Project Compiler - Parser

Out: 23 Nov 2011
Helpsession: 28 Nov 2011 (time and location TBA)
Parser Due: 2 Dec 2011, 11:59PM
Compiler Due: 16 Dec 2011, 11:59PM

0 Motivation

A compiler is a program that translates a high-level language into assembly code. Some common examples are `javac` (for Java), `cc` and `gcc` (for C), and `CC` and `g++` (for C++).

You just wrote Life and Maze in assembly. It's possible that you would prefer never to write anything in assembly again. A compiler would help to facilitate this.

More plainly, a compiler makes it a lot easier for you to turn your program ideas into machine code.

1 Requirements

- Modify the provided Java code such that it realizes the described specification.
- Hand in your fully functional parser by the date at the top of this handout.
- Hand in test programs that FULLY test your parser (including all arithmetic operations and block statements). We cannot stress enough the necessity of thoroughly testing your parser to ensure it works correctly.
- Your handin must include a `README` file that
  - describes the bugs in your parser (or asserts their absence)
  - lists any changes made to the support code (i.e. classes added or edited)
  - explains your test program(s)
  - explains anything else the grader needs to know

2 Details

This project has been broken into several parts, each with a separate handout. Although all parts of the handout have been made available to you, you need only read the handout associated with each part to complete it.

Note that this project is a very large one. Be sure to plan your time accordingly.

You will be creating a compiler that reads a program written in the made-up procedural language Blaise and outputs a corresponding MIPS assembly file.

Your compiler should be airtight. *There should be absolutely no way for us to make it crash.* It should report an error and *gracefully* exit when given bad input. Further, your compiler should

refuse to generate code when given invalid input; a compiler that produces incorrect assembly or produces assembly for invalid code is even more dangerous than a compiler that crashes. Your error messages should be explicit and, whenever possible, include the line number on which the error was encountered. They should help someone trying to write code in Blaise to understand why their code was unable to compile.

2.1 How to compile

All the information you need to know to build this compiler has been covered in lecture, so refer to the slides and your notes for more details.

Your compiler will, at the highest level, read a text file containing Blaise source code and write a text file containing MIPS assembly code. This process can be seen as four subprocesses:

- **Scanning.** Scan the text input file. In practice, this means chunking the input into the smallest meaningful groups (called "tokens"), which are much easier to handle than individual characters or entire lines. In this step, we reject input that is not lexically correct. (This is handled in the support code).
- **Parsing.** Build a syntax tree from the stream of tokens generated by the scanner, rejecting input that is not syntactically correct.
- **Semantic analysis.** Ensure that the syntax tree built by the parser is internally consistent, or "makes sense." We reject semantically broken input that would make it impossible to generate code.
- **Code generation.** Using the parse tree produced earlier, generate corresponding assembly code (since we're now assured that it's possible to do so).

The first two subprocesses happen in tandem: you scan the file in small pieces and build the syntax tree recursively as you scan. The final two subprocesses happen in sequence.

In this portion of the assignment, you will write a parser for the Blaise language. Given some Blaise code, your parser should try to build an Abstract Syntax Tree (AST) to be used in the later stages of compilation. If you reach code that violates the grammar below, you should report the error and terminate - you do not need to continue parsing and report other errors.

2.2 The Blaise grammar

Below you will find the grammar that defines the way the Blaise language works. Please be sure you understand what these rules all mean and how they fit together to form a cohesive (if small) programming language. This is crucial to being able to build your parser.

The grammar is composed of terminals and rules.

The Blaise rules are only a small subset of rules that make up more powerful languages like C and Java, but they are still enough to make this a useful assignment. The table that follows defines the rules of the Blaise language. Anything that appears in `typewriter` font is a character (or string of characters) that should appear in the code exactly as written (such as `while`). Any place whitespace is allowed, you may use as much as you like.

The rule $\{X ::= Y\}$ means that anywhere you have a part of the language X you can replace it with the part of the language Y. Note that some rules are of the form $\{X ::= \ \}$. This means that the part of the language X can be replaced by emptiness, or nothing.

This is an LL(1) grammar. For more information on LL grammars, see the Compiler I lecture. In short, this means that you will need only one token at a time in order to determine which rule of the grammar you will follow.

Blaise Terminals

ID	*any string of characters that identifies a variable or a procedure, such as* `foo` *or* `bar`
INTEGER	*a non-negative integer, such as* `12` *or* `5` *or* `0`

(Continued on next page)

Blaise Rules

⟨Program⟩	::=	⟨ListVar⟩ ⟨ListProc ⟩
⟨ListVar⟩	::=	
⟨ListVar⟩	::=	⟨VarDecl⟩ ⟨ListVar ⟩
⟨VarDecl⟩	::=	`var int` ID ⟨DeclSfx⟩ `;`
⟨DeclSfx⟩	::=	
⟨DeclSfx⟩	::=	`[` INTEGER `]`
⟨ListProc⟩	::=	
⟨ListProc⟩	::=	`procedure int` ID `(` ⟨ProcArgs⟩ `)` ⟨Instr⟩ ⟨ListProc⟩
⟨ProcArgs⟩	::=	
⟨ProcArgs⟩	::=	`int` ID ⟨ProcArgsSfx⟩
⟨ProcArgsSfx⟩	::=	
⟨ProcArgsSfx⟩	::=	`, int` ID ⟨ProcArgsSfx⟩
⟨Instr⟩	::=	⟨Decl ⟩
⟨Instr⟩	::=	ID ⟨IDInstrSfx⟩ `;`
⟨Instr⟩	::=	`if (` ⟨Expr⟩ `)` ⟨Instr⟩ ⟨ElseInstr⟩ `endif`
⟨Instr⟩	::=	`while (` ⟨Expr⟩ `)` ⟨Instr⟩
⟨Instr⟩	::=	`output (` ⟨Expr⟩ `) ;`
⟨Instr⟩	::=	`input (` ID ⟨IDSfx⟩ `) ;`
⟨Instr⟩	::=	{ ⟨ListInstr⟩ }
⟨Instr⟩	::=	`return` ⟨Expr⟩ `;`
⟨IDInstrSfx⟩	::=	⟨IDSfx⟩ `:=` ⟨Expr⟩
⟨IDInstrSfx⟩	::=	`(` ⟨ProcCallArgs⟩ `)`
⟨ListInstr⟩	::=	
⟨ListInstr⟩	::=	⟨Instr⟩ ⟨ListInstr⟩
⟨IDSfx⟩	::=	
⟨IDSfx⟩	::=	`[` ⟨Expr⟩ `]`
⟨Decl⟩	::=	`int` ID ⟨DeclSfx⟩ `;`
⟨ElseInstr⟩	::=	`else` ⟨Instr⟩
⟨ElseInstr⟩	::=	
⟨Expr⟩	::=	⟨Factor⟩ ⟨ExprSfx⟩
⟨ExprSfx⟩	::=	
⟨ExprSfx⟩	::=	`+` ⟨Expr⟩
⟨ExprSfx⟩	::=	`-` ⟨Expr⟩
⟨ExprSfx⟩	::=	`*` ⟨Expr⟩
⟨ExprSfx⟩	::=	`/` ⟨Expr⟩
⟨ExprSfx⟩	::=	`%` ⟨Expr⟩
⟨ExprSfx⟩	::=	`<` ⟨Expr⟩
⟨ExprSfx⟩	::=	`<=` ⟨Expr⟩
⟨ExprSfx⟩	::=	`>` ⟨Expr⟩
⟨ExprSfx⟩	::=	`>=` ⟨Expr⟩
⟨ExprSfx⟩	::=	`==` ⟨Expr⟩
⟨ExprSfx⟩	::=	`&&` ⟨Expr⟩
⟨ExprSfx⟩	::=	`||` ⟨Expr⟩
⟨Factor⟩	::=	ID ⟨IDFactorSfx⟩
⟨Factor⟩	::=	INTEGER
⟨Factor⟩	::=	`(` ⟨Expr⟩ `)`
⟨Factor⟩	::=	`!` `(` ⟨Expr⟩ `)`
⟨Factor⟩	::=	`true`
⟨Factor⟩	::=	`false`
⟨IDFactorSfx⟩	::=	⟨IDSfx⟩
⟨IDFactorSfx⟩	::=	`(` ⟨ProcCallArgs⟩ `)`
⟨ProcCallArgs⟩	::=	
⟨ProcCallArgs⟩	::=	⟨Expr ⟩ ⟨ProcCallArgsSfx⟩
⟨ProcCallArgsSfx⟩	::=	
⟨ProcCallArgsSfx⟩	::=	`,` ⟨Expr ⟩ ⟨ProcCallArgsSfx⟩

Although comments are not formally part of the Blaise grammar, you may write them in your Blaise code. Anything in a line following the `#` character will be treated as a comment. This is handled for you by the tokenizer in the support code, so you need not worry about it in your compiler implementation.

2.2.1 Notes

Order of operations: No order of operations is specified in the grammar (unlike other languages; for example, Java gives grouping priority to AND over OR and priority to times over plus). The natural way to write programs in the absence of order of operations is left-to-right. However, it is much simpler to parse the expressions right-to-left (can you see why?). The programmer can use parentheses to enforce standard order of operations. With this in mind, we require that you parse arithmetic expressions right-to-left. **Failure to adhere to this standard will cause the test cases used for grading your compiler to fail.** For example,

```
procedure int main() {
  int a;

  a := 5 - 4 - 3;
  output(a);
}
```

will output `4`, not `-2`.

Input and output: The `input` and `output` commands let you perform rudimentary IO on integers by generating syscalls. `input` will pop up a dialog box to prompt the user for a non-negative integer, then puts that value in the given variable or array element. `output` prints the value of the given expression to the Mipscope console. Note that, although these look like procedure calls, they are not and have their own place within the grammar.

Blocks: A block is a group of instructions that are grouped together inside { and } braces. In Blaise, each block will create a new scope. The `if`, `else`, and `while` constructs *implicitly* create blocks (just as { and } explicitly create them).

However, implict blocks should be represented differently in the parse tree than explicit ones. Explicit blocks should use a `NodeInstrBlock` whereas implicit ones should not. In other words, an implicit block of code should not be represented with a `NodeInstrBlock` in the parse tree.

2.2.2 Blaise examples

We have provided several fully functional examples of working Blaise code in the stencil. Nevertheless, these examples are far from exhaustive. You should be creating your own test cases for this part (and all parts) of the assignment, testing both valid and invalid input.

2.3 Support

We're providing extensive support code that relies heavily on polymorphism. If object-oriented design concepts aren't so fresh in your mind, you may want to refer to the CS 15 or CS 18 lecture slides (or your preferred reference on OOP).

We encourage you to look through most of the support code; it's a great way to get a feel for the methods you actually have to write for this assignment. Most of your job will be simply filling in the methods in the given classes. In this portion of the assignment, you will be modifying the `Parser` class.

Don't feel that you have to read and understand the support code in its entirety. We provide you with the full source to the support code only because

- it provides you with a reference, filling in gaps in your understanding that might be left by lecture material and the handouts
- if you are pursuing extra credit, you will likely need to modify it
- it might help to satisfy your curiosity

2.3.1 Installing the project

Run `cs031_install comp` to install the project. This will give you a whole bunch of .java files, an ant build file, and some Blaise code examples.

By default, everything is installed to `~/course/cs031/comp`.

2.3.2 Class overview

Most of the design work is done for you. The following is an overview of the types of classes that the design includes that you will need in this portion of the assignment. An asterisk indicates many classes starting with the given prefix.

- **Compiler** This class contains a `main` method that you will need to add to. It already contains some code necessary for error-handling and dealing with command-line arguments (the input and output filenames).
- **Token*** These classes implement the scanning subprocess by taking an input file, scanning it for meaningful sequences of characters, and returning tokens representing the sequences. This has been done for you.
- **Parser** This is the main parsing class. It holds a `Tokenizer` and otherwise does not have much data of its own. The only method called from outside the `Parser` is `parse()`. The utility methods of this class have been written for you. We have provided a little bit of code in the `parse()` method to get you started; you will have to add in more methods to help out in the parsing.
- **Node classes** These nodes of the syntax tree all extend the abstract `Node` class. They will be used to build the tree in your parser. There are a lot of different subclasses, but they can be grouped by name and used as follows:

  - **NodeProgram** An instance of this is at the root of your syntax tree. It has two children, a list of global variables and a list of procedures.
  - **NodeList*** Utility classes that form a LISP-style linked list of something, whether it's instructions, procedures, arguments, or global variables. Each Cons class has as its left child one item, and as its right child either another Cons (with another left child) or a Nil (indicating the end of the list).
  - **NodeInstrDecl*** Local declarations of single variables and arrays. These do not have children. They are a type of instruction.
  - **NodeVarDecl*** Global declarations of single variables and arrays. These do not have children. They are **NOT** a type of instruction.
  - **NodeInstr*** Nodes for imperative statements that do **not** evaluate to a value and typically do have side effects (contrast this to expressions, which do evaluate to some value). Examples: flow-of-control statements (like if/then/else) and assignments. Many of them have children that are expressions or other instructions, forming a tree rooted at any given instruction.
  - **NodeArithmeticExpr** An abstract base class for all arithmetic expressions (of semantic type Integer).
  - **NodeExpr*** A `NodeExpr` represents an Expression – is a construct that evaluates to something. You can view it just in terms of what it evaluates to, regardless of what other procedures are involved in calculating this value. Some expressions are recursive, taking one or more expressions as children. For instance, an addition expression takes two subexpressions and evaluates to their sum. Some are leaf nodes and evaluate to a value immediately. For instance, an integer-literal expression just evaluates to the value of the integer that it stores.
  - **NodeRelation*** `NodeRelation`s are types of `NodeExpr`s that express boolean relations (NodeRelation* classes are subclasses of `NodeExpr`).
  - **NodeProc** A class for procedure declarations. These have as children a list of arguments and the instruction associated with the procedure.
  - **NodeProcCallArg** A class for the value passed to a procedure as an argument. It has as a child the expression representing this value.
  - **NodeProcDeclArg** A class for a formal argument in a procedure declaration. No children.

- **Exception*** A Blaise program can contain several different kinds of errors, and they will be detected at different places in your compiler. An important part of your work for this assignment will be to realize all the possible errors, figure out how to detect them, and handle them appropriately. When you discover an error, describe the error and terminate by simply throwing an instance of one of these exception classes. The exception will propagate up the Java stack until it is caught. However, you must make sure that you catch the exception somewhere!

2.4 Scanning

Scanning a text file entails breaking up the sequences of characters which constitute the code into a series of tokens. This is all implemented for you in the `Tokenizer` class.

This does not mean that you should ignore the details of this class! Your code depends on understanding the meaning of `Tokenizer`'s tokens (e.g. `TokenType.ID`, `TokenType.INTEGER`, etc.). So read the documentation for this class and test it with sample code in order to understand its functionality. Note that you may need to advance the token stream twice before accessing the first token depending on how you advance your token stream.

2.5 Parsing

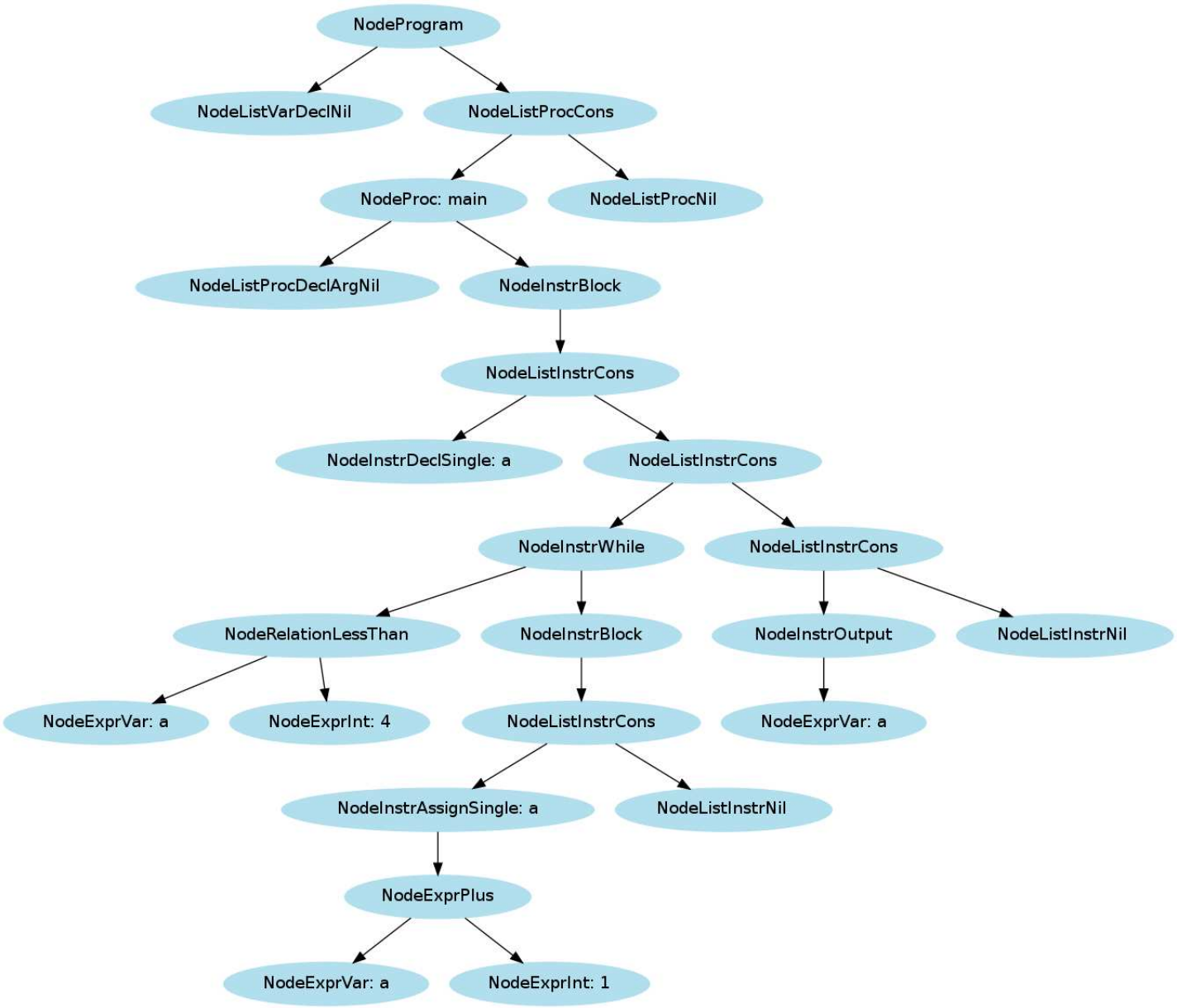
Once you understand the tokenizer, you are ready to start building the abstract syntax tree (see lecture slides if you need to refresh on AST's). Your tree is composed of lots of nodes (see the `Node` classes) and is constructed by the `Parser` class.

Most of the node classes are self-explanatory, but a few deserve special attention. A `NodeProgram` will be the root node of your parse tree. It has two children: a `NodeListVarDecl`, representing the entire list of global variables, and `NodeListProc`, representing the list of procedures.

NodeList classes are abstract; you will only use their subclasses. For example, `NodeListInstr` has subclasses `NodeListInstrCons` and `NodeListInstrNil`. The cons and nil node types do not correspond to elements of the grammar; rather, they are structural nodes used to chain together the declaration and instruction nodes.

To help you understand how the node classes fit together, consider the following Blaise program and its parse tree (on the following page).

```
procedure int main() {
  int a;
  while (a < 4) {
    a := a + 1;
  }
  output (a);
}
```



To make your task easier, we have written several convenience methods for you in Parser.java. You are free to use (or not use) these utility methods. You can find out how they work by reading the comments in Parser.java.

You will have to do some error-checking as you build the tree. If you come across a syntax error (e.g. "semicolon expected"), you should throw a `SyntaxException` (whose constructor takes a line number and a message; your compiler should catch the exception, print out the line number and message, and exit gracefully). Call `currentLine()` on the tokenizer to get the current line. Note that sometimes currentLine() will not return the correct line number. Don't panic. This is not your fault.

2.6 Compiling and running your code

From inside your compiler directory, type `ant compile` to compile your code.

To run your compiler, type `ant run`. This will execute Compiler.java. It will read the file 'input.bl' and try to create as output the file 'output.mal'. If you want to change these arguments, see the ANT_README file in the stencil for an explanation.

To clean up and tidy all your files, run `ant clean`.

2.7 Using Eclipse

If you would rather use Eclipse than ant, you can configure your project in Eclipse by doing the following:

- Make a new Java Project and import the code (you can use *General* > *Filesystem* to do so)
- Go to *Run* > *Run Configurations*
- Right click on *Java Application* and click *New*
- Under the *Main* tab fill-in or browse for your project, and then search for your main class (this will be your Compiler class)
- Now go to arguments and enter the input and output files as you would on the command line.
- To run your project you can either find it in the *Run* menu or select it from the drop down by the green-and-white play button on the toolbar.
- Finally, if for some reason you have to set up debugging (you shouldn't have to because it should have happened automatically after setting up your run configuration), follow the same process, replacing *Run Configurations* with *Debug Configurations*.

2.8 Running the demo

We will be providing you with a demonstration Blaise compiler, so you can get an example of how output could look. It will also generate a sample parse tree so you can see what they should look like. You can run the demo by typing
`compiler-demo` *`blaise input file MIPS output file`*

2.9 Modifying the support code

You should not modify the support code. The ONLY exception to this rule is that you may modify `MIPSCodeGenerator`.

In particular, there must be no sharing of data between the parser, semantic visitor, and code generation phase beyond what the stencil passes in the `main` method of the `Compiler` class.

All changes to `MIPSCodeGenerator` must be documented both inline and in the README.

3 Grading

This and all portions of your program will be graded primarily based on its adherence to and fulfillment of the assignment specifications. However, your work will also be evaluated in terms of the quality of your commenting of your Java code, your adherence to some reasonable coding conventions, your avoidance of blatantly inefficient techniques, and the quality of your README file. Please note that your commenting and your README are most important when a specific part of your program does not function properly; these two tools will allow the TAs to assign partial credit as appropriate.

Unlike with Maze, where all functionality points were based on the quality of your final handin, the code you submit for each milestone will be graded once the project is due.

4 Handing in

To hand in, type `cs031_handin comp_parser`. It will prompt you for confirmation and then hand in your entire current working directory.

If you have successfully handed in, you should get an email saying such. (This email cannot be used as evidence that you handed in, but it is an indication to you that we received your files.)

Your README should be called `README`.

Please make sure you don't hand in any hidden eclipse files, as they take up a lot of space.

If you want to change your handin, just run the script again and the new handin will supersede the old handin (we actually won't see anything but the most recent one).

After the handin deadline, we will release our Parser for you to use, so that bugs in your version do not affect your ability to do the later parts of the assignment.

Note that we **will** accept late handins for this portion of the assignment and that you may use your late days.

5 Final words

Demo vs. spec If there's a discrepancy between the behavior of the demo and the specifications, you should ignore the demo and follow the spec. You should also email us if this happens so we can fix the demo!

Error checking Make sure that your compiler checks for errors and gracefully handles such situations. Your compiler as a whole **MUST** throw only Lexical, Syntax, and Semantic exceptions (the next handout covers Semantic exceptions). Be sure to catch these and print out the messages of these exceptions somewhere! Be sure as well that your printed messages begin with either "LexicalException", "SyntaxException", or "SemanticException", depending on what type of exception is being thrown. **You will lose points if you do not do this.**

Commenting Although you should already be in this habit, be sure to comment any code that you modify or write. In particular, we expect good inline comments for whatever you write. Any methods you write or modify, **including method stubs we provide**, should be sufficiently commented so that we can tell what they should do.

Test code The examples that we provide in the stencil clearly do not test all language constructs or boundary cases. You should (as usual) write your own test code.

Write good test code that forces your compiler to use all of the language constructs contained in Blaise. Blaise isn't a big language, so this shouldn't take too much time, and it will be an invaluable debugging tool.

Remember to test subtle cases. Make sure you test that you are labeling errors correctly, or a potential user of your compiler may be confused by the output.

Visualizing the parse tree We have given you support code that will allow you to generate a graphical representation of the abstract syntax tree. In the main method in `Compiler.java` you will see the line:
`viewParseTree(program)`
This line will create the file `graph.png`. To view png files, try the `eog` command. This can be a useful tool in debugging your parsing - take advantage of it!

README If you're unsure about whether something should be in your README, it probably should be. Don't write a novel, but include answers to all the questions a TA might have while grading your program and looking at your code.

Note that you should get in the habit of handing in a README with every project you hand in. Many upper-level classes will expect this, whether or not it's asked for explicitly.