SFU CMPT 379
Compilers
Summer 2023
Milestone 1

Milestone due Friday, June 9, 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 Tan-0 and Tan-1. The base compiler that I have provided takes Tan-0 programs as input and produces an intermediate code called Abstract Stack Language. The base compiler and the Abstract Stack Machine emulator can be downloaded from the course website.

All development is to be done in Java. I recommend using eclipse for your development work if you don't already have a favorite java environment. Eclipse is powerful, and it's free: it can be found at **www.eclipse.org**. I **require** that you use your own repository (using whatever tool you like) to hold your assignment work. Your repository must **not** be publicly accessible. 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.

The compiler that you develop includes my (Shermer's) intellectual property, and it also might be used by current or future students in the course to cheat. Therefore you are legally and academically prohibited from storing your project in a public repository or from otherwise making public (posting) your code, for now and forever. You may keep a **private** copy of your project to show employers as part of your portfolio. Note that other students, TAs, and professors know how to use various search engines, so keeping your project publicly in an obscure place is not effective. This applies to all of the milestones (assignments) in this course.

On the next page I have listed several checkpoints for completing the assignment. Checkpoints will not be marked, but I do recommend that you 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 very useful for testing and debugging—don't forget that they are there!

Submit a zipped version of your project (minus the object files and executables; we will recompile it). We will test your compiler using the "TanCompiler" application in your "applications" package; ensure that this applications runs your compiler, and that it takes a single filename argument. Your compiler must not produce an output file (**and remove the output file if it already exists**) if there are any errors in the input. (It is not acceptable to simply throw an exception, for instance.) The compiler provided does this already; my advice is to not make any functional change to the provided application(s).

Milestone 1 Checkpoints

1a. Download the base compiler. (You may need to download eclipse if you plan to use it.)

1b. Commit the base compiler to your 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 [I am using jdk-18.0.2.1, but any later jre should work]. 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 TanCompiler in the applications package. Run the program and see if the assert triggers.

1d. Run the tan tests that come with the compiler: inside the tan-S project folder, navigate to input/tan-0, left-click on "coins.tan" and right-click on "Run Configurations" and choose "TanCompileMe (tan-S)" This will create a file "coins.asm" in tan-S/output (**not** in tan-S/input/tan-0/output). Compare the coins.asm that you get with the one already in tan/input/tan-0/output; they should be the same (except maybe some Windows vs. Mac line-ending problems). You can do the same with all of the tan files in input/tan-0; the ones starting with "err_" should generate a user error. ("Compiler error" means an error in the compiler; "user error" means an error in the file being compiled.)

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 a few ASM programs, to ensure that your understanding of the machine (particularly the memory model, loading, and storing) is correct. You cannot write a compiler if you don't understand the target machine. **Take the time to learn it thoroughly**.

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 (with the mouse) 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 Tan-0 program (to keep things orderly, put it in the project's "inputs" folder); run this program through TanCompiler 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.tan", 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 Tan-0

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

Tokens:

integerLiteral → [**0..9**]+ // has type "integer"
booleanLiteral → **true** | **false** // has type "boolean"

identifier → [**a..z_**]+
punctuator → *unaryOperator* | *operator* | *punctuation*
unaryOperator → **–**
operator → **+** | ***** | **>**
punctuation → **;** | **{** | **}** | **:=**

Grammar:

S → **main** *mainBlock*
mainBlock → **{** *statement** **}**

statement → *declaration*
printStatement

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

printStatement → **print** *printExpressionList* **;** // print the expr values
printExpressionList → *printSeparator** (*expression printSeparator*+)* *expression*?
// a separated list of expressions (can be zero of them)
printSeparator → **\|** | **\n** | **\s**
expression → *unaryOperator expression* // only "**-**" implemented as unary
expression operator expression // all operations left-associative (binary "**-**" not implemented)
literal

literal → *integerLiteral* | *booleanLiteral* | *identifier*

Any word (sequence of roman letters and underscores) shown in bold in the grammar above is a keyword and cannot be used as an identifier. Also, **true** and **false** are keywords. 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 of their declaration statement.

In a print statement, the appearance of an *expression* in the printExpressionList means that the value of the expression is printed. The appearance of **\n** means that a newline is printed. The appearance of **\s** means that a space is printed. A ****, on the other hand, prints nothing.

print 3 \s\s 4 \ 5 \s\n;

prints a 3, then two spaces, then a 4, then a 5 (no space between 4 and 5), then a space, then a newline.

The operand of a unaryOperator must be of integer type. The operands in a binary expression must both be of integer type. The operators provided do not take any boolean operands. Booleans are stored as 0 (false) and 1 (true) in memory. (Do not store true values as anything other than 1.)

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

Language Tan-1

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

Tokens:

integerLiteral → [**0..9**]$^{+}$ // has type "integer"
floatingLiteral → ([**0..9**]$^{+}$ **.** [**0..9**]$^{+}$) ((**e** | **E**) (**+**|**-**) [**0..9**]$^{+}$)$^{?}$ // has type "floating"
booleanLiteral → **true** | **false** // has type "boolean"
stringLiteral → " [^"**\n**]* " // has type "string"
// In this *specification* only **\n** denotes the newline character.
// **(**tan-1 does not have character escapes in strings;
// **do not** interpret **$n**, **#n**, or **\n** in a string constant as a newline, for instance)
// In tan, only in print statements (beween arguments) is **\n** treated as newline.

characterLiteral → **'α'** |**%**[**0..7**] [**0..7**] [**0..7**] // has type "character"
// **α** is any printable ascii character (encoding is decimal 32 to 126)

identifier → [**a..zA..Z_@**][**a..zA..Z_@0..9**]*

punctuator → *operator* | *punctuation*
operator → *unaryOperator* | *arithmeticOperator* | *comparisonOperator*
unaryOperator → **+** | **–**
arithmeticOperator → **+** | **–** | ***** | **/**
comparisonOperator → **<** | **<=** | **==** | **!=** | **>** | **>=**
punctuation → **;** | **{** | **}** | **(** | **)** | **[** | **]** | **#** |**%**| **:=**

comment → **#** [^**#\n**]* (**#** |**\n**) // starts at # ends at next # or newline.
// a comment cannot start inside a string or character.

Grammar:

S → **main** *blockStatement*
blockStatement → **{** *statement** **}** // only the name of the nonterminal has changed.

statement → *declaration*
assignmentStatement
printStatement
blockStatement

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

assignmentStatement → *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 → *printSeparator** (*expression printSeparator*$^{+}$)* *expression*$^{?}$
// a separated list of expressions (can be zero of them)
printSeparator → ****| **\n** | **\s** | **\t** // **\t** prints a tab

expression → *unaryOperator expression*
expression operator expression // all operations left-associative
(*expression* **)**
< type **>(**expression**)** // casting
literal

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

literal → *integerLiteral* | *floatingLiteral* | *booleanLiteral* | *characterLiteral* | *stringLiteral* | *identifier*

A character literal can take two forms. The simplest form, **'α'**, like in java or C++, allows you to specify any printable ascii character. For instance, 'C' denotes an upper-case C character. **'"'** (three single-quotes) is permitted to denote the character ' (single quote). No character escapes (such as '\n') are allowed. The numeric form, **%**[**0..7**][**0..7**][**0..7**], allows you to specify any ascii symbol by giving its numeric code in octal (base-8). Three octal digits are required; use leading 0's if necessary. The octal codes for different characters are often found on tables of the ascii encoding. For instance, %040 denotes the space character.

An integer, floating, or character literal may have a numeric interpretation that exceeds what can be represented. If this is the case, you should report it as an error. For instance, an integer 1234567890987654321 is to large to be represented as an ordinary int, so it is reported as an error. 2E400 is also too large. Allowable characters are 0 through 127 (decimal). To detect an integer being too large, you'll have to catch a NumberFormatException thrown by Integer.parseInt (in NumberToken.java). To detect a floating-point number being too large, you'll have to check the return value of Double.parseDouble to see if it is Double.POSITIVE_INFINITY. To detect a character being too large, you'll have to both catch a NumberFormatException and check the return value of Integer.parseInt.

A comment starts with a hash symbol (**#**) and continues to the first place that another hash symbol or a newline is encountered. Comment tokens are not created, so comments are effectively 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.

Tan-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:

Boolean	1
Character	1
String	4
Integer	4
Floating	8

An identifier declared with **const** or **var** is called a *variable*, even though those declared with **const** do not vary. A variable declared with **const** 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 a record (chunk of memory) containing that string. This is why a variable of the string type consumes exactly 4 bytes (the size of a pointer, or integer, on the ASM). Types whose variables contain pointers are called *reference types*. The record for a string in Tan-1 is as follows:

3 (as a 4-byte integer)	9 (4 bytes)	Length (4 bytes)	Characters (*length* bytes)	Null character (1 byte)

The 3 in the first four bytes identifies this as a string record. The 9 in the next four bytes are status bits for this record; more detail on this will be provided on the next milestone. String records are immutable (they will not change).

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

Print statements are defined as in Tan-0, with the addition of **\t** to represent a horizontal tab character. The output of print statements is what is used to mark your project. Be sure to get them right! Use "%c" for a character; do **not** print the single quotes around or the percent sign (%) before the character. Use "%s" for a string; do not print the double-quotes around the string. Use "%f" for floating values (floating printing will be ugly, but we don't have the time to fix it).

Both plus and minus (**+** and **-**) are allowed as unary operators in Tan-1. The plus unary operator does nothing. Both unary operators apply to both integers and floats.

A binary 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 records 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 and the casting operator have the highest precedence,
unary **–** and **+** have the next highest precedence,
/ and ***** have the next highest,
- 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.

Block statements introduce a new scope in which variables can be declared. Variables in inner scopes (e.g. nested block statements) will hide variables in outer scopes with the same name. Use "enterSubscope" and "leaveSubscope" in visitEnter/visitLeave for BlockStatementNodes. This is similar to what is done for ProgramNodes.

Other language details may be discussed in lecture.