new ArrayList<Integer>();
Symbol token = lexer.next_token();

while (token.sym != sym.EOF) {
tokenList.add(token.sym);
token = lexer.next_token();
}

assertEquals(
"It should produce 3 tokens for the string '" + text + "'", 3,
tokenList.size());
assertEquals("The first token should be an INT_CONST", sym.INT_CONST,
tokenList.get(0).intValue());
assertEquals("The second token should be a DIVIDE", sym.DIVIDE,
tokenList.get(1).intValue());
assertEquals("The third token should be an INT_CONST", sym.INT_CONST,
tokenList.get(2).intValue());

}

/**
```

```
 * Tests for correct parsing of division operator and its operands.
 *
 * Specifically, ensures that Lexer produces token streams of ID DIVIDE ID
 * for strings like "a / b" and INT_CONST DIVIDE INT_CONST for strings like
 * "1 / 2".
 *
 * @throws IOException
 */
@Test
public void divisionSymbolTest2() throws IOException {

String text = "a / b";
StringReader stringReader = new StringReader(text);
Lexer lexer = new Lexer(stringReader);
List<Integer> tokenList = new ArrayList<Integer>();
Symbol token = lexer.next_token();

while (token.sym != sym.EOF) {
tokenList.add(token.sym);
token = lexer.next_token();
}

assertEquals(
"It should produce 3 tokens for the string '" + text + "'", 3,
tokenList.size());
assertEquals("The first token should be an ID", sym.ID, tokenList
.get(0).intValue());
assertEquals("The second token should be a DIVIDE", sym.DIVIDE,
tokenList.get(1).intValue());
assertEquals("The third token should be an ID", sym.ID, tokenList
.get(2).intValue());

}

/**
 * Tests for correct parsing of multiplication operator and its operands.
 *
 * Specifically, ensures that Lexer produces token streams of ID TIMES ID
 * for strings like "a * b" and INT_CONST TIMES INT_CONST for strings like
 * "1 * 2".
 *
 * @throws IOException
 */
@Test
public void multiplicationSymbolTest1() throws IOException {
```

```
String text = "1 * 2";
StringReader stringReader = new StringReader(text);
Lexer lexer = new Lexer(stringReader);
List<Integer> tokenList = new ArrayList<Integer>();
Symbol token = lexer.next_token();

while (token.sym != sym.EOF) {
tokenList.add(token.sym);
token = lexer.next_token();
}

assertEquals(
"It should produce 3 tokens for the string '" + text + "'", 3,
tokenList.size());
assertEquals("The first token should be an INT_CONST", sym.INT_CONST,
tokenList.get(0).intValue());
assertEquals("The second token should be a TIMES", sym.TIMES, tokenList
.get(1).intValue());
assertEquals("The third token should be an INT_CONST", sym.INT_CONST,
tokenList.get(2).intValue());

}

/**
 * Tests for correct parsing of multiplication operator and its operands.
 *
 * Specifically, ensures that Lexer produces token streams of ID TIMES ID
 * for strings like "a * b" and INT_CONST TIMES INT_CONST for strings like
 * "1 * 2".
 *
 * @throws IOException
 */
@Test
public void multiplicationSymbolTest2() throws IOException {

String text = "a * b";
StringReader stringReader = new StringReader(text);
Lexer lexer = new Lexer(stringReader);
List<Integer> tokenList = new ArrayList<Integer>();
Symbol token = lexer.next_token();

while (token.sym != sym.EOF) {
tokenList.add(token.sym);
token = lexer.next_token();
}
```

```
assertEquals(
"It should produce 3 tokens for the string '" + text + "'", 3,
tokenList.size());
assertEquals("The first token should be an ID", sym.ID, tokenList
.get(0).intValue());
assertEquals("The second token should be a TIMES", sym.TIMES, tokenList
.get(1).intValue());
assertEquals("The third token should be an ID", sym.ID, tokenList
.get(2).intValue());

}

/**
 * Tests for correct parsing of a parenthesized expression.
 *
 * Specifically, ensures that Lexer produces a token stream of L_PAREN ID
 * R_PAREN for strings like "(a)"
 *
 * @throws IOException
 */
@Test
public void parenthesesSymbolTest() throws IOException {

String text = "(a)";
StringReader stringReader = new StringReader(text);
Lexer lexer = new Lexer(stringReader);
List<Integer> tokenList = new ArrayList<Integer>();
Symbol token = lexer.next_token();

while (token.sym != sym.EOF) {
tokenList.add(token.sym);
token = lexer.next_token();
}

assertEquals(
"It should produce 3 tokens for the string '" + text + "'", 3,
tokenList.size());
assertEquals("The first token should be an L_PAREN", sym.L_PAREN,
tokenList.get(0).intValue());
assertEquals("The second token should be a ID", sym.ID, tokenList
.get(1).intValue());
assertEquals("The third token should be an R_PAREN", sym.R_PAREN,
tokenList.get(2).intValue());

}
```

```
/**
 * Tests for correct parsing of the postfix increment operator
 *
 * Specifically, ensures that Lexer produces a token stream of ID INCR for
 * strings like "a++"
 *
 * @throws IOException
 */
@Test
public void incrementSymbolTest() throws IOException {

String text = "a++";
StringReader stringReader = new StringReader(text);
Lexer lexer = new Lexer(stringReader);
List<Integer> tokenList = new ArrayList<Integer>();
Symbol token = lexer.next_token();

while (token.sym != sym.EOF) {
tokenList.add(token.sym);
token = lexer.next_token();
}

assertEquals(
"It should produce 2 tokens for the string '" + text + "'", 2,
tokenList.size());
assertEquals("The first token should be a ID", sym.ID, tokenList.get(0)
.intValue());
assertEquals("The second token should be a INCR", sym.INCR, tokenList
.get(1).intValue());
}

/**
 * Tests for correct parsing of the postfix decrement operator
 *
 * Specifically, ensures that Lexer produces a token stream of ID DECR for
 * strings like "a--"
 *
 * @throws IOException
 */
@Test
public void decrementSymbolTest() throws IOException {

String text = "a--";
StringReader stringReader = new StringReader(text);
Lexer lexer = new Lexer(stringReader);
List<Integer> tokenList = new ArrayList<Integer>();
```

```
Symbol token = lexer.next_token();

while (token.sym != sym.EOF) {
tokenList.add(token.sym);
token = lexer.next_token();
}

assertEquals(
"It should produce 2 tokens for the string '" + text + "'", 2,
tokenList.size());
assertEquals("The first token should be a ID", sym.ID, tokenList.get(0)
.intValue());
assertEquals("The second token should be a DECR", sym.DECR, tokenList
.get(1).intValue());
}

/**
 * Tests for correct parsing of the statements with the logical 'and'
 * operator
 *
 * Specifically, ensures that Lexer produces a token stream of BOOL_CONST
 * AND BOOL_CONST for strings like "true and false" and a token stream of ID
 * AND ID for strings like "variable and variable"
 *
 * @throws IOException
 */
@Test
public void andSymbolTest1() throws IOException {

String text = "true and false";
StringReader stringReader = new StringReader(text);
Lexer lexer = new Lexer(stringReader);
List<Integer> tokenList = new ArrayList<Integer>();
Symbol token = lexer.next_token();

while (token.sym != sym.EOF) {
tokenList.add(token.sym);
token = lexer.next_token();
}

assertEquals(
"It should produce 3 tokens for the string '" + text + "'", 3,
tokenList.size());
assertEquals("The first token should be a BOOL_CONST", sym.BOOL_CONST,
tokenList.get(0).intValue());
assertEquals("The second token should be a AND", sym.AND, tokenList
```

```
.get(1).intValue());
assertEquals("The third token should be a BOOL_CONST", sym.BOOL_CONST,
tokenList.get(2).intValue());
}

/**
 * Tests for correct parsing of the statements with the logical 'and'
 * operator
 *
 * Specifically, ensures that Lexer produces a token stream of BOOL_CONST
 * AND BOOL_CONST for strings like "true and false" and a token stream of ID
 * AND ID for strings like "variable and variable"
 *
 * @throws IOException
 */
@Test
public void andSymbolTest2() throws IOException {

String text = "variable and variable";
StringReader stringReader = new StringReader(text);
Lexer lexer = new Lexer(stringReader);
List<Integer> tokenList = new ArrayList<Integer>();
Symbol token = lexer.next_token();

while (token.sym != sym.EOF) {
tokenList.add(token.sym);
token = lexer.next_token();
}

assertEquals(
"It should produce 3 tokens for the string '" + text + "'", 3,
tokenList.size());
assertEquals("The first token should be a ID", sym.ID, tokenList.get(0)
.intValue());
assertEquals("The second token should be a AND", sym.AND, tokenList
.get(1).intValue());
assertEquals("The third token should be a ID", sym.ID, tokenList.get(2)
.intValue());
}

/**
 * Tests for correct parsing of the statements with the logical 'or'
 * operator
 *
 * Specifically, ensures that Lexer produces a token stream of BOOL_CONST OR
 * BOOL_CONST for strings like "true or false" and a token stream of ID OR
```

```
 * ID for strings like "variable or variable"
 *
 * @throws IOException
 */
@Test
public void orSymbolTest1() throws IOException {

String text = "true or false";
StringReader stringReader = new StringReader(text);
Lexer lexer = new Lexer(stringReader);
List<Integer> tokenList = new ArrayList<Integer>();
Symbol token = lexer.next_token();

while (token.sym != sym.EOF) {
tokenList.add(token.sym);
token = lexer.next_token();
}

assertEquals(
"It should produce 3 tokens for the string '" + text + "'", 3,
tokenList.size());
assertEquals("The first token should be a BOOL_CONST", sym.BOOL_CONST,
tokenList.get(0).intValue());
assertEquals("The second token should be a OR", sym.OR, tokenList
.get(1).intValue());
assertEquals("The third token should be a BOOL_CONST", sym.BOOL_CONST,
tokenList.get(2).intValue());
}

/**
 * Tests for correct parsing of the statements with the logical 'or'
 * operator
 *
 * Specifically, ensures that Lexer produces a token stream of BOOL_CONST OR
 * BOOL_CONST for strings like "true or false" and a token stream of BOOL OR
 * BOOL for strings like "bool or bool"
 *
 * @throws IOException
 */
@Test
public void orSymbolTest2() throws IOException {

String text = "variable or variable";
StringReader stringReader = new StringReader(text);
Lexer lexer = new Lexer(stringReader);
List<Integer> tokenList = new ArrayList<Integer>();
```

```
Symbol token = lexer.next_token();

while (token.sym != sym.EOF) {
tokenList.add(token.sym);
token = lexer.next_token();
}

assertEquals(
"It should produce 3 tokens for the string '" + text + "'", 3,
tokenList.size());
assertEquals("The first token should be a ID", sym.ID, tokenList.get(0)
.intValue());
assertEquals("The second token should be a OR", sym.OR, tokenList
.get(1).intValue());
assertEquals("The third token should be a ID", sym.ID, tokenList.get(2)
.intValue());
}

/**
 * Tests for correct parsing of the statements with the logical 'not'
 * operator
 *
 * Specifically, ensures that Lexer produces a token stream of NOT
 * BOOL_CONST for strings like "not true"
 *
 * @throws IOException
 */
@Test
public void notSymbolTest() throws IOException {

String text = "not true";
StringReader stringReader = new StringReader(text);
Lexer lexer = new Lexer(stringReader);
List<Integer> tokenList = new ArrayList<Integer>();
Symbol token = lexer.next_token();

while (token.sym != sym.EOF) {
tokenList.add(token.sym);
token = lexer.next_token();
}

assertEquals(
"It should produce 2 tokens for the string '" + text + "'", 2,
tokenList.size());
assertEquals("The first token should be a NOT", sym.NOT,
tokenList.get(0).intValue());
```

```
assertEquals("The second token should be a BOOL", sym.BOOL_CONST,
tokenList.get(1).intValue());
}

/**
 * Tests for correct parsing of the statements with the 'greater than'
 * comparison operator
 *
 * Specifically, ensures that Lexer produces a token stream of INT_CONST
 * GRTR INT_CONST for strings like "2 > 1" and a token stream of ID GRTR ID
 * for strings like "number > number"
 *
 * @throws IOException
 */
@Test
public void greaterThanSymbolTest1() throws IOException {

String text = "2 > 1";
StringReader stringReader = new StringReader(text);
Lexer lexer = new Lexer(stringReader);
List<Integer> tokenList = new ArrayList<Integer>();
Symbol token = lexer.next_token();

while (token.sym != sym.EOF) {
tokenList.add(token.sym);
token = lexer.next_token();
}

assertEquals(
"It should produce 3 tokens for the string '" + text + "'", 3,
tokenList.size());
assertEquals("The first token should be a INT_CONST", sym.INT_CONST,
tokenList.get(0).intValue());
assertEquals("The second token should be a GRTR", sym.GRTR, tokenList
.get(1).intValue());
assertEquals("The third token should be a INT_CONST", sym.INT_CONST,
tokenList.get(2).intValue());
}

/**
 * Tests for correct parsing of the statements with the 'greater than'
 * comparison operator
 *
 * Specifically, ensures that Lexer produces a token stream of INT_CONST
 * GRTR INT_CONST for strings like "2 > 1" and a token stream of ID GRTR ID
 * for strings like "number > number"
```

```
 *
 * @throws IOException
 */
@Test
public void greaterThanSymbolTest2() throws IOException {

String text = "number > number";
StringReader stringReader = new StringReader(text);
Lexer lexer = new Lexer(stringReader);
List<Integer> tokenList = new ArrayList<Integer>();
Symbol token = lexer.next_token();

while (token.sym != sym.EOF) {
tokenList.add(token.sym);
token = lexer.next_token();
}

assertEquals(
"It should produce 3 tokens for the string '" + text + "'", 3,
tokenList.size());
assertEquals("The first token should be a ID", sym.ID, tokenList.get(0)
.intValue());
assertEquals("The second token should be a GRTR", sym.GRTR, tokenList
.get(1).intValue());
assertEquals("The third token should be a ID", sym.ID, tokenList.get(2)
.intValue());
}

/**
 * Tests for correct parsing of the statements with the 'greater than or
 * equal to' comparison operator
 *
 * Specifically, ensures that Lexer produces a token stream of INT_CONST
 * GRTR_EQL INT_CONST for strings like "2 >= 1" and a token stream of ID
 * GRTR_EQL ID for strings like "number >= number"
 *
 * @throws IOException
 */
@Test
public void greaterThanOrEqualSymbolTest1() throws IOException {

String text = "2 >= 1";
StringReader stringReader = new StringReader(text);
Lexer lexer = new Lexer(stringReader);
List<Integer> tokenList = new ArrayList<Integer>();
Symbol token = lexer.next_token();
```

```
while (token.sym != sym.EOF) {
tokenList.add(token.sym);
token = lexer.next_token();
}

assertEquals(
"It should produce 3 tokens for the string '" + text + "'", 3,
tokenList.size());
assertEquals("The first token should be a INT_CONST", sym.INT_CONST,
tokenList.get(0).intValue());
assertEquals("The second token should be a GRTR_EQL", sym.GRTR_EQL,
tokenList.get(1).intValue());
assertEquals("The third token should be a INT_CONST", sym.INT_CONST,
tokenList.get(2).intValue());
}

/**
 * Tests for correct parsing of the statements with the 'greater than or
 * equal to' comparison operator
 *
 * Specifically, ensures that Lexer produces a token stream of INT_CONST
 * GRTR_EQL INT_CONST for strings like "2 >= 1" and a token stream of ID
 * GRTR_EQL ID for strings like "number >= number"
 *
 * @throws IOException
 */
@Test
public void greaterThanOrEqualSymbolTest2() throws IOException {

String text = "number >= number";
StringReader stringReader = new StringReader(text);
Lexer lexer = new Lexer(stringReader);
List<Integer> tokenList = new ArrayList<Integer>();
Symbol token = lexer.next_token();

while (token.sym != sym.EOF) {
tokenList.add(token.sym);
token = lexer.next_token();
}

assertEquals(
"It should produce 3 tokens for the string '" + text + "'", 3,
tokenList.size());
assertEquals("The first token should be a ID", sym.ID, tokenList.get(0)
.intValue());
```

```
assertEquals("The second token should be a GRTR_EQL", sym.GRTR_EQL,
tokenList.get(1).intValue());
assertEquals("The third token should be a ID", sym.ID, tokenList.get(2)
.intValue());
}

/**
 * Tests for correct parsing of the statements with the 'less than'
 * comparison operator
 *
 * Specifically, ensures that Lexer produces a token stream of INT_CONST
 * LESS INT_CONST for strings like "2 < 1" and a token stream of ID LESS ID
 * for strings like "number < number"
 *
 * @throws IOException
 */
@Test
public void lessThanSymbolTest1() throws IOException {

String text = "2 < 1";
StringReader stringReader = new StringReader(text);
Lexer lexer = new Lexer(stringReader);
List<Integer> tokenList = new ArrayList<Integer>();
Symbol token = lexer.next_token();

while (token.sym != sym.EOF) {
tokenList.add(token.sym);
token = lexer.next_token();
}

assertEquals(
"It should produce 3 tokens for the string '" + text + "'", 3,
tokenList.size());
assertEquals("The first token should be a INT_CONST", sym.INT_CONST,
tokenList.get(0).intValue());
assertEquals("The second token should be a LESS", sym.LESS, tokenList
.get(1).intValue());
assertEquals("The third token should be a INT_CONST", sym.INT_CONST,
tokenList.get(2).intValue());
}

/**
 * Tests for correct parsing of the statements with the 'less than'
 * comparison operator
 *
 * Specifically, ensures that Lexer produces a token stream of INT_CONST
```

```
 * LESS INT_CONST for strings like "2 < 1" and a token stream of ID LESS ID
 * for strings like "number < number"
 *
 * @throws IOException
 */
@Test
public void lessThanSymbolTest2() throws IOException {

String text = "number < number";
StringReader stringReader = new StringReader(text);
Lexer lexer = new Lexer(stringReader);
List<Integer> tokenList = new ArrayList<Integer>();
Symbol token = lexer.next_token();

while (token.sym != sym.EOF) {
tokenList.add(token.sym);
token = lexer.next_token();
}

assertEquals(
"It should produce 3 tokens for the string '" + text + "'", 3,
tokenList.size());
assertEquals("The first token should be a ID", sym.ID, tokenList.get(0)
.intValue());
assertEquals("The second token should be a LESS", sym.LESS, tokenList
.get(1).intValue());
assertEquals("The third token should be a ID", sym.ID, tokenList.get(2)
.intValue());
}

/**
 * Tests for correct parsing of the statements with the 'less than or equal
 * to' comparison operator
 *
 * Specifically, ensures that Lexer produces a token stream of INT_CONST
 * LESS_EQL INT_CONST for strings like "2 <= 1" and a token stream of ID
 * LESS_EQL ID for strings like "number <= number"
 *
 * @throws IOException
 */
@Test
public void lessThanOrEqualSymbolTest1() throws IOException {

String text = "2 <= 1";
StringReader stringReader = new StringReader(text);
Lexer lexer = new Lexer(stringReader);
```

```
List<Integer> tokenList = new ArrayList<Integer>();
Symbol token = lexer.next_token();

while (token.sym != sym.EOF) {
tokenList.add(token.sym);
token = lexer.next_token();
}

assertEquals(
"It should produce 3 tokens for the string '" + text + "'", 3,
tokenList.size());
assertEquals("The first token should be a INT_CONST", sym.INT_CONST,
tokenList.get(0).intValue());
assertEquals("The second token should be a LESS_EQL", sym.LESS_EQL,
tokenList.get(1).intValue());
assertEquals("The third token should be a INT_CONST", sym.INT_CONST,
tokenList.get(2).intValue());
}

/**
 * Tests for correct parsing of the statements with the 'less than or equal
 * to' comparison operator
 *
 * Specifically, ensures that Lexer produces a token stream of INT_CONST
 * LESS_EQL INT_CONST for strings like "2 <= 1" and a token stream of ID
 * LESS_EQL ID for strings like "number <= number"
 *
 * @throws IOException
 */
@Test
public void lessThanOrEqualSymbolTest2() throws IOException {

String text = "number <= number";
StringReader stringReader = new StringReader(text);
Lexer lexer = new Lexer(stringReader);
List<Integer> tokenList = new ArrayList<Integer>();
Symbol token = lexer.next_token();

while (token.sym != sym.EOF) {
tokenList.add(token.sym);
token = lexer.next_token();
}

assertEquals(
"It should produce 3 tokens for the string '" + text + "'", 3,
tokenList.size());
```

```
assertEquals("The first token should be a ID", sym.ID, tokenList.get(0)
.intValue());
assertEquals("The second token should be a LESS_EQL", sym.LESS_EQL,
tokenList.get(1).intValue());
assertEquals("The third token should be a ID", sym.ID, tokenList.get(2)
.intValue());
}

/**
 * Tests for correct parsing of the statements with the 'not equal to'
 * comparison operator
 *
 * Specifically, ensures that Lexer produces a token stream of INT_CONST
 * NOT_EQLS INT_CONST for strings like "2 != 1" and a token stream of ID
 * NOT_EQLS ID for strings like "number != number"
 *
 * @throws IOException
 */
@Test
public void notEqualSymbolTest1() throws IOException {

String text = "2 != 1";
StringReader stringReader = new StringReader(text);
Lexer lexer = new Lexer(stringReader);
List<Integer> tokenList = new ArrayList<Integer>();
Symbol token = lexer.next_token();

while (token.sym != sym.EOF) {
tokenList.add(token.sym);
token = lexer.next_token();
}

assertEquals(
"It should produce 3 tokens for the string '" + text + "'", 3,
tokenList.size());
assertEquals("The first token should be a INT_CONST", sym.INT_CONST,
tokenList.get(0).intValue());
assertEquals("The second token should be a NOT_EQLS", sym.NOT_EQLS,
tokenList.get(1).intValue());
assertEquals("The third token should be a INT_CONST", sym.INT_CONST,
tokenList.get(2).intValue());
}

/**
 * Tests for correct parsing of the statements with the 'not equal to'
 * comparison operator
```

```
 *
 * Specifically, ensures that Lexer produces a token stream of INT_CONST
 * NOT_EQLS INT_CONST for strings like "2 != 1" and a token stream of ID
 * NOT_EQLS ID for strings like "number != number"
 *
 * @throws IOException
 */
@Test
public void notEqualSymbolTest2() throws IOException {

String text = "number != number";
StringReader stringReader = new StringReader(text);
Lexer lexer = new Lexer(stringReader);
List<Integer> tokenList = new ArrayList<Integer>();
Symbol token = lexer.next_token();

while (token.sym != sym.EOF) {
tokenList.add(token.sym);
token = lexer.next_token();
}

assertEquals(
"It should produce 3 tokens for the string '" + text + "'", 3,
tokenList.size());
assertEquals("The first token should be a ID", sym.ID, tokenList.get(0)
.intValue());
assertEquals("The second token should be a NOT_EQLS", sym.NOT_EQLS,
tokenList.get(1).intValue());
assertEquals("The third token should be a ID", sym.ID, tokenList.get(2)
.intValue());
}

/**
 * Tests for correct parsing of the statements with the 'equal to'
 * comparison operator
 *
 * Specifically, ensures that Lexer produces a token stream of INT_CONST
 * DBL_EQLS INT_CONST for strings like "2 == 1" and a token stream of ID
 * DBL_EQLS ID for strings like "number == number"
 *
 * @throws IOException
 */
@Test
public void equalSymbolTest1() throws IOException {

String text = "2 == 1";
```

```
StringReader stringReader = new StringReader(text);
Lexer lexer = new Lexer(stringReader);
List<Integer> tokenList = new ArrayList<Integer>();
Symbol token = lexer.next_token();

while (token.sym != sym.EOF) {
tokenList.add(token.sym);
token = lexer.next_token();
}

assertEquals(
"It should produce 3 tokens for the string '" + text + "'", 3,
tokenList.size());
assertEquals("The first token should be a INT_CONST", sym.INT_CONST,
tokenList.get(0).intValue());
assertEquals("The second token should be a DBL_EQLS", sym.DBL_EQLS,
tokenList.get(1).intValue());
assertEquals("The third token should be a INT_CONST", sym.INT_CONST,
tokenList.get(2).intValue());
}

/**
 * Tests for correct parsing of the statements with the 'equal to'
 * comparison operator
 *
 * Specifically, ensures that Lexer produces a token stream of INT_CONST
 * DBL_EQLS INT_CONST for strings like "2 == 1" and a token stream of ID
 * DBL_EQLS ID for strings like "number == number"
 *
 * @throws IOException
 */
@Test
public void equalSymbolTest2() throws IOException {

String text = "number == number";
StringReader stringReader = new StringReader(text);
Lexer lexer = new Lexer(stringReader);
List<Integer> tokenList = new ArrayList<Integer>();
Symbol token = lexer.next_token();

while (token.sym != sym.EOF) {
tokenList.add(token.sym);
token = lexer.next_token();
}

assertEquals(
```

```
"It should produce 3 tokens for the string '" + text + "'", 3,
tokenList.size());
assertEquals("The first token should be a ID", sym.ID, tokenList.get(0)
.intValue());
assertEquals("The second token should be a DBL_EQLS", sym.DBL_EQLS,
tokenList.get(1).intValue());
assertEquals("The third token should be a ID", sym.ID, tokenList.get(2)
.intValue());
}

/**
 * Tests for correct parsing of the statements with 'if'
 *
 * Specifically, ensures that Lexer produces a token stream of IF L_PAREN ID
 * GRTR ID R_PAREN for strings like "if(number > number)"
 *
 * @throws IOException
 */
@Test
public void ifSymbolTest() throws IOException {

String text = "if(number > number)";
StringReader stringReader = new StringReader(text);
Lexer lexer = new Lexer(stringReader);
List<Integer> tokenList = new ArrayList<Integer>();
Symbol token = lexer.next_token();

while (token.sym != sym.EOF) {
tokenList.add(token.sym);
token = lexer.next_token();
}

assertEquals(
"It should produce 6 tokens for the string '" + text + "'", 6,
tokenList.size());
assertEquals("The first token should be a IF", sym.IF, tokenList.get(0)
.intValue());
assertEquals("The second token should be a L_PAREN", sym.L_PAREN,
tokenList.get(1).intValue());
assertEquals("The third token should be a ID", sym.ID, tokenList.get(2)
.intValue());
assertEquals("The fourth token should be a GRTR", sym.GRTR, tokenList
.get(3).intValue());
assertEquals("The fifth token should be a ID", sym.ID, tokenList.get(4)
.intValue());
assertEquals("The sixth token should be a R_PAREN", sym.R_PAREN,
```

```
tokenList.get(5).intValue());
}

/**
 * Tests for correct parsing of the statements surrounded by braces '{' and
 * '}'
 *
 * Specifically, ensures that Lexer produces a token stream of L_BRACE INT
 * ID R_BRACE for strings like "{  int variable }"
 *
 * @throws IOException
 */
@Test
public void braceSymbolTest() throws IOException {

String text = "{  int variable }";
StringReader stringReader = new StringReader(text);
Lexer lexer = new Lexer(stringReader);
List<Integer> tokenList = new ArrayList<Integer>();
Symbol token = lexer.next_token();

while (token.sym != sym.EOF) {
tokenList.add(token.sym);
token = lexer.next_token();
}

assertEquals(
"It should produce 4 tokens for the string '" + text + "'", 4,
tokenList.size());
assertEquals("The first token should be a L_BRACE", sym.L_BRACE,
tokenList.get(0).intValue());
assertEquals("The second token should be a INT", sym.INT, tokenList
.get(1).intValue());
assertEquals("The third token should be a ID", sym.ID, tokenList.get(2)
.intValue());
assertEquals("The fourth token should be a R_BRACE", sym.R_BRACE,
tokenList.get(3).intValue());
}

/**
 * Tests for correct parsing of the statements with an 'else' clause
 *
 * Specifically, ensures that Lexer produces a token stream of IF L_PAREN ID
 * GRTR ID R_PAREN L_BRACE INT ID R_BRACE ELSE L_BRACE INT ID R_BRACE for
 * strings like
 * "if(number > numberTwo){  int variable } else {  int other }"
```

```
 *
 * @throws IOException
 */
@Test
public void elseSymbolTest() throws IOException {

String text = "if(number > numberTwo){  int variable } else {  int other }";
StringReader stringReader = new StringReader(text);
Lexer lexer = new Lexer(stringReader);
List<Integer> tokenList = new ArrayList<Integer>();
Symbol token = lexer.next_token();

while (token.sym != sym.EOF) {
tokenList.add(token.sym);
token = lexer.next_token();
}

assertEquals("It should produce 15 tokens for the string '" + text
+ "'", 15, tokenList.size());
assertEquals("The first token should be a IF", sym.IF, tokenList.get(0)
.intValue());
assertEquals("The second token should be a L_PAREN", sym.L_PAREN,
tokenList.get(1).intValue());
assertEquals("The third token should be a ID", sym.ID, tokenList.get(2)
.intValue());
assertEquals("The fourth token should be a GRTR", sym.GRTR, tokenList
.get(3).intValue());
assertEquals("The fifth token should be a ID", sym.ID, tokenList.get(4)
.intValue());
assertEquals("The sixth token should be a R_PAREN", sym.R_PAREN,
tokenList.get(5).intValue());
assertEquals("The seventh token should be a L_BRACE", sym.L_BRACE,
tokenList.get(6).intValue());
assertEquals("The eighth token should be a INT", sym.INT, tokenList
.get(7).intValue());
assertEquals("The ninth token should be a ID", sym.ID, tokenList.get(8)
.intValue());
assertEquals("The tenth token should be a R_BRACE", sym.R_BRACE,
tokenList.get(9).intValue());
assertEquals("The eleventh token should be a ELSE", sym.ELSE, tokenList
.get(10).intValue());
assertEquals("The twelfth token should be a L_BRACE", sym.L_BRACE,
tokenList.get(11).intValue());
assertEquals("The thirteenth token should be a INT", sym.INT, tokenList
.get(12).intValue());
assertEquals("The fourteenth token should be a ID", sym.ID, tokenList
```

```
.get(13).intValue());
assertEquals("The fifteenth token should be a R_BRACE", sym.R_BRACE,
tokenList.get(14).intValue());
}

@Test
public void lexerTextTest() throws IOException {

String text = "\"friend\", \"time\", \"now\"";
StringReader stringReader = new StringReader(text);
Lexer lexer = new Lexer(stringReader);
List<Integer> tokenList = new ArrayList<Integer>();
Symbol token = lexer.next_token();

while (token.sym != sym.EOF) {
tokenList.add(token.sym);
token = lexer.next_token();
}

assertEquals("It should produce 5 tokens for the string '" + text
+ "'", 5, tokenList.size());
assertEquals("The first token should be a ID", sym.ID, tokenList.get(0)
.intValue());
assertEquals("The second token should be a COMMA", sym.COMMA,
tokenList.get(1).intValue());
assertEquals("The second token should be a ID", sym.ID,
tokenList.get(2).intValue());
assertEquals("The second token should be a COMMA", sym.COMMA,
tokenList.get(3).intValue());
assertEquals("The second token should be a ID", sym.ID,
tokenList.get(4).intValue());
 }


/**
 * Tests for correct parsing of the statements with an 'elseif' clause
 *
 * Specifically, ensures that Lexer produces a token stream of IF L_PAREN ID
 * GRTR ID R_PAREN L_BRACE INT ID SEMICOL R_BRACE ELSEIF L_PAREN ID DBL_EQLS
 * ID R_PAREN L_BRACE INT ID SEMICOL R_BRACE ELSE L_BRACE INT ID SEMICOL
 * R_BRACE for strings like "if(number > numberTwo){ int variable; }
 * elseif(number == numberTwo) { int next; } else { int other; }"
 *
 * @throws IOException
 */
@Test
```

```
public void elseIfSymbolTest() throws IOException {

String text = "if(number > numberTwo){  int variable; } elseif (number == numberTwo)"
+ "{  int next; } else {  int other; }";
StringReader stringReader = new StringReader(text);
Lexer lexer = new Lexer(stringReader);
List<Integer> tokenList = new ArrayList<Integer>();
Symbol token = lexer.next_token();

while (token.sym != sym.EOF) {
tokenList.add(token.sym);
token = lexer.next_token();
}

assertEquals("It should produce 28 tokens for the string '" + text
+ "'", 28, tokenList.size());
assertEquals("The first token should be a IF", sym.IF, tokenList.get(0)
.intValue());
assertEquals("The second token should be a L_PAREN", sym.L_PAREN,
tokenList.get(1).intValue());
assertEquals("The third token should be a ID", sym.ID, tokenList.get(2)
.intValue());
assertEquals("The fourth token should be a GRTR", sym.GRTR, tokenList
.get(3).intValue());
assertEquals("The fifth token should be a ID", sym.ID, tokenList.get(4)
.intValue());
assertEquals("The sixth token should be a R_PAREN", sym.R_PAREN,
tokenList.get(5).intValue());
assertEquals("The seventh token should be a L_BRACE", sym.L_BRACE,
tokenList.get(6).intValue());
assertEquals("The eighth token should be a INT", sym.INT, tokenList
.get(7).intValue());
assertEquals("The ninth token should be a ID", sym.ID, tokenList.get(8)
.intValue());
assertEquals("The tenth token should be a SEMICOL", sym.SEMICOL,
tokenList.get(9).intValue());
assertEquals("The eleventh token should be a R_BRACE", sym.R_BRACE,
tokenList.get(10).intValue());
assertEquals("The twelfth token should be a ELSEIF", sym.ELSEIF,
tokenList.get(11).intValue());
assertEquals("The thirteenth token should be a L_PAREN", sym.L_PAREN,
tokenList.get(12).intValue());
assertEquals("The fourteenth token should be a ID", sym.ID, tokenList
.get(13).intValue());
assertEquals("The fifteenth token should be a DBL_EQLS", sym.DBL_EQLS,
tokenList.get(14).intValue());
```

```
assertEquals("The sixteenth token should be a ID", sym.ID, tokenList
.get(15).intValue());
assertEquals("The seventeenth token should be a R_PAREN", sym.R_PAREN,
tokenList.get(16).intValue());
assertEquals("The eighteenth token should be a L_BRACE", sym.L_BRACE,
tokenList.get(17).intValue());
assertEquals("The nineteenth token should be a INT", sym.INT, tokenList
.get(18).intValue());
assertEquals("The twentieth token should be a ID", sym.ID, tokenList
.get(19).intValue());
assertEquals("The twenty first token should be a SEMICOL", sym.SEMICOL,
tokenList.get(20).intValue());
assertEquals("The twenty second token should be a R_BRACE",
sym.R_BRACE, tokenList.get(21).intValue());
assertEquals("The twenty third token should be a ELSE", sym.ELSE,
tokenList.get(22).intValue());
assertEquals("The twenth fourth token should be a L_BRACE",
sym.L_BRACE, tokenList.get(23).intValue());
assertEquals("The twenty fifth token should be a INT", sym.INT,
tokenList.get(24).intValue());
assertEquals("The twenty sixth token should be a ID", sym.ID, tokenList
.get(25).intValue());
assertEquals("The twenty seventh token should be a SEMICOL",
sym.SEMICOL, tokenList.get(26).intValue());
assertEquals("The twenty eighth token should be a R_BRACE",
sym.R_BRACE, tokenList.get(27).intValue());
}

/**
 * Tests for correct parsing of the statements with the 'assignment' symbol
 *
 * Specifically, ensures that Lexer produces a token stream of INT ID ASSIGN
 * INT_CONST for strings like "int variable = 1"
 *
 * @throws IOException
 */
@Test
public void assignmentSymbolTest1() throws IOException {

String text = "int variable = 1";
StringReader stringReader = new StringReader(text);
Lexer lexer = new Lexer(stringReader);
List<Integer> tokenList = new ArrayList<Integer>();
Symbol token = lexer.next_token();

while (token.sym != sym.EOF) {
```

```
tokenList.add(token.sym);
token = lexer.next_token();
}

assertEquals(
"It should produce 4 tokens for the string '" + text + "'", 4,
tokenList.size());
assertEquals("The first token should be a INT", sym.INT,
tokenList.get(0).intValue());
assertEquals("The second token should be a ID", sym.ID, tokenList
.get(1).intValue());
assertEquals("The third token should be a ASSIGN", sym.ASSIGN,
tokenList.get(2).intValue());
assertEquals("The fourth token should be a INT_CONST", sym.INT_CONST,
tokenList.get(3).intValue());
}

/**
 * Tests for correct parsing of the statements with the 'assignment' symbol
 *
 * Specifically, ensures that Lexer produces a token stream of BOOL ID
 * ASSIGN BOOL_CONST for strings like "bool variable = true"
 *
 * @throws IOException
 */
@Test
public void assignmentSymbolTest2() throws IOException {

String text = "bool variable = true";
StringReader stringReader = new StringReader(text);
Lexer lexer = new Lexer(stringReader);
List<Integer> tokenList = new ArrayList<Integer>();
Symbol token = lexer.next_token();

while (token.sym != sym.EOF) {
tokenList.add(token.sym);
token = lexer.next_token();
}

assertEquals(
"It should produce 4 tokens for the string '" + text + "'", 4,
tokenList.size());
assertEquals("The first token should be a BOOL", sym.BOOL, tokenList
.get(0).intValue());
assertEquals("The second token should be a ID", sym.ID, tokenList
.get(1).intValue());
```

```
assertEquals("The third token should be a ASSIGN", sym.ASSIGN,
tokenList.get(2).intValue());
assertEquals("The fourth token should be a BOOL_CONST", sym.BOOL_CONST,
tokenList.get(3).intValue());
}

/**
 * Tests for correct parsing of the statements with the 'assignment' symbol
 *
 * Specifically, ensures that Lexer produces a token stream of REAL ID
 * ASSIGN REAL_CONST for strings like "bool variable = true"
 *
 * @throws IOException
 */
@Test
public void assignmentSymbolTest3() throws IOException {

String text = "real variable = 1.3";
StringReader stringReader = new StringReader(text);
Lexer lexer = new Lexer(stringReader);
List<Integer> tokenList = new ArrayList<Integer>();
Symbol token = lexer.next_token();

while (token.sym != sym.EOF) {
tokenList.add(token.sym);
token = lexer.next_token();
}

assertEquals(
"It should produce 4 tokens for the string '" + text + "'", 4,
tokenList.size());
assertEquals("The first token should be a REAL", sym.REAL, tokenList
.get(0).intValue());
assertEquals("The second token should be a ID", sym.ID, tokenList
.get(1).intValue());
assertEquals("The third token should be a ASSIGN", sym.ASSIGN,
tokenList.get(2).intValue());
assertEquals("The fourth token should be a REAL_CONST", sym.REAL_CONST,
tokenList.get(3).intValue());
}

/**
 * Tests for correct parsing of the statements with the 'assignment' symbol
 *
 * Specifically, ensures that Lexer produces a token stream of TEXT ID
 * ASSIGN TEXT_LITERAL for strings like "TEXT variable = friend"
```

```
 *
 * @throws IOException
 */
@Test
public void assignmentSymbolTest4() throws IOException {

String text = "text variable = \"friend\"";
StringReader stringReader = new StringReader(text);
Lexer lexer = new Lexer(stringReader);
List<Integer> tokenList = new ArrayList<Integer>();
Symbol token = lexer.next_token();

while (token.sym != sym.EOF) {
tokenList.add(token.sym);
token = lexer.next_token();
}

assertEquals(
"It should produce 4 tokens for the string '" + text + "'", 4,
tokenList.size());
assertEquals("The first token should be a TEXT", sym.TEXT, tokenList
.get(0).intValue());
assertEquals("The second token should be a ID", sym.ID, tokenList
.get(1).intValue());
assertEquals("The third token should be a ASSIGN", sym.ASSIGN,
tokenList.get(2).intValue());
assertEquals("The fourth token should be a TEXT_LITERAL",
sym.TEXT_LITERAL, tokenList.get(3).intValue());
}

/**
 * Tests for correct parsing of the statements with 'while'
 *
 * Specifically, ensures that Lexer produces a token stream of WHILE L_PAREN
 * ID LESS INT_CONST R_PAREN for strings like "while(count < 10)"
 *
 * @throws IOException
 */
@Test
public void whileSymbolTest() throws IOException {

String text = "while(count < 10)";
StringReader stringReader = new StringReader(text);
Lexer lexer = new Lexer(stringReader);
List<Integer> tokenList = new ArrayList<Integer>();
Symbol token = lexer.next_token();
```

```
while (token.sym != sym.EOF) {
tokenList.add(token.sym);
token = lexer.next_token();
}

assertEquals(
"It should produce 6 tokens for the string '" + text + "'", 6,
tokenList.size());
assertEquals("The first token should be a WHILE", sym.WHILE, tokenList
.get(0).intValue());
assertEquals("The second token should be a L_PAREN", sym.L_PAREN,
tokenList.get(1).intValue());
assertEquals("The third token should be a ID", sym.ID, tokenList.get(2)
.intValue());
assertEquals("The fourth token should be a LESS", sym.LESS, tokenList
.get(3).intValue());
assertEquals("The fifth token should be a INT_CONST", sym.INT_CONST,
tokenList.get(4).intValue());
assertEquals("The sixth token should be a R_PAREN", sym.R_PAREN,
tokenList.get(5).intValue());
}

/**
 * Tests for correct parsing of the statements with the 'return' symbol
 *
 * Specifically, ensures that Lexer produces a token stream of RETURN ID for
 * strings like "return variable"
 *
 * @throws IOException
 */
@Test
public void returnSymbolTest() throws IOException {

String text = "return variable";
StringReader stringReader = new StringReader(text);
Lexer lexer = new Lexer(stringReader);
List<Integer> tokenList = new ArrayList<Integer>();
Symbol token = lexer.next_token();

while (token.sym != sym.EOF) {
tokenList.add(token.sym);
token = lexer.next_token();
}

assertEquals(
```

```
"It should produce 2 tokens for the string '" + text + "'", 2,
tokenList.size());
assertEquals("The first token should be a RETURN", sym.RETURN,
tokenList.get(0).intValue());
assertEquals("The second token should be a ID", sym.ID, tokenList
.get(1).intValue());
}

/**
 * Tests for correct parsing of the statements with brackets '[' and ']'
 *
 * Specifically, ensures that Lexer produces a token stream of ID L_BRKT
 * INT_CONST R_BRKT for strings like "a[2]"
 *
 * @throws IOException
 */
@Test
public void bracketSymbolTest() throws IOException {

String text = "a[2]";
StringReader stringReader = new StringReader(text);
Lexer lexer = new Lexer(stringReader);
List<Integer> tokenList = new ArrayList<Integer>();
Symbol token = lexer.next_token();

while (token.sym != sym.EOF) {
tokenList.add(token.sym);
token = lexer.next_token();
}

assertEquals(
"It should produce 4 tokens for the string '" + text + "'", 4,
tokenList.size());
assertEquals("The first token should be a ID", sym.ID, tokenList.get(0)
.intValue());
assertEquals("The second token should be a L_BRKT", sym.L_BRKT,
tokenList.get(1).intValue());
assertEquals("The third token should be a INT_CONST", sym.INT_CONST,
tokenList.get(2).intValue());
assertEquals("The fourth token should be a R_BRKT", sym.R_BRKT,
tokenList.get(3).intValue());
}

/**
 * Tests for correct parsing of the statements with 'switch' statements Also
 * tests for the correct parsing of 'case' and 'default' since they are
```

```
 * inherent to switch statements
 *
 * Specifically, ensures that Lexer produces a token stream of SWITCH
 * L_PAREN ID R_PAREN CASE INT_CONST COL ID ASSIGN INT_CONST CASE INT_CONST
 * COL ID ASSIGN INT_CONST DEFAULT COL ID ASSIGN INT_CONST for strings like
 * "switch(test) case 1: variable=1 case 2: variable=2 default: variable=10"
 *
 * @throws IOException
 */
@Test
public void switchSymbolTest() throws IOException {

String text = "switch(test) case 1: variable=1 case 2: variable=2 default: variable=10";
StringReader stringReader = new StringReader(text);
Lexer lexer = new Lexer(stringReader);
List<Integer> tokenList = new ArrayList<Integer>();
Symbol token = lexer.next_token();

while (token.sym != sym.EOF) {
tokenList.add(token.sym);
token = lexer.next_token();
}

assertEquals("It should produce 21 tokens for the string '" + text
+ "'", 21, tokenList.size());
assertEquals("The first token should be a SWITCH", sym.SWITCH,
tokenList.get(0).intValue());
assertEquals("The second token should be a L_PAREN", sym.L_PAREN,
tokenList.get(1).intValue());
assertEquals("The third token should be a ID", sym.ID, tokenList.get(2)
.intValue());
assertEquals("The fourth token should be a R_PAREN", sym.R_PAREN,
tokenList.get(3).intValue());
assertEquals("The fifth token should be a CASE", sym.CASE, tokenList
.get(4).intValue());
assertEquals("The sixth token should be a INT_CONST", sym.INT_CONST,
tokenList.get(5).intValue());
assertEquals("The seventh token should be a COL", sym.COL, tokenList
.get(6).intValue());
assertEquals("The eighth token should be a ID", sym.ID, tokenList
.get(7).intValue());
assertEquals("The ninth token should be a ASSIGN", sym.ASSIGN,
tokenList.get(8).intValue());
assertEquals("The tenth token should be a INT_CONST", sym.INT_CONST,
tokenList.get(9).intValue());
assertEquals("The eleventh token should be a CASE", sym.CASE, tokenList
```

```
.get(10).intValue());
assertEquals("The twelfth token should be a INT_CONST", sym.INT_CONST,
tokenList.get(11).intValue());
assertEquals("The thirteenth token should be a COL", sym.COL, tokenList
.get(12).intValue());
assertEquals("The fourteenth token should be a ID", sym.ID, tokenList
.get(13).intValue());
assertEquals("The fifteenth token should be a ASSIGN", sym.ASSIGN,
tokenList.get(14).intValue());
assertEquals("The sixteenth token should be a INT_CONST",
sym.INT_CONST, tokenList.get(15).intValue());
assertEquals("The seventeenth token should be a DEFAULT", sym.DEFAULT,
tokenList.get(16).intValue());
assertEquals("The eighteenth token should be a COL", sym.COL, tokenList
.get(17).intValue());
assertEquals("The nineteenth token should be a ID", sym.ID, tokenList
.get(18).intValue());
assertEquals("The twentieth token should be a ASSIGN", sym.ASSIGN,
tokenList.get(19).intValue());
assertEquals("The twenty first token should be a INT_CONST",
sym.INT_CONST, tokenList.get(20).intValue());

}

/**
 * Tests for correct parsing of the 'list' symbol
 *
 * Specifically, ensures that Lexer produces a token stream of LIST LESS INT
 * GRTR ID L_PAREN INT_CONST R_PAREN for strings like "list<int> myList(5)"
 *
 * @throws IOException
 */
@Test
public void listSymbolTest() throws IOException {

String text = "list<int> myList(5)";
StringReader stringReader = new StringReader(text);
Lexer lexer = new Lexer(stringReader);
List<Integer> tokenList = new ArrayList<Integer>();
Symbol token = lexer.next_token();

while (token.sym != sym.EOF) {
tokenList.add(token.sym);
token = lexer.next_token();
}
```

```
assertEquals(
"It should produce 8 tokens for the string '" + text + "'", 8,
tokenList.size());
assertEquals("The first token should be a LIST", sym.LIST, tokenList
.get(0).intValue());
assertEquals("The second token should be a LESS", sym.LESS, tokenList
.get(1).intValue());
assertEquals("The third token should be a INT", sym.INT,
tokenList.get(2).intValue());
assertEquals("The fourth token should be a GRTR", sym.GRTR, tokenList
.get(3).intValue());
assertEquals("The first token should be a ID", sym.ID, tokenList.get(4)
.intValue());
assertEquals("The second token should be a L_PAREN", sym.L_PAREN,
tokenList.get(5).intValue());
assertEquals("The third token should be a INT_CONST", sym.INT_CONST,
tokenList.get(6).intValue());
assertEquals("The fourth token should be a R_PAREN", sym.R_PAREN,
tokenList.get(7).intValue());
}

/**
 * Tests for correct parsing of the '@Main' symbol
 *
 * Specifically, ensures that Lexer produces token streams of MAIN L_BRACE
 * R_BRACE for strings like "@Main{}"
 *
 * @throws IOException
 */
@Test
public void mainSymbolTest() throws IOException {

String text = "@Main{}";
StringReader stringReader = new StringReader(text);
Lexer lexer = new Lexer(stringReader);
List<Integer> tokenList = new ArrayList<Integer>();
Symbol token = lexer.next_token();

while (token.sym != sym.EOF) {
tokenList.add(token.sym);
token = lexer.next_token();
}

assertEquals(
"It should produce 3 tokens for the string '" + text + "'", 3,
tokenList.size());
```

```
assertEquals("The first token should be an MAIN", sym.MAIN, tokenList
.get(0).intValue());
assertEquals("The second token should be a L_BRACE", sym.L_BRACE,
tokenList.get(1).intValue());
assertEquals("The third token should be an R_BRACE", sym.R_BRACE,
tokenList.get(2).intValue());

}

/**
 * Tests for correct parsing of the '@Functions' symbol
 *
 * Specifically, ensures that Lexer produces token streams of FUNCTIONS
 * L_BRACE R_BRACE for strings like "@Functions{}"
 *
 * @throws IOException
 */
@Test
public void functionsSymbolTest() throws IOException {

String text = "@Functions{}";
StringReader stringReader = new StringReader(text);
Lexer lexer = new Lexer(stringReader);
List<Integer> tokenList = new ArrayList<Integer>();
Symbol token = lexer.next_token();

while (token.sym != sym.EOF) {
tokenList.add(token.sym);
token = lexer.next_token();
}

assertEquals(
"It should produce 3 tokens for the string '" + text + "'", 3,
tokenList.size());
assertEquals("The first token should be an FUNCTIONS", sym.FUNCTIONS,
tokenList.get(0).intValue());
assertEquals("The second token should be a L_BRACE", sym.L_BRACE,
tokenList.get(1).intValue());
assertEquals("The third token should be an R_BRACE", sym.R_BRACE,
tokenList.get(2).intValue());

}

/**
 * Tests for correct parsing of the '@Map' symbol
 *
```

```
 * Specifically, ensures that Lexer produces token streams of MAP L_BRACE
 * R_BRACE for strings like "@Map{}"
 *
 * @throws IOException
 */
@Test
public void mapSymbolTest() throws IOException {

String text = "@Map{}";
StringReader stringReader = new StringReader(text);
Lexer lexer = new Lexer(stringReader);
List<Integer> tokenList = new ArrayList<Integer>();
Symbol token = lexer.next_token();

while (token.sym != sym.EOF) {
tokenList.add(token.sym);
token = lexer.next_token();
}

assertEquals(
"It should produce 3 tokens for the string '" + text + "'", 3,
tokenList.size());
assertEquals("The first token should be an MAP", sym.MAP, tokenList
.get(0).intValue());
assertEquals("The second token should be a L_BRACE", sym.L_BRACE,
tokenList.get(1).intValue());
assertEquals("The third token should be an R_BRACE", sym.R_BRACE,
tokenList.get(2).intValue());

}

/**
 * Tests for correct parsing of  Text
 *
 * @throws IOException
 */
@Test
public void textTest() throws IOException {

String text = "text face";
StringReader stringReader = new StringReader(text);
Lexer lexer = new Lexer(stringReader);
List<Integer> tokenList = new ArrayList<Integer>();
Symbol token = lexer.next_token();

while (token.sym != sym.EOF) {
```

```
tokenList.add(token.sym);
token = lexer.next_token();
}

assertEquals(
"It should produce 2 tokens for the string '" + text + "'", 2,
tokenList.size());
assertEquals("The first token should be TEXT", sym.TEXT, tokenList
.get(0).intValue());
assertEquals("The second token should be a ID", sym.ID,
tokenList.get(1).intValue());
}

/**
 * Tests for correct parsing of the '@Reduce' symbol
 *
 * Specifically, ensures that Lexer produces token streams of REDUCE L_BRACE
 * R_BRACE for strings like "@Reduce{}"
 *
 * @throws IOException
 */
@Test
public void reduceSymbolTest() throws IOException {

String text = "@Reduce{}";
StringReader stringReader = new StringReader(text);
Lexer lexer = new Lexer(stringReader);
List<Integer> tokenList = new ArrayList<Integer>();
Symbol token = lexer.next_token();

while (token.sym != sym.EOF) {
tokenList.add(token.sym);
token = lexer.next_token();
}

assertEquals(
"It should produce 3 tokens for the string '" + text + "'", 3,
tokenList.size());
assertEquals("The first token should be an REDUCE", sym.REDUCE,
tokenList.get(0).intValue());
assertEquals("The second token should be a L_BRACE", sym.L_BRACE,
tokenList.get(1).intValue());
assertEquals("The third token should be an R_BRACE", sym.R_BRACE,
tokenList.get(2).intValue());

}
```

```
/**
 * Tests for correct parsing of the 'arrow' symbol
 *
 * Specifically, ensures that Lexer produces token streams of MAP L_PAREN
 * INT ID COMMA REAL ID R_PAREN ARROW L_PAREN INT COMMA INT R_PAREN L_BRACE
 * R_BRACE for strings like "@Map(int num, real numTwo) -> (int, int){}"
 *
 * @throws IOException
 */
@Test
public void arrowSymbolTest() throws IOException {

String text = "@Map(int num, real numTwo) -> (int, int){}";
StringReader stringReader = new StringReader(text);
Lexer lexer = new Lexer(stringReader);
List<Integer> tokenList = new ArrayList<Integer>();
Symbol token = lexer.next_token();

while (token.sym != sym.EOF) {
tokenList.add(token.sym);
token = lexer.next_token();
}

assertEquals("It should produce 16 tokens for the string '" + text
+ "'", 16, tokenList.size());
assertEquals("The first token should be an MAP", sym.MAP, tokenList
.get(0).intValue());
assertEquals("The second token should be a L_PAREN", sym.L_PAREN,
tokenList.get(1).intValue());
assertEquals("The third token should be an INT", sym.INT, tokenList
.get(2).intValue());
assertEquals("The fourth token should be an ID", sym.ID,
tokenList.get(3).intValue());
assertEquals("The fifth token should be a COMMA", sym.COMMA, tokenList
.get(4).intValue());
assertEquals("The sixth token should be an REAL", sym.REAL, tokenList
.get(5).intValue());
assertEquals("The seventh token should be an ID", sym.ID, tokenList
.get(6).intValue());
assertEquals("The eighth token should be a R_PAREN", sym.R_PAREN,
tokenList.get(7).intValue());
assertEquals("The ninth token should be an ARROW", sym.ARROW, tokenList
.get(8).intValue());
assertEquals("The tenth token should be an L_PAREN", sym.L_PAREN,
tokenList.get(9).intValue());
```

```
assertEquals("The eleventh token should be a INT", sym.INT, tokenList
.get(10).intValue());
assertEquals("The twelfth token should be an COMMA", sym.COMMA,
tokenList.get(11).intValue());
assertEquals("The thirteenth token should be an INT", sym.INT,
tokenList.get(12).intValue());
assertEquals("The fourteenth token should be a R_PAREN", sym.R_PAREN,
tokenList.get(13).intValue());
assertEquals("The fifteenth token should be an L_BRACE", sym.L_BRACE,
tokenList.get(14).intValue());
assertEquals("The sixteenth token should be an R_BRACE", sym.R_BRACE,
tokenList.get(15).intValue());

}

/**
 * Tests for correct parsing of the 'foreach' and 'in' symbols
 *
 * Specifically, ensures that Lexer produces token streams of FOREACH ID IN
 * ID for strings like "foreach variable in myList"
 *
 * @throws IOException
 */
@Test
public void foreachSymbolTest() throws IOException {

String text = "foreach variable in myList";
StringReader stringReader = new StringReader(text);
Lexer lexer = new Lexer(stringReader);
List<Integer> tokenList = new ArrayList<Integer>();
Symbol token = lexer.next_token();

while (token.sym != sym.EOF) {
tokenList.add(token.sym);
token = lexer.next_token();
}

assertEquals(
"It should produce 4 tokens for the string '" + text + "'", 4,
tokenList.size());
assertEquals("The first token should be an FOREACH", sym.FOREACH,
tokenList.get(0).intValue());
assertEquals("The second token should be a ID", sym.ID, tokenList
.get(1).intValue());
assertEquals("The third token should be an IN", sym.IN, tokenList
.get(2).intValue());
```

```
assertEquals("The third token should be an ID", sym.ID, tokenList
.get(3).intValue());

}

/**
 * Tests for correct parsing of the 'for' statements
 *
 * Specifically, ensures that Lexer produces token streams of FOR L_PAREN
 * INT ID ASSIGN INT_CONST SEMICOL ID LESS INT_CONST SEMICOL ID INCR R_PAREN
 * for strings like "for (int i = 0; i < 10; i++)"
 *
 * @throws IOException
 */
@Test
public void forSymbolTest() throws IOException {

String text = "for (int i = 0; i < 10; i++)";
StringReader stringReader = new StringReader(text);
Lexer lexer = new Lexer(stringReader);
List<Integer> tokenList = new ArrayList<Integer>();
Symbol token = lexer.next_token();

while (token.sym != sym.EOF) {
tokenList.add(token.sym);
token = lexer.next_token();
}

assertEquals("It should produce 14 tokens for the string '" + text
+ "'", 14, tokenList.size());
assertEquals("The first token should be an FOR", sym.FOR, tokenList
.get(0).intValue());
assertEquals("The second token should be a L_PAREN", sym.L_PAREN,
tokenList.get(1).intValue());
assertEquals("The third token should be an INT", sym.INT, tokenList
.get(2).intValue());
assertEquals("The fourth token should be an ID", sym.ID,
tokenList.get(3).intValue());
assertEquals("The fifth token should be an ASSIGN", sym.ASSIGN,
tokenList.get(4).intValue());
assertEquals("The sixth token should be an INT_CONST", sym.INT_CONST,
tokenList.get(5).intValue());
assertEquals("The seventh token should be an SEMICOL", sym.SEMICOL,
tokenList.get(6).intValue());
assertEquals("The eighth token should be an ID", sym.ID,
tokenList.get(7).intValue());
```

```
assertEquals("The ninth token should be an LESS", sym.LESS, tokenList
.get(8).intValue());
assertEquals("The tenth token should be an INT_CONST", sym.INT_CONST,
tokenList.get(9).intValue());
assertEquals("The eleventh token should be an SEMICOL", sym.SEMICOL,
tokenList.get(10).intValue());
assertEquals("The twelfth token should be an ID", sym.ID, tokenList
.get(11).intValue());
assertEquals("The thirteenth token should be an INCR", sym.INCR,
tokenList.get(12).intValue());
assertEquals("The fourteenth token should be an R_PAREN", sym.R_PAREN,
tokenList.get(13).intValue());

}

/**
 * Tests for correct parsing of a program with the
 *
 * @Main, @Map, @Reduce and @Functions sections
 *
 *        Specifically, ensures that Lexer produces token streams of
 *        FUNCTIONS L_BRACE R_BRACE MAP L_PAREN INT ID COMMA REAL ID R_PAREN
 *        ARROW L_PAREN INT COMMA INT R_PAREN L_BRACE R_BRACE REDUCE L_PAREN
 *        INT ID COMMA REAL ID R_PAREN ARROW L_PAREN INT COMMA INT R_PAREN
 *        L_BRACE R_BRACE MAIN L_BRACE R_BRACE for strings like
 *        "@Functions{}@Map(int num, real numTwo) -> (int, int){}
 * @Reduce(int num, real numTwo) -> (int, int){}@Main{}"
 *
 * @throws IOException
 */
@Test
public void programStructureTest() throws IOException {

String text = "@Functions{}@Map(int num, real numTwo) -> (int, int){}"
+ "@Reduce(int num, real numTwo) -> (int, int){}@Main{}";
StringReader stringReader = new StringReader(text);
Lexer lexer = new Lexer(stringReader);
List<Integer> tokenList = new ArrayList<Integer>();
Symbol token = lexer.next_token();

while (token.sym != sym.EOF) {
tokenList.add(token.sym);
token = lexer.next_token();
}

assertEquals("It should produce 38 tokens for the string '" + text
```

```
+ "'", 38, tokenList.size());
assertEquals("The first token should be an FUNCTIONS", sym.FUNCTIONS,
tokenList.get(0).intValue());
assertEquals("The second token should be a L_BRACE", sym.L_BRACE,
tokenList.get(1).intValue());
assertEquals("The third token should be an R_BRACE", sym.R_BRACE,
tokenList.get(2).intValue());
assertEquals("The fourth token should be an MAP", sym.MAP, tokenList
.get(3).intValue());
assertEquals("The fifth token should be a L_PAREN", sym.L_PAREN,
tokenList.get(4).intValue());
assertEquals("The sixth token should be an INT", sym.INT, tokenList
.get(5).intValue());
assertEquals("The seventh token should be an ID", sym.ID, tokenList
.get(6).intValue());
assertEquals("The eighth token should be a COMMA", sym.COMMA, tokenList
.get(7).intValue());
assertEquals("The ninth token should be an REAL", sym.REAL, tokenList
.get(8).intValue());
assertEquals("The tenth token should be an ID", sym.ID, tokenList
.get(9).intValue());
assertEquals("The eleventh token should be a R_PAREN", sym.R_PAREN,
tokenList.get(10).intValue());
assertEquals("The twelfth token should be an ARROW", sym.ARROW,
tokenList.get(11).intValue());
assertEquals("The thirteenth token should be an L_PAREN", sym.L_PAREN,
tokenList.get(12).intValue());
assertEquals("The fourteenth token should be a INT", sym.INT, tokenList
.get(13).intValue());
assertEquals("The fifteenth token should be an COMMA", sym.COMMA,
tokenList.get(14).intValue());
assertEquals("The sixteenth token should be an INT", sym.INT, tokenList
.get(15).intValue());
assertEquals("The seventeenth token should be a R_PAREN", sym.R_PAREN,
tokenList.get(16).intValue());
assertEquals("The eighteenth token should be a L_BRACE", sym.L_BRACE,
tokenList.get(17).intValue());
assertEquals("The nineteenth token should be an R_BRACE", sym.R_BRACE,
tokenList.get(18).intValue());
assertEquals("The twentieth token should be an REDUCE", sym.REDUCE,
tokenList.get(19).intValue());
assertEquals("The twenty first token should be a L_PAREN", sym.L_PAREN,
tokenList.get(20).intValue());
assertEquals("The twenty second token should be an INT", sym.INT,
tokenList.get(21).intValue());
assertEquals("The twenty third token should be an ID", sym.ID,
```

```
tokenList.get(22).intValue());
assertEquals("The twenty fourth token should be a COMMA", sym.COMMA,
tokenList.get(23).intValue());
assertEquals("The twenty fifth token should be an REAL", sym.REAL,
tokenList.get(24).intValue());
assertEquals("The twenty sixth token should be an ID", sym.ID,
tokenList.get(25).intValue());
assertEquals("The twenty seventh token should be a R_PAREN",
sym.R_PAREN, tokenList.get(26).intValue());
assertEquals("The twenty eighth token should be an ARROW", sym.ARROW,
tokenList.get(27).intValue());
assertEquals("The twenty ninth token should be an L_PAREN",
sym.L_PAREN, tokenList.get(28).intValue());
assertEquals("The thirtieth token should be a INT", sym.INT, tokenList
.get(29).intValue());
assertEquals("The thirty first token should be an COMMA", sym.COMMA,
tokenList.get(30).intValue());
assertEquals("The thirty second token should be an INT", sym.INT,
tokenList.get(31).intValue());
assertEquals("The thirty third token should be a R_PAREN", sym.R_PAREN,
tokenList.get(32).intValue());
assertEquals("The thirty fourth token should be a L_BRACE",
sym.L_BRACE, tokenList.get(33).intValue());
assertEquals("The thirty fifth token should be an R_BRACE",
sym.R_BRACE, tokenList.get(34).intValue());
assertEquals("The thirty sixth token should be an MAIN", sym.MAIN,
tokenList.get(35).intValue());
assertEquals("The thirty seventh token should be a L_BRACE",
sym.L_BRACE, tokenList.get(36).intValue());
assertEquals("The thirty eighth token should be an R_BRACE",
sym.R_BRACE, tokenList.get(37).intValue());

}

@Test
public void exceptionTypesTest() throws IOException {

String catchStart = "catch (";
String catchEnd = ")";
List<String> exceptionNameList = new ArrayList<String>(Arrays.asList(
"FileNotFoundException", "FileLoadException",
"ArrayOutOfBoundsException", "IncorrectArgumentException",
"TypeMismatchException", "NullReferenceException",
"ArithmeticException"));

for (String s : exceptionNameList) {
```

```
String text = catchStart + s + catchEnd;
StringReader stringReader = new StringReader(text);
Lexer lexer = new Lexer(stringReader);
List<Integer> tokenList = new ArrayList<Integer>();
Symbol token = lexer.next_token();

while (token.sym != sym.EOF) {
tokenList.add(token.sym);
token = lexer.next_token();
}

assertEquals("It should produce 4 tokens for the string '" + text
+ "'", 4, tokenList.size());
assertEquals("The first token should be sym.CATCH",
sym.CATCH, tokenList.get(0).intValue());
assertEquals("The second token should be a L_PAREN", sym.L_PAREN,
tokenList.get(1).intValue());
//assertEquals("The third token should be an EXCEPTION", sym.EXCEPTION, tokenList.get(2).int
assertEquals("The fourth token should be an R_PAREN", sym.R_PAREN,
tokenList.get(3).intValue());

}
}
}
```

B.3.4 NodeTester.java

```
package test;

import static org.junit.Assert.*;

import java.util.List;
import java.util.ArrayList;
import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;
import util.ast.node.*;
import util.ast.node.PostfixExpressionNode.PostfixType;
import util.ast.node.BiOpNode.OpType;
import util.ast.node.SectionNode.SectionName;

import util.type.Types;
```

```
/**
 *
 * Tests for the functionality provided by the AbstractSyntaxTree class.
 *
 * @author Samuel Messing
 * @author Kurry Tran
 *
 */
public class NodeTester {

private ExpressionNode A;
private ExpressionNode B;
private ExpressionNode C;
private ExpressionNode D;
private ExpressionNode E;
private ExpressionNode F;
private ExpressionNode G;

private ArgumentsNode __argumentsNode;
private BiOpNode __biOpNode;
private CatchesNode __catchesNode;
private ConstantNode __constantNode;
private DerivedTypeNode __derivedTypeNode;
private ElseIfStatementNode __elseIfStatementNode;
private ExceptionTypeNode __exceptionTypeNode;
private ExpressionNode __expressionNode;
private FunctionNode __functionNode;
private GuardingStatementNode __guardingStatementNode;
private IdNode __idNode;
private IfElseStatementNode __ifElseStatementNode;
private IterationStatementNode __iterationStatementNode;
private JumpStatementNode __jumpStatementNode;
private MockExpressionNode __mockExpressionNode;
private MockNode __mockNode;
private ParametersNode __parametersNode;
private PostfixExpressionNode __postfixExpressionNode;
private PrimaryExpressionNode __primaryExpressionNode;
private PrimitiveTypeNode __primitiveTypeNode;
private ProgramNode __programNode;
private RelationalExpressionNode __relationalExpressionNode;
private SectionNode __sectionNode;
private SectionTypeNode __sectionTypeNode;
private SelectionStatementNode __selectionStatementNode;
private StatementListNode __statementListNode;
private StatementNode __statementNode;
private SwitchStatementNode __switchStatementNode;
```

```
private TypeNode __typeNode;
private UnOpNode __unOpNode;

public static final BiOpNode.OpType OPTYPE_ASSIGN = BiOpNode.OpType.ASSIGN;
public static final String ERROR_MESSAGE_TO_STRING = "Nodes should return the proper name wh
public static final String ERROR_MESSAGE_GET_NAME = "Nodes should return the proper name whe


@BeforeClass
public static void setUpClass() throws Exception {
}

@AfterClass
public static void tearDownClass() throws Exception {
}

@Before
public void setUp() {
// A (MultExprNode) -> B (MultExprNode) * C (idNode)
// B -> D * E
C = new IdNode("C");
D = new UnOpNode(UnOpNode.OpType.CAST, new MockExpressionNode());
E = new UnOpNode(UnOpNode.OpType.CAST, new MockExpressionNode());
B = new BiOpNode(BiOpNode.OpType.TIMES, D, E);
A = new BiOpNode(BiOpNode.OpType.TIMES, B, C);

/* FUNCTION_CALL<unknown>
 * |-- IdNode<unknown,emit>
 * L__ ArgumentsNode
 *  |-- IdNode<unknown,count>
 *  L__ IdNode<unknown,word>
 *
 */
__argumentsNode = new ArgumentsNode(new IdNode("values"),new IdNode("next"));
__postfixExpressionNode = new PostfixExpressionNode(PostfixType.METHOD_WITH_PARAMS,  new IdN

/* StatementListNode
 * BiOpNode<ASSIGN>
 * IdNode<Primitive Type: INT, count>
 * ConstantNode<Primitive Type: INT>
 *
 */
__statementListNode = new StatementListNode(new BiOpNode(BiOpNode.OpType.ASSIGN,new IdNode('
__sectionNode = new SectionNode(__statementListNode, SectionNode.SectionName.MAIN);
__biOpNode = new BiOpNode(BiOpNode.OpType.ASSIGN, A, B);
__catchesNode = new CatchesNode(__idNode, __statementListNode);
```

```
__constantNode = new ConstantNode(null, null);
__derivedTypeNode = new DerivedTypeNode(Types.Derived.LIST, null);
__elseIfStatementNode = new ElseIfStatementNode(A, __statementListNode, __elseIfStatementNod
__exceptionTypeNode = new ExceptionTypeNode(null);
__functionNode = new FunctionNode(null, null, __parametersNode, __statementListNode);
__guardingStatementNode = new GuardingStatementNode(__statementListNode, __catchesNode);
__idNode = new IdNode("foo");
__ifElseStatementNode = new IfElseStatementNode(A, __statementListNode, __elseIfStatementNod
__iterationStatementNode = new IterationStatementNode(A, A, __statementListNode) ;
__jumpStatementNode = new JumpStatementNode(null);
__mockExpressionNode = new MockExpressionNode();
__mockNode= new MockNode();
__parametersNode = new ParametersNode(null, null);
__primaryExpressionNode = new PrimaryExpressionNode();
__primitiveTypeNode = new PrimitiveTypeNode(null);
__programNode = new ProgramNode(__sectionNode, __sectionNode, __sectionNode, __sectionNode);
__relationalExpressionNode = new RelationalExpressionNode(null, A, A);
__sectionTypeNode = new SectionTypeNode(__idNode, __idNode, __typeNode, __typeNode);
 __switchStatementNode = new SwitchStatementNode(A, __statementListNode);
__selectionStatementNode = new SelectionStatementNode(A);
__statementNode = new StatementNode();
__unOpNode = new UnOpNode(null, A);

}

@After
public void tearDown() {
}

@Test
public void getNameTest1(){
assertEquals( ERROR_MESSAGE_GET_NAME, "ArgumentsNode", __argumentsNode.getName());
assertEquals( ERROR_MESSAGE_GET_NAME, "BiOpNode", __biOpNode.getName());
assertEquals( ERROR_MESSAGE_GET_NAME, "CatchesNode", __catchesNode.getName());
assertEquals( ERROR_MESSAGE_GET_NAME, "ConstantNode", __constantNode.getName());
assertEquals( ERROR_MESSAGE_GET_NAME, "DerivedTypeNode", __derivedTypeNode.getName());
assertEquals( ERROR_MESSAGE_GET_NAME, "ElseIfStatementNode", __elseIfStatementNode.getName()
assertEquals( ERROR_MESSAGE_GET_NAME, "ExceptionTypeNode", __exceptionTypeNode.getName());
assertEquals( ERROR_MESSAGE_GET_NAME, "ExpressionNode", __expressionNode.getName());
assertEquals( ERROR_MESSAGE_GET_NAME, "FunctionNode", __functionNode.getName());
assertEquals( ERROR_MESSAGE_GET_NAME, "GuardingStatementNode", __guardingStatementNode.getNa
assertEquals( ERROR_MESSAGE_GET_NAME, "IdNode", __idNode.getName());
assertEquals( ERROR_MESSAGE_GET_NAME, "IfElseStatementNode", __ifElseStatementNode.getName()
assertEquals( ERROR_MESSAGE_GET_NAME, "IterationStatementNode", __iterationStatementNode.get
assertEquals( ERROR_MESSAGE_GET_NAME, "JumpStatementNode", __jumpStatementNode.getName());
assertEquals( ERROR_MESSAGE_GET_NAME, "MockExpressionNode", __mockExpressionNode.getName());
```

```
assertEquals( ERROR_MESSAGE_GET_NAME, "MockNode", __mockNode.getName());
assertEquals( ERROR_MESSAGE_GET_NAME, "ParametersNode", __parametersNode.getName());
assertEquals( ERROR_MESSAGE_GET_NAME, "PostFixExpressionNode", __postfixExpressionNode.getNa
assertEquals( ERROR_MESSAGE_GET_NAME, "PrimaryExpressionNode", __primaryExpressionNode.getNa
assertEquals( ERROR_MESSAGE_GET_NAME, "PrimitiveTypeNode", __primitiveTypeNode.getName());
assertEquals( ERROR_MESSAGE_GET_NAME, "ProgramNode", __programNode.getName());
assertEquals( ERROR_MESSAGE_GET_NAME, "RelationalExpressionNode", __relationalExpressionNode
assertEquals( ERROR_MESSAGE_GET_NAME, "SectionNode", __sectionNode.getName());
assertEquals( ERROR_MESSAGE_GET_NAME, "StatementListNode", __statementListNode.getName());
assertEquals( ERROR_MESSAGE_GET_NAME, "StatementNode", __statementNode.getName());
assertEquals( ERROR_MESSAGE_GET_NAME, "SwitchStatementNode", __switchStatementNode.getName()
assertEquals( ERROR_MESSAGE_GET_NAME, "UnOpNode", __unOpNode.getName());
}

@Test
public void getNameTest2(){
assertEquals( ERROR_MESSAGE_GET_NAME, "ArgumentsNode", __argumentsNode.getName());
assertEquals( ERROR_MESSAGE_GET_NAME, "BiOpNode", __biOpNode.getName());
assertEquals( ERROR_MESSAGE_GET_NAME, "CatchesNode", __catchesNode.getName());
assertEquals( ERROR_MESSAGE_GET_NAME, "ConstantNode", __constantNode.getName());
assertEquals( ERROR_MESSAGE_GET_NAME, "DerivedTypeNode", __derivedTypeNode.getName());
assertEquals( ERROR_MESSAGE_GET_NAME, "ElseIfStatementNode", __elseIfStatementNode.getName()
assertEquals( ERROR_MESSAGE_GET_NAME, "ExceptionTypeNode", __exceptionTypeNode.getName());
assertEquals( ERROR_MESSAGE_GET_NAME, "ExpressionNode", __expressionNode.getName());
assertEquals( ERROR_MESSAGE_GET_NAME, "FunctionNode", __functionNode.getName());
assertEquals( ERROR_MESSAGE_GET_NAME, "GuardingStatementNode", __guardingStatementNode.getNa
assertEquals( ERROR_MESSAGE_GET_NAME, "IdNode", __idNode.getName());
assertEquals( ERROR_MESSAGE_GET_NAME, "IfElseStatementNode", __ifElseStatementNode.getName()
assertEquals( ERROR_MESSAGE_GET_NAME, "IterationStatementNode", __iterationStatementNode.get
assertEquals( ERROR_MESSAGE_GET_NAME, "JumpStatementNode", __jumpStatementNode.getName());
assertEquals( ERROR_MESSAGE_GET_NAME, "MockExpressionNode", __mockExpressionNode.getName());
assertEquals( ERROR_MESSAGE_GET_NAME, "MockNode", __mockNode.getName());
assertEquals( ERROR_MESSAGE_GET_NAME, "ParametersNode", __parametersNode.getName());
assertEquals( ERROR_MESSAGE_GET_NAME, "PostFixExpressionNode", __postfixExpressionNode.getNa
assertEquals( ERROR_MESSAGE_GET_NAME, "PrimaryExpressionNode", __primaryExpressionNode.getNa
assertEquals( ERROR_MESSAGE_GET_NAME, "PrimitiveTypeNode", __primitiveTypeNode.getName());
assertEquals( ERROR_MESSAGE_GET_NAME, "ProgramNode", __programNode.getName());
assertEquals( ERROR_MESSAGE_GET_NAME, "RelationalExpressionNode", __relationalExpressionNode
assertEquals( ERROR_MESSAGE_GET_NAME, "SectionNode", __sectionNode.getName());
assertEquals( ERROR_MESSAGE_GET_NAME, "StatementListNode", __statementListNode.getName());
assertEquals( ERROR_MESSAGE_GET_NAME, "StatementNode", __statementNode.getName());
assertEquals( ERROR_MESSAGE_GET_NAME, "SwitchStatementNode", __switchStatementNode.getName()
assertEquals( ERROR_MESSAGE_GET_NAME, "UnOpNode", __unOpNode.getName());
}

@Test
```

```
public void toStringTest1() {

String properName = "BiOpNode<TIMES>";

assertEquals("Nodes should return the proper name when toString() is called.", properName, A

}

@Test
public void toStringTest2() {

String properName = "BiOpNode<TIMES>";

assertEquals(
"Nodes should return the proper name when toString() is called.",
properName, A.toString());

}

@Test(expected = UnsupportedOperationException.class)
public void doubleAddParentTest() {
// note that this.addChild(that) implicitly sets that's parent to be
// this. In this case, D already has it's parent set to B.
B.setParent(D);
}

@Test
public void addChildTest() {
MockNode parent = new MockNode();
MockNode childOne = new MockNode();
MockNode childTwo = new MockNode();
parent.addChild(childOne);
parent.addChild(childTwo);
assertEquals("addChild should properly add children", 2, parent
.getChildren().size());
assertEquals(
"addChild should always add a new child as rightmost child",
childTwo, parent.getChildren().get(1));
assertEquals(
"addChild should always add a new child as rightmost child",
childOne, parent.getChildren().get(0));
parent.addChild(null);
assertEquals("adding a null child should do nothing", 2, parent
.getChildren().size());
}
```

```
@Test
public void removeChildTest() {
A.removeChild(C);
assertEquals("removeChild should properly remove a child", 1, A
.getChildren().size());
assertEquals("removeChild should properly remove a child", false, A
.hasChild(C));
assertEquals("removeChild should properly unset childs parent", null, C
.getParent());
}

@Test
public void setUnsetParentTest() {
MockNode parent = new MockNode();
MockNode child = new MockNode();
assertEquals("Nodes should have null parents by default", null, child
.getParent());
parent.addChild(child);
assertEquals(
"Adding a child to a parent should set parent to be child's parent.",
parent, child.getParent());
child.unsetParent();
assertEquals("Unsetting a child's parent should set parent to null",
null, child.getParent());
assertEquals(
"Unsetting a child's parent removes the child from the parent's children list.",
false, parent.getChildren().contains(child));
}
}
```

B.3.5 ParserTester.java

```
package test;

import static org.junit.Assert.*;

import java.io.StringReader;

import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;

import front_end.Lexer;
```

```
import front_end.Parser;

import util.ast.node.ProgramNode;

/**
 * Tests basic functionality of Parser.java.
 *
 *
 * @author Samuel Messing
 *
 */
public class ParserTester {

@Test
public void firstTest() throws Exception {
String text = "@Map (int lineNum, text line) -> (text, int) {   #{ this is a        block co
StringReader stringReader = new StringReader(text);
Lexer lexer = new Lexer(stringReader);
Parser parser = new Parser(lexer);
ProgramNode root = (ProgramNode) parser.parse().value;
}

}
```

B.3.6 SymbolTableTexter.java

```
package test;

import static org.junit.Assert.*;


import java.io.BufferedWriter;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

import java.util.Set;

import org.junit.Test;

import back_end.SymbolTableVisitor;
import front_end.Lexer;
import front_end.Parser;
```

```
import util.ast.AbstractSyntaxTree;
import util.ast.node.ProgramNode;
import util.symbol_table.SymbolTable;

/**
 *
 * @author Jason Helpern
 *
 */
public class SymbolTableTester {

@Test
public void reservedSymbolTableTest() {
/*
 * create a new symbol table, this should be root
 * and should be populated with all reserved words and functions
 */
SymbolTable sym = new SymbolTable(null);
Set<String> reservedSet = sym.table.keySet();
for(String s : reservedSet){
System.out.println("KEY: "+ s + " VALUE: " + sym.get(s).type);
}
}


@Test
public void symbolTableTest() {
/*
 * create a new symbol table, this should be root
 * and should be populated with all reserved words and functions
 */
String filename = "src/test/Factorial.hog";
ProgramNode root = null;
FileReader fileReader;
    try {
fileReader = new FileReader(new File(filename));
        // Parser p = new Parser(new Lexer(System.in));
     Parser p = new Parser(new Lexer(fileReader));
     root = (ProgramNode) p.parse().value;

      }
      catch (FileNotFoundException e) {
       System.out.println("file not found.");
      }
      catch (Exception ex) {
```

```
       ex.printStackTrace();
      }

    AbstractSyntaxTree ast = new AbstractSyntaxTree(root);
    root.print();

    SymbolTableVisitor symTabVisitor = new SymbolTableVisitor(ast);
    symTabVisitor.walk();
    //write tests
    SymbolTable.print();
    System.out.println("\n\n\n");
    SymbolTable.printSymbolTable();
}


@Test
public void WordCountTest() {
/*
 * create a new symbol table, this should be root
 * and should be populated with all reserved words and functions
 */
String filename = "src/test/WordCount.hog";
ProgramNode root = null;
FileReader fileReader;
    try {
fileReader = new FileReader(new File(filename));
        // Parser p = new Parser(new Lexer(System.in));
     Parser p = new Parser(new Lexer(fileReader));
     root = (ProgramNode) p.parse().value;

      }
      catch (FileNotFoundException e) {
       System.out.println("file not found.");
      }
      catch (Exception ex) {
       ex.printStackTrace();
      }

    AbstractSyntaxTree ast = new AbstractSyntaxTree(root);
    root.print();
    System.out.println("\n\n\n");

    SymbolTableVisitor symTabVisitor = new SymbolTableVisitor(ast);
    symTabVisitor.walk();
    //write tests
    SymbolTable.print();
```

```
    System.out.println("\n\n\n");
    SymbolTable.printSymbolTable();

}

}
```

B.3.7 TypesCheckingTester.java

```
package test;

import static org.junit.Assert.assertEquals;


import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.util.Iterator;

import org.junit.Test;

import util.ast.AbstractSyntaxTree;
import util.ast.node.IdNode;
import util.ast.node.Node;
import util.ast.node.PostfixExpressionNode;
import util.ast.node.ProgramNode;
import util.symbol_table.Symbol;
import util.symbol_table.SymbolTable;
import back_end.SymbolTableVisitor;
import back_end.TypeCheckingVisitor;
import front_end.Lexer;
import front_end.Parser;

/**
 *
 * @author Jason Halpern
 *
 */

public class TypeCheckingTester {

@Test
public void typeCheckingTest() {
String filename = "src/test/TypeDecoratorTest.hog";
ProgramNode root = null;
```

```
FileReader fileReader;
    try {
fileReader = new FileReader(new File(filename));
        // Parser p = new Parser(new Lexer(System.in));
     Parser p = new Parser(new Lexer(fileReader));
     root = (ProgramNode) p.parse().value;

      }
      catch (FileNotFoundException e) {
       System.out.println("file not found.");
      }
      catch (Exception ex) {
       ex.printStackTrace();
      }

    AbstractSyntaxTree ast = new AbstractSyntaxTree(root);
    root.print();

    SymbolTableVisitor symTabVisitor = new SymbolTableVisitor(ast);
    symTabVisitor.walk();
    SymbolTable.printSymbolTable();
    System.out.println("\n\n\n");
    SymbolTable.print();

    root.print();

    TypeCheckingVisitor typeCheckVisitor = new TypeCheckingVisitor(ast);
    typeCheckVisitor.walk();

    root.print();

Iterator<Node> postOrderTraversal = ast.postOrderTraversal();
Node nextNode;
Symbol tempSymbol;
PostfixExpressionNode n;
while (postOrderTraversal.hasNext()) {
nextNode = postOrderTraversal.next();
if(nextNode instanceof IdNode){
tempSymbol = SymbolTable.getSymbolForIdNode((IdNode)nextNode);
System.out.println("Identifier: " + ((IdNode) nextNode).getIdentifier() + " " + tempSymbol.g
}
if(nextNode instanceof PostfixExpressionNode){
n = (PostfixExpressionNode)nextNode;
//tempSymbol = SymbolTable.getSymbolForIdNode((IdNode)nextNode);
System.out.println("Function Name : " + n.getNameOfFunctionOrMethod() + " " + n.getTypeName
}
```

```
}

}

@Test
public void typeCheckingTestTwo() {
String filename = "src/test/TypeDecoratorTestTwo.hog";
ProgramNode root = null;
FileReader fileReader;
    try {
fileReader = new FileReader(new File(filename));
        // Parser p = new Parser(new Lexer(System.in));
     Parser p = new Parser(new Lexer(fileReader));
     root = (ProgramNode) p.parse().value;

      }
      catch (FileNotFoundException e) {
       System.out.println("file not found.");
      }
      catch (Exception ex) {
       ex.printStackTrace();
      }

    AbstractSyntaxTree ast = new AbstractSyntaxTree(root);
    root.print();

    SymbolTableVisitor symTabVisitor = new SymbolTableVisitor(ast);
    symTabVisitor.walk();
    SymbolTable.printSymbolTable();
    System.out.println("\n\n\n");
    SymbolTable.print();

    root.print();

    TypeCheckingVisitor typeCheckVisitor = new TypeCheckingVisitor(ast);
    typeCheckVisitor.walk();

    root.print();

Iterator<Node> postOrderTraversal = ast.postOrderTraversal();
Node nextNode;
Symbol tempSymbol;
PostfixExpressionNode n;
while (postOrderTraversal.hasNext()) {
nextNode = postOrderTraversal.next();
if(nextNode instanceof IdNode){
```

```
tempSymbol = SymbolTable.getSymbolForIdNode((IdNode)nextNode);
System.out.println("Identifier: " + ((IdNode) nextNode).getIdentifier() + " " + tempSymbol.g
}
if(nextNode instanceof PostfixExpressionNode){
n = (PostfixExpressionNode)nextNode;
//tempSymbol = SymbolTable.getSymbolForIdNode((IdNode)nextNode);
System.out.println("Function Name : " + n.getNameOfFunctionOrMethod() + " " + n.getTypeName
}
}

}
}
```

B.3.8 TypesTester.java

```
package test;

import static org.junit.Assert.*;
import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;

import util.ast.node.BiOpNode;
import util.ast.node.DerivedTypeNode;
import util.ast.node.ExceptionTypeNode;
import util.ast.node.PrimitiveTypeNode;
import util.ast.node.TypeNode;
import util.ast.node.UnOpNode;
import util.error.TypeMismatchError;
import util.symbol_table.SymbolTable;
import util.type.Types;

/**
 * A method to test the convenience class for the Types convenience class.
 *
 * @author Samuel Messing
 * @author Jason Halpern
 *
 */
public class TypesTester {

private TypeNode boolNode;
private TypeNode intNode;
```

```
private TypeNode realNode;
private TypeNode textNode;
private TypeNode listInt;
private TypeNode exceptionFileNotFound;
private TypeNode dict_Text_ListInt;
private TypeNode setBool;
private TypeNode listSetBool;
private TypeNode listListSetBool;
private TypeNode setListListSetBool;
private TypeNode iterNode;

@Before
public void setUp() {

boolNode = new PrimitiveTypeNode(Types.Primitive.BOOL);
intNode = new PrimitiveTypeNode(Types.Primitive.INT);
realNode = new PrimitiveTypeNode(Types.Primitive.REAL);
textNode = new PrimitiveTypeNode(Types.Primitive.TEXT);
listInt = new DerivedTypeNode(Types.Derived.LIST, intNode);
exceptionFileNotFound = new ExceptionTypeNode(
Types.Exception.FILE_NOT_FOUND);
setBool = new DerivedTypeNode(Types.Derived.SET, boolNode);
iterNode = new DerivedTypeNode(Types.Derived.ITER, intNode);
listSetBool = new DerivedTypeNode(Types.Derived.LIST, setBool);
listListSetBool = new DerivedTypeNode(Types.Derived.LIST, listSetBool);
setListListSetBool = new DerivedTypeNode(Types.Derived.SET,
listListSetBool);

}

@Test
public void testTypeEquvilance() {

assertFalse(Types.isSameType(boolNode, intNode));
assertTrue(Types.isSameType(dict_Text_ListInt, dict_Text_ListInt));
TypeNode secondIntNode = new PrimitiveTypeNode(Types.Primitive.INT);
assertTrue(Types.isSameType(secondIntNode, intNode));
TypeNode secondListInt = new DerivedTypeNode(Types.Derived.LIST,
secondIntNode);
assertTrue(Types.isSameType(secondListInt, listInt));
assertFalse(Types.isSameType(listListSetBool, listSetBool));
TypeNode secondExceptionFileNotFound = new ExceptionTypeNode(
Types.Exception.FILE_NOT_FOUND);
assertTrue(Types.isSameType(exceptionFileNotFound,
secondExceptionFileNotFound));
```

```
}

@Test
public void testGetHigherNumericType() throws TypeMismatchError {
assertTrue(Types.isSameType(intNode, Types.getHigherNumericType(
intNode, intNode)));
assertTrue(Types.isSameType(realNode, Types.getHigherNumericType(
realNode, intNode)));
assertFalse(Types.isSameType(boolNode, Types.getHigherNumericType(
realNode, realNode)));
assertFalse(Types.isSameType(boolNode, Types.getHigherNumericType(
intNode, intNode)));
assertFalse(Types.isSameType(textNode, Types.getHigherNumericType(
intNode, intNode)));
assertFalse(Types.isSameType(textNode, Types.getHigherNumericType(
realNode, realNode)));
assertFalse(Types.isSameType(textNode, realNode));
assertFalse(Types.isSameType(textNode, intNode));
assertFalse(Types.isSameType(textNode, boolNode));
assertFalse(Types.isSameType(boolNode, intNode));
assertFalse(Types.isSameType(boolNode, realNode));
assertTrue(Types.isSameType(boolNode, boolNode));
}

@Test
public void testUnOpIsCompatible() throws TypeMismatchError {
assertTrue(Types.isCompatible(UnOpNode.OpType.DECR, intNode));
assertTrue(Types.isCompatible(UnOpNode.OpType.DECR, realNode));
assertFalse(Types.isCompatible(UnOpNode.OpType.DECR, boolNode));
assertFalse(Types.isCompatible(UnOpNode.OpType.DECR, textNode));

assertTrue(Types.isCompatible(UnOpNode.OpType.INCR, intNode));
assertTrue(Types.isCompatible(UnOpNode.OpType.INCR, realNode));
assertFalse(Types.isCompatible(UnOpNode.OpType.INCR, boolNode));
assertFalse(Types.isCompatible(UnOpNode.OpType.INCR, textNode));

assertFalse(Types.isCompatible(UnOpNode.OpType.NOT, intNode));
assertFalse(Types.isCompatible(UnOpNode.OpType.NOT, realNode));
assertTrue(Types.isCompatible(UnOpNode.OpType.NOT, boolNode));
assertFalse(Types.isCompatible(UnOpNode.OpType.NOT, textNode));

assertTrue(Types.isCompatible(UnOpNode.OpType.UMINUS, intNode));
assertTrue(Types.isCompatible(UnOpNode.OpType.UMINUS, realNode));
assertFalse(Types.isCompatible(UnOpNode.OpType.UMINUS, boolNode));
assertFalse(Types.isCompatible(UnOpNode.OpType.UMINUS, textNode));
```

```
}

@Test
public void testBiOpIsCompatible() throws TypeMismatchError {
assertTrue(Types.isCompatible(BiOpNode.OpType.AND, boolNode, boolNode));
assertFalse(Types.isCompatible(BiOpNode.OpType.AND, textNode, boolNode));
assertFalse(Types.isCompatible(BiOpNode.OpType.AND, intNode, boolNode));
assertFalse(Types.isCompatible(BiOpNode.OpType.AND, textNode, textNode));

assertTrue(Types.isCompatible(BiOpNode.OpType.OR, boolNode, boolNode));
assertFalse(Types.isCompatible(BiOpNode.OpType.OR, textNode, boolNode));
assertFalse(Types.isCompatible(BiOpNode.OpType.OR, intNode, boolNode));
assertFalse(Types.isCompatible(BiOpNode.OpType.OR, textNode, textNode));

assertFalse(Types.isCompatible(BiOpNode.OpType.LESS, boolNode, boolNode));
assertFalse(Types.isCompatible(BiOpNode.OpType.LESS, textNode, boolNode));
assertFalse(Types.isCompatible(BiOpNode.OpType.LESS, intNode, boolNode));
assertFalse(Types.isCompatible(BiOpNode.OpType.LESS, textNode, textNode));
assertTrue(Types.isCompatible(BiOpNode.OpType.LESS, intNode, intNode));
assertTrue(Types.isCompatible(BiOpNode.OpType.LESS, realNode, realNode));
assertFalse(Types.isCompatible(BiOpNode.OpType.LESS, setBool, setBool));

assertFalse(Types.isCompatible(BiOpNode.OpType.GRTR, boolNode, boolNode));
assertFalse(Types.isCompatible(BiOpNode.OpType.GRTR, textNode, boolNode));
assertFalse(Types.isCompatible(BiOpNode.OpType.GRTR, intNode, boolNode));
assertFalse(Types.isCompatible(BiOpNode.OpType.GRTR, textNode, textNode));
assertTrue(Types.isCompatible(BiOpNode.OpType.GRTR, intNode, intNode));
assertTrue(Types.isCompatible(BiOpNode.OpType.GRTR, realNode, realNode));

assertFalse(Types.isCompatible(BiOpNode.OpType.GRTR_EQL, boolNode, boolNode));
assertFalse(Types.isCompatible(BiOpNode.OpType.GRTR_EQL, textNode, boolNode));
assertFalse(Types.isCompatible(BiOpNode.OpType.GRTR_EQL, intNode, boolNode));
assertFalse(Types.isCompatible(BiOpNode.OpType.GRTR_EQL, textNode, textNode));
assertTrue(Types.isCompatible(BiOpNode.OpType.GRTR_EQL, intNode, intNode));
assertTrue(Types.isCompatible(BiOpNode.OpType.GRTR_EQL, realNode, realNode));

assertFalse(Types.isCompatible(BiOpNode.OpType.LESS_EQL, boolNode, boolNode));
assertFalse(Types.isCompatible(BiOpNode.OpType.LESS_EQL, textNode, boolNode));
assertFalse(Types.isCompatible(BiOpNode.OpType.LESS_EQL, intNode, boolNode));
assertFalse(Types.isCompatible(BiOpNode.OpType.LESS_EQL, textNode, textNode));
assertTrue(Types.isCompatible(BiOpNode.OpType.LESS_EQL, intNode, intNode));
assertTrue(Types.isCompatible(BiOpNode.OpType.LESS_EQL, realNode, realNode));

assertTrue(Types.isCompatible(BiOpNode.OpType.DBL_EQLS, boolNode, boolNode));
assertTrue(Types.isCompatible(BiOpNode.OpType.DBL_EQLS, textNode, textNode));
assertTrue(Types.isCompatible(BiOpNode.OpType.DBL_EQLS, intNode, intNode));
```

```
assertTrue(Types.isCompatible(BiOpNode.OpType.DBL_EQLS, intNode, realNode));
assertFalse(Types.isCompatible(BiOpNode.OpType.DBL_EQLS, textNode, boolNode));
assertFalse(Types.isCompatible(BiOpNode.OpType.DBL_EQLS, intNode, boolNode));
assertFalse(Types.isCompatible(BiOpNode.OpType.DBL_EQLS, textNode, boolNode));

assertTrue(Types.isCompatible(BiOpNode.OpType.NOT_EQLS, boolNode, boolNode));
assertTrue(Types.isCompatible(BiOpNode.OpType.NOT_EQLS, textNode, textNode));
assertTrue(Types.isCompatible(BiOpNode.OpType.NOT_EQLS, intNode, intNode));
assertTrue(Types.isCompatible(BiOpNode.OpType.NOT_EQLS, intNode, realNode));
assertFalse(Types.isCompatible(BiOpNode.OpType.NOT_EQLS, textNode, boolNode));
assertFalse(Types.isCompatible(BiOpNode.OpType.NOT_EQLS, intNode, boolNode));
assertFalse(Types.isCompatible(BiOpNode.OpType.NOT_EQLS, textNode, boolNode));

assertFalse(Types.isCompatible(BiOpNode.OpType.DBL_EQLS, listInt, setBool));
assertTrue(Types.isCompatible(BiOpNode.OpType.DBL_EQLS, listInt, listInt));

}

@Test
public void checkTypeHasMethodTest() {
assertTrue(Types.checkTypeHasMethod(setBool, "add"));
assertTrue(Types.checkTypeHasMethod(setBool, "clear"));
assertTrue(Types.checkTypeHasMethod(setBool, "contains"));
assertTrue(Types.checkTypeHasMethod(setBool, "containsAll"));
assertTrue(Types.checkTypeHasMethod(setBool, "isEmpty"));
assertTrue(Types.checkTypeHasMethod(setBool, "iterator"));
assertTrue(Types.checkTypeHasMethod(setBool, "remove"));
assertTrue(Types.checkTypeHasMethod(setBool, "removeAll"));
assertTrue(Types.checkTypeHasMethod(setBool, "size"));
assertFalse(Types.checkTypeHasMethod(setBool, "length"));
assertFalse(Types.checkTypeHasMethod(setBool, "replace"));
assertFalse(Types.checkTypeHasMethod(setBool, "hasNext"));
assertFalse(Types.checkTypeHasMethod(setBool, "next"));

assertTrue(Types.checkTypeHasMethod(listInt, "add"));
assertTrue(Types.checkTypeHasMethod(listInt, "clear"));
assertTrue(Types.checkTypeHasMethod(listInt, "get"));
assertTrue(Types.checkTypeHasMethod(listInt, "iterator"));
assertTrue(Types.checkTypeHasMethod(listInt, "size"));
assertTrue(Types.checkTypeHasMethod(listInt, "sort"));
assertFalse(Types.checkTypeHasMethod(listInt, "next"));
assertFalse(Types.checkTypeHasMethod(listInt, "isEmpty"));
assertFalse(Types.checkTypeHasMethod(listInt, "containsAll"));

assertTrue(Types.checkTypeHasMethod(textNode, "length"));
assertTrue(Types.checkTypeHasMethod(textNode, "replace"));
```

```
assertTrue(Types.checkTypeHasMethod(textNode, "tokenize"));
assertFalse(Types.checkTypeHasMethod(textNode, "isEmpty"));
assertFalse(Types.checkTypeHasMethod(textNode, "peek"));
assertFalse(Types.checkTypeHasMethod(textNode, "size"));
assertFalse(Types.checkTypeHasMethod(textNode, "add"));

assertTrue(Types.checkTypeHasMethod(iterNode, "next"));
assertTrue(Types.checkTypeHasMethod(iterNode, "hasNext"));
assertTrue(Types.checkTypeHasMethod(iterNode, "peek"));
assertFalse(Types.checkTypeHasMethod(iterNode, "replace"));
assertFalse(Types.checkTypeHasMethod(iterNode, "size"));
assertFalse(Types.checkTypeHasMethod(iterNode, "clear"));
}

@Test(expected = TypeMismatchError.class)
public void testGetHigherNumericType2() throws TypeMismatchError {
Types.getHigherNumericType(boolNode, setBool);
}

}
```