Scriptable Interpreted Computer Testing Calculator (100 points)

You are to write an interpreter for the Scriptable Interpreted Computer Testing Calculator (SICTC) programming language, in 3 parts.

Include a main file, where you can call each individual part of the project (I.E. loop through and print each token generated by the scanner).

1 Scanner (30 points)

Due 4 April 2018

Phase 1: Write a program that takes an input data file (the format is below), creates a list of the **tokens** from that file, and then prints out that list of tokens (one per line).

Thus, your main program should have the user specify a filename (either using Python's `input` function, or by passing a string to a function), invoke the **scanner** to create the list of tokens, which should be returned by the scanner.

If the input includes characters not consistent with a token, stop and print an error message (including the line number). Note that the order that the tokens appear should not matter for this phase, as long as the tokens themselves are correct.

See the language specification for the list of token types and how comments work. Note that how tokens are listed in the language specification is not necessarily how you should separate your tokens into different kinds for purposes of parsing.

2 Parser (30 points)

Due 18 April 2018

Phase 2: Write a **recursive-descent parser** to check that the input is syntactically correct.

Using the list of tokens generated by your program from Phase 1, create a function to parse each **production** from the language grammar. If a token fails to match, or you run out of tokens before finishing, print an error message and stop. This error message should include the line number and indicate what kind of token was expected.

See the language specification for the language grammar rules.

3 Semantic Analyzer (40 points)

Due 9 May 2018

Be sure to make a backup copy of your parser before starting this phase! Your code for this section will interleave with the parser code.

Phase 3: There are two semantic checks you will need to make:

1. Make sure no variables are declared more than once.
2. Make sure every variable used in the body of the program has been declared.

To do this, you will need to make a symbol table. I recommend using a Python `dict`. You will also need to keep track of temporary variables needed, labels needed, and constants. An integer counter is sufficient for those.

You will then create an **attribute grammar**, which specifies how syntax is associated with semantics and add it to the code you wrote for your parser. This will allow you to evaluate programs written in the SICTC programming language.

Language Specification

Token Types

Note: unless you do the extra credit to add decimal numbers, implement division as floor division.

The scanner must recognize the following tokens:

Integer constant	one or more decimal digits
Arithmetic operators	+ − * / %
Relational operators	= != < <= > >=
Assignment operator	:=
Punctuation	: ; ( ) . ,
ID (variable/program names)	a string of ASCII letters, case insensitive
Keywords (12)	`begin do end false halt if print program then true var while`

Comments

Comments begin and end with pound signs (`#`), may be multiple lines, and may share a line with tokens. There can be more than one comment per line. It is an error if a comment is not closed before the end of the program.

Details

- The language is free-format—whitespace (spaces, tabs, newlines, etc.) is ignored. Comments act as whitespace. Note that this means that you can't simply split a line based on whitespace and expect to get the correct answer every time. You will have to check character by character.
- Variable and program names are strings of ASCII letters, case insensitive.

- The print command does not support any formatting, nor does it support strings or characters of any kind. It simply prints the value stored in a variable, followed by a newline character.
- All variables store only integers, all expressions are integer-valued, and all expressions must be fully parenthesized.
- There are no procedures, functions, structs, records, classes, pointers, arrays, lists, objects, etc. in this language.
- Relational operators return 0 for false and 1 for true. `if` and `while` execute their bodies if their expression is non-zero.
- These details *will* change if you do the extra credit.

Grammar

Here, ϵ represents the empty string, | means "or", bold text represents **terminals** (of the specified kind, not necessarily that literal text) and italic text in a diamond represents <*nonterminals*>.

<*SICTC-program*>	::=	<*header*> <*body*>
<*header*>	::=	**program: id**; <*declarations*>
<*declarations*>	::=	**var:** <*id-list*> **;**
<*id-list*>	::=	**id** <*id-list2*>
<*id-list2*>	::=	**,** <*id-list*>
		| ϵ
<*body*>	::=	**begin:** <*statement-list*> **halt.**
<*statement-list*>	::=	<*statement*> <*statement-list*>
		| ϵ
<*statement*>	::=	**id :=** <*expression*> **;**
		| **print: id ;**
		| **if:** <*expression*> **, then:** <*statement-list*> **end.**
		| **while:** <*expression*> **, do:** <*statement-list*> **end.**
<*expression*>	::=	**id**
		| **integer**
		| **boolean**
		| **(** <*expression*> **operator** <*expression*> **)**

Sample Program

```
 1  program: Sample;
 2  # This is a sample program in SICTC programming language.
 3  Comments appear within pound signs.  #
 4  var: a, x, s, t; # all variables are integers #
 5  begin:
 6    a:=100;
 7    if: (a>0), then:
 8      x:=1; s:=0;
 9      while: (x<=a), do:
10        s:=(s+x); x:=(x+1);
11      end. # Indenting isn't necessary, #
12    end. # but is still a good habit. #
13    t:=((a*(a+1))/2);
14    print: s;
```

```
15    print: t;
16  halt.
```

Hand In and Notes

Turn in a listing of your program and a sample run for each phase, when it is due. Make sure your code works or fails on all samples provided, appropriately. I will provide examples of both code that should work and code that should fail. Be sure to include everyone's name in the comments at the top of your program.

- Remember to include comments at the top of your program and a few lines for each function/method (for pre- and post-conditions).
- Begin by writing programs that only do the very easiest part of each assignment, and then slowly add functionality to a working program. Don't try to write the whole thing in one go.
- **Start right away!** I wrote this program from scratch in about 15 hours and it's about 700 lines of code. You will want to space this work out!
- Don't start on any of the extra credit until you have fully completed the semantic analysis. It will be easier to go back and add code for the extra credit after the fact than it will be to try to write each part with the extra credit included. The extra credit will be entered in the gradebook as a separate assignment.
- You are allowed to modify the code of a previously submitted phase, if you find that you need to change something to make writing the next phase easier. If this is the case, just resubmit your modified code along with the code of the next phase.

Extra Credit

- (3 points) As specified, negative integer constants are not allowed. Extend your scanner and modify the grammar to allow negative integer constants and the negation unary operator -.
- (5 points) As specified, decimals are not allowed. Extend your scanner and modify the grammar to handle decimal numbers. Decimals in both standard (i.e. 3.14) and scientific (i.e. $6.28e3$ or $6.28e-4$) should be accepted. If you do the first extra credit, your code must handle negative decimals as well.
- (5 points) As specified, string literals are not allowed. Extend your scanner and modify the grammar to handle strings, with the concatenation operator +. Note that concatenating a string with an integer (or decimal) should be an error.
- (2 points) `if` statements do not permit an `else` clause. Add this. The `else` clause should be optional.
- (5 points) Add the ternary operator `?:` to your program, as another terminal under "expression". An expression `(a)? b: c` evaluates to `b` if the value of `a` is true, and otherwise to `c`. (Treat the value 0 as false and nonzero values as true, as elsewhere).
- (5 points) Add either a counted loop (for loop) or a bottom-tested loop (do-while loop).
- (1 point each, maximum 4) Add the boolean operators and, not, or, and xor. You could add keywords or use symbols like &, !, | , and ^.
- (10 points) Add a Read-Evaluate-Print Loop (REPL) that can be called instead of the default interpreter. The REPL should include a new statement and associated keyword `run` which, when given an argument of a path to a valid SICTC program, runs that program using the default interpreter. `run` is

not a valid keyword in any program ran by the default interpreter, only the REPL. The REPL should also maintain variable values in between statements entered, including those used in a `run`. (I.E. if you set $a := 5$, calling $print : a$; will always print 5 until a is set to a new value.)