SFU CMPT 379
Compilers
Fall 2015
Assignment 1

Assignment due Tuesday, October 6, by 11:59 pm.

For this assignment, you are to convert a compiler I have provided into a compiler that works for an expanded language.

Included with this assignment are brief descriptions of the languages Grouse-0 and Grouse-1. The base compiler that I have provided takes Grouse-0 programs as input and produces an intermediate code called Abstract Stack Language. The base compiler and the Abstract Stack Machine emulator can be found in a subversion repository at https://punch.cs.sfu.ca/svn/CMPT379-Fall-2015

All development is to be done in Java and must work on the eclipse platform. I therefore recommend using eclipse for your development work. It's powerful, and it's free: it can be found at **www.eclipse.org.** I use the "Subversive" package for eclipse to handle interactions with subversion repositories (there is another subversion package that seems to work equally well; which one you use is not important). I **require** that you use your own repository (using whatever tool you like) to hold your assignment work. Keep your repository on a different machine or disk drive than your development machine or drive. Be sure to commit **at least** once each day that you work on your compiler, and preferably more often than that.

On the next page I have listed several checkpoints for completing the assignment. Checkpoints will not be marked, but I do expect you to follow the order of checkpoints given. Some checkpoints contain requirements and important notes. Note all of the preliminaries necessary to begin work on the compiler proper.

Be sure you **test** each feature as you complete it. Do not go on to further items without completing and testing what you have already done. Your testing may be as structured as unit tests, component tests, regression tests (a very few sample tests are provided with the base compiler) or may be as simple as manually checking the output of a phase for correctness on a few examples.

Note that several **applications that print out the results of various phases** of the compilation process are also provided with the base compiler (see checkpoint 1f). These are useful for testing and debugging.

Submit a zipped version of your eclipse project (minus the object files and executables; we will recompile your project). **Project setups for other compilers/IDEs are not accepted.** We will test your compiler using the "GrouseCompiler" application in your "applications" package; ensure it runs your compiler and that it takes a single filename argument. It must not produce an output file (and remove the output file if it already exists) if there are any errors in the input. The compiler provided does this already; my advice is to not make any functional change to GrouseCompiler.java.

Assignment 1 Checkpoints

1a. Download the base compiler. (You may need to download eclipse, and/or a subversion package for eclipse if you plan to use it.)

1b. Disconnect the base compiler from the course repository and commit it to your own repository. (You may need to create a repository in order to do this.)

1c. Ensure that the -ea flag is used for all java executions. This argument enables assertions in java. If you use eclipse to develop, navigate to Window>Preferences>Java>Installed JREs, select your JRE, and edit it to add "-ea" to the default arguments. (See also Window>Preferences>Java>JUnit; there is a checkbox for adding -ea to all jUnit runs.) You can also set the –ea flag on the "arguments" tab when editing a run configuration. Test to ensure that exceptions are enabled by placing an "assert false;" statement somewhere it will be executed, such as in GrouseCompiler in the applications package. Run the program and see if the assert triggers.

1d. Run the tests that come with the compiler—in eclipse, select the project's src/applications/tests directory and execute "Run As...", choosing jUnit Test (alt-shift-x t). Verify that the tests all pass. Find the test files in the project (there aren't many of them) and see what they do. There are a very few unit tests and some major component regression tests. I also test by compiling Grouse files and then running them through the ASM emulator.

1e. Study the file asmCodeGenerator/codeStorage/ASMOpcode.java until you understand the operation of the Abstract Stack Machine (we will also discuss this machine in lecture). A few sample ASM programs are provided in the "ASM_Emulator" directory of the project; examine them. Write and run several ASM programs, to ensure that your understanding of the machine is correct. You cannot write a compiler if you don't understand the target machine.

1f. Run your ASM program(s) by using ASMEmu.exe, located in the "ASM_Emulator" directory of the project. The emulator takes one argument, which is the filename of an ASM file. You can run ASMEmu from the command line or as an "External Tool" in eclipse (the run button with the red suitcase). You can also find two ASMEmu .launch files in the runConfigurations folder of the project. They should appear in your "External tools" menu, from which you can copy and edit them using "External tools configurations..." The launch "ASM Me!" requires that you select the .asm file that you want to run before invocation.

1g. You may also run your programs using ASM_Simulator.jar, also located in the "ASM_Emulator" directory of the project. This simulator performs a very close approximation to what ASMEmu.exe does. It may be useful for stepping through your program's execution. The simulator was a student project and I do not provide support for it. Unfortunately, the simulator is **not exactly the same** as ASMEmu, and ASMEmu is what your programs will be tested with, so **do not** use the simulator as your only testing environment. I will not accept excuses of the form "…but it works on the simulator!"

1h. Explore the "applications" package and see what each application does; there are applications provided for printing the output of most phases of the compiler. Write a Grouse-0 program (to keep things orderly, put it in the project's "inputs" folder); run this program through GrouseCompiler and examine and/or run the corresponding .asm file generated in the "outputs" folder. The compiler expects that the current directory is the project directory when it is running; it places its output in ./outputs/<basename>.asm, where <basename> is the basename of the input file. (for example, if the input is "inputs/firstProg.Grouse", the basename is "firstProg" and the compiler writes a file "./output/firstProg.asm".)

1i. Break the assignment down into several features that can be added one-at-a-time to the compiler. For instance, "comments", "floating type", and "comparison operators" are each features (and there are several others).

1j. Implement each feature in turn. You decide on the order of implementation, but start with something simple like "comments". Implement a feature end-to-end in the compiler (through all the phases and different cases it may involve) and test before moving on to the next feature. Failure to implement in this fashion typically results in a spotty, buggy compiler (and correspondingly lower marks).

Language Grouse-0

Whitespace can be used to separate tokens, but is not necessary if the text is unambiguous.

Tokens:

integerConstant → [**0..9**]$^{+}$ // has type "integer"
booleanConstant → **true** | **false** // has type "boolean"

identifier → [**a..z**]$^{+}$
punctuator → *operator* | *punctuation*
operator → **+** | ***** | **>**
punctuation → **;** | **,** | **{** | **}** | **:=**

Grammar:

S → **main** *block*
block → **{** *statement** **}**

statement → *declaration*
printStatement

declaration → **imm** *identifier* **:=** *expression* **;** // immutable (constant) value
// identifier gets the type of the expression.

printStatement → **print** *printExpressionList* **;** // print the expr values
printExpressionList → *printExpression**
printExpression → *expression*$^{?}$ **,**$^{?}$ **nl**$^{?}$

expression → *expression operator expression* // all operations left-associative
literal

literal → *integerConstant* | *booleanConstant* | *identifier*

Any word (sequence of roman letters) shown in bold on the specification above is a keyword and cannot be used as an identifier. Identifiers must be *declared* (appear as the identifier in a declaration) before they are used as a literal. (They are only considered declared after the end semicolon of their declaration.)

In a print statement, the appearance of an *expression* means that the value of the expression is printed. The appearance of a comma means that a space is printed. The appearance of **nl** means that a newline is printed. The statement

print 3,, 4, nl

prints a 3, then two spaces, then a 4, then a space, then a newline.

The operands in an expression must both be of integer type. The operators provided do not take any boolean operands.

The result of "*expression* **>** *expression*" is boolean, and the results of "*expression* **+** *expression*" and "*expression* ***** *expression*" are integer.

Language Grouse-1

Whitespace can be used to separate tokens, but is not necessary if the text is unambiguous.

Tokens:

integerConstant → **-**$^?$ [**0..9**]$^+$ // has type "integer"

floatingConstant → **-**$^?$ ([**0..9**]$^+$ **.** [**0..9**]$^+$) (**e** (**+**|**-**)$^?$ [**0..9**]$^+$)$^?$ // has type "floating"

booleanConstant → **true** | **false** // has type "boolean"

stringConstant → " [^"**\n**]* " // **\n** denotes newline. has type "string"

characterConstant → **'*x*** // ***x*** is any printable ascii character (decimal 32 to 126)
// starts with single-quote. **'*x*** has type "character"

identifier → [**a..zA..Z_**][**a..zA..Z_~0..9**]* //at most 32 characters in an identifier

punctuator → *operator* | *punctuation*
operator → *arithmeticOperator* | *comparisonOperator*
arithmeticOperator → **+** | **–** | ***** | **/**
comparisonOperator → **<** | **<=** | **==** | **!=** | **>** | **>=**
punctuation → **;** | **,** | **{** | **}** | **(** | **)** | **:** | **:=**

comment → **//** [^**/\n**]*(**/**[^**/\n**]$^+$)* (**//** | **/\n** |**\n**)

Grammar:

S → **main** *block*
block → **{** *statement** **}**

statement → *declaration*
letStatement
printStatement

declaration → **imm** *identifier* **:=** *expression* **;** // immutable "variable"
var *identifier* **:=** *expression* **;** // mutable variable

letStatement → **let** *target* **:=** *expression* **;** // Reassignment. target and expr must have same type.
target → *identifier* // identifier must have been declared with **var**

printStatement → **print** *printExpressionList* **;** // print the expr values
printExpressionList → *printExpression**
printExpression→ *expression*$^?$ **,**$^?$ **nl**$^?$

expression → *expression operator expression* // all operations left-associative
(*expression* **)**
expression **:** *type* // casting
literal

type → **bool** | **char** | **string** | **int** | **float** // boolean, character, string, integer, floating

literal → *integerConstant* | *floatingConstant* | *booleanConstant* | *characterConstant* |*stringConstant* | *identifier*

A comment starts with two consecutive slashes (**//**) and continues to the first place that another two consecutive slashes or a newline are encountered. (The starting two slashes and ending two slashes of a comment may not share a slash.) Comment tokens are removed from the token stream before parsing.

Any *word* (sequence of *letters*) shown in bold on the specification above is a keyword and cannot be used as an identifier.

The 32-character limit on identifier size does not end an identifier token at 32 characters. In lexical analysis, allow identifier lexemes to be of arbitrary length (keep adding characters as long as you get one of the identifier-continuing characters). After finding the entire identifier lexeme, issue a lexical error if it is longer than 32 characters.

Grouse-1 has five types: Boolean, character, string, integer, and floating (which is equivalent to **double**-precision in C++ or Java). The latter two are called *numeric types*. The number of bytes consumed by variables of the five types are 1, 1, 4, 4, and 8, respectively. You must use a format string of "%g" to print a float. Use "%c" for a character; do **not** print a quote character before each character. Use "%s" for a string; do not print the double-quotes around the string.

An identifier declared with **imm** or **var** is called a *variable*, even though those declared with **imm** do not vary. A variable declared with **imm** may also be called a *constant* or an *immutable*, and one declared with **var** may also be called a *mut* or *mutable*. Variables are initialized with the value of the expression in their declaration, and are given the type of that expression. Thereafter, they do not change type.

The value of a string expression, or a string variable, is a pointer (reference) to memory containing that string. Types whose variables contain pointers are called *reference types*.

Variables must be declared before they are used as a literal. (They are only considered declared after the end semicolon of their declaration.)

In a print statement, the appearance of an *expression* means that the value of the expression is printed (for reference types, the value of the referred-to object is printed). The appearance of a comma means that a space is printed. The appearance of **nl** means that a newline is printed. The statement

print 3,, 4, nl nl

prints a 3, then two spaces, then a 4, then a space, then two newlines. Do not omit any spaces or print extra spaces or newlines around an expression. The output of print statements is what is used to mark your project. Be sure to get them right!

An expression with an arithmetic operator must have either (integer, integer) operands or (floating, floating). They do not take mixed-type operands. Division (integer or floating) by zero yields a run-time error, which you must detect and report (and halt execution); do not allow the emulator to give the error. The result of an arithmetic expression has the type of its arguments. Integer division truncates the result towards zero.

An expression with a comparison operator takes the following types of operand pairs: either (character, character), (integer, integer), or (floating, floating). In addition, **==** and **!=** take (boolean, boolean) and (string, string) operands (for reference types, the pointers are compared, not the objects that are pointed-to). No comparison takes mixed-type operands. The result of a comparison is boolean.

Only certain casts (type conversions) are allowed. Booleans may not be cast to any other type. Characters may be cast to integers (they yield an integer between 0 and 127, inclusive). Integer may be cast to character (by using the bottom 7 bits of the integer as the character) and to floating. Floating may be cast to integer by truncation (rounding towards 0). Integer and character may be cast to boolean (zero yields false, nonzero yields true). Any type can be cast to itself. No other casts are allowed.

Operators have the following precedence:

parentheses have the highest precedence,
next highest is the casting operator **:** ,
/ and ***** have the next highest precedence,
- and **+** the next highest, and
the comparisons all have the lowest precedence.

When reporting a runtime error, the code **must** print the string "Runtime error" (capitalize *Runtime* but not *error*). It may print other details and whatever before or after this string. The code that marks your assignment specifically looks for this string in your output. If it is present, your code is judged to have issued an error. If it is not, your code is judged to have not issued an error.

Other language details will be discussed in class. Do not miss lecture!