In this phase you will undertake the modifications to the Semantic phase of the PT Pascal compiler to turn it into a semantic analyzer for Qust. These changes will be more extensive than those of phases 1 or 2, and the Semantic phase is much harder to understand, so start early on this phase! The biggest problem in this phase will be simply understanding what is going on in it. Read ahead in the text if necessary. Ask questions of your TA. Ask questions on the course forum. Get at it!

As usual, we will try to make the output of our semantic phase as much like the output of the original PT semantic phase as possible, in order to minimize the changes we will have to make to the PT code generator in Phase 4. For example, we will extend and reuse the existing PT while statement T-codes to implement Qust loops as well.

The following suggestions are provided to guide you in this phase, but as usual you are free to implement the new features in any way you like, provided you can show that your solution will work.

Suggestions for Implementing Phase 3

Definitions

Begin by modifying the semantic phase input token list to correspond to the set of tokens emitted by your Phase 2 Qust Parser, and then modify the semantic phase output T-code list to include the new Qust T-codes used in the extensions listed below. Comment out or remove any parts of semantic.ssl and semantic.pt that are no longer valid because of the changes.

Extensions to the T-Code Machine Model

The Qust abstract machine has the following new T-code instructions to handle Qust strings. These instructions use the standard PT abstract machine expression stack (Stack) and internal registers (L, R). The notation used below is the same as in appendix B1 of the PT report. L <-- Stack means pop the expression stack into internal register L. Stack <-- (L + R) means push the sum of the values in internal registers L and R onto the expression stack, and so on.

StringStack is a new internal evaluation stack in the string-handling "chip" of the Qust abstract machine. The string-handling chip is a new unit, much like a floating point unit, that understands how to do the string operations in "hardware". In addition to the StringStack, the string-handling chip has two internal string "registers", SR and SL.

```
tLiteralString (string)          StringStack <-- (string)
tFetchString                     L <-- Stack;  SR <-- Memory[L];
                                 StringStack <-- SR
tAssignString                    SR <-- StringStack;  L <-- Stack;
                                 Memory[L] <-- SR
```

1 Note that tSubstring implicitly implements scaling by the width of a string, in the same way tSubstringInteger implements scaling by the size of an integer.

2 Note that equality and inequality (== and !=) are the only comparison operators defined on Qust strings.

Input/output of strings is done using new versions of the old trap codes trReadString and trWriteString. tTrap trReadString reads the characters up to the next line boundary and pushes them as a string onto the StringStack. tTrap trWriteString writes the string on top of the StringStack to the output.

The Qust abstract machine also has the following additional T-code instructions to handle Qust loops and case statement default alternatives:

```
tWhilePreBreak null operation

tWhileBreakIf null operation

tCaseOtherwise null operation
```

3 Note that a T-code implementing a null operation does not mean that it is redundant or optional; it helps the code generator understand the meaning of the T-code stream.

Iterative Development

Phase 3 is much more challenging than the first two phases, so I’ve provided detailed steps to make sure you are addressing all the issues. Ask your TA if you find that you do not understand anything in this list of steps. This is not necessarily a complete list, depending on the details of your solution. If you notice anything missing or incorrect, please post a question on the CISC 458 forum.

Because many language features are independent of one another, in the next two phases of the project we can take advantage of test-driven iterative development, working on and testing each new feature one at a time. Remember that features not
Step 1: Programs

Because we tried to keep the semantic token stream output by the Parser phase as similar as possible to the PT Pascal one, the handling of programs themselves is pretty well unchanged in Qust. Begin by updating the definitions in semantic.ssl as described above, comment out or remove any old PT features no longer in Qust, then build the semantic phase and test with a null program as input, before making any other changes. A Qust null program looks like this:

```c
mod main (output) {
}
```

Step 2: Blocks (Statements and Declarations)

As you know from the Parser phase, Qust allows both declarations and statements to be intermixed. Modify the PT Pascal Block rule in semantic.ssl to merge in the alternatives of the old PT Pascal Statement rule to remove the distinction. Remember that our Qust Parser outputs sequences of statements and declarations with sBegin at the beginning of each block and sEnd at the end. Remove handling of both begin and repeat statements, which are not in Qust, if you have not already done so.

Because we wrapped every sequence of declarations and statements in sBegin and sEnd in the Parser output, every such sequence is now a single Block as far as the semantic phase can tell. Since in Qust each block of declarations and statements also defines its own scope, replace the old Statement rule with one that simply pushes a new scope, calls the Block rule, then pops the scope.

Make these changes, then test with an input program that uses some integer constants, variables, assignments, and while statements to make sure things are still working before making any more changes.

Step 3: Types

Type specifications in Qust are similar to PT, but have a number of changes, in particular to arrays. Change the array type definition handling in semantic.ssl to reverse the order of accepting component and index types, as in Qust. Qust does not have Pascal subrange types, so remove the handling of subrange types in SimpleType, and add an alternative for sRange alone which makes a constant 1 for the lower bound before calling SubrangeUpperBound to handle the Qust array size.

Make these changes, rebuild the semantic phase, and then test using a simple Qust program that uses integer arrays to make sure that things are still working before moving on to the next set of changes.

Step 4: Initial Values

Unlike PT, Qust allows for initial values in variable declarations handling of optional types and initial values in Variable Declarations. If there is both a type and an initial value, then just call Assignment to assign it after processing the type of the variable. If there is no type (i.e., only an initial value), make the variable the default integer type and then call Assignment to assign the initial value.

Make these changes, then test using a Qust program with both typed and untyped variable declarations with and without initial values. (But don't try string types yet.)

Step 5: Modules

Add a new ModuleDefinition rule in semantic.ssl to handle the declaration of modules as specified in the Qust language specification. Specify a new symbol kind syModule in the type SymbolKind for modules, and use it for the module's symbol declared in the enclosing module or main program scope. (Although the module's symbol cannot be used for any particular purpose in the program, you should enter it in the symbol table so as not to allow redeclaration of the module name as something else.)

Module definitions are much like procedure definitions, except that there is no need for a tSkipProc around the body or tProcedureEnd at the end, since the meaning of a module declaration is simply the execution of the declarations and statements in its body, and a module has no header or formal parameters to process. Module scopes are like procedure scopes in PT, that is, each module has its own local scope. However, the symbol table entries for public procedures (declared as pub fn in Qust) must be transferred to the enclosing scope when the end of the module is encountered so that they can be called from outside the module. Qust modules export their public procedures “unqualified” - that is, if module M has public procedure P, then it is called from outside the module in Qust simply as P, not as M.P.

The easiest way to implement this is to mark each public procedure in the symbol table with a special symbol kind that says it is a public procedure. Specify a new symbol kind syPublicProcedure in the type SymbolKind, and add handling of sPublic in procedure definitions to set the symbol kind of public procedures to the new symbol kind syPublicProcedure. Then add syPublicProcedure to also be accepted everywhere in semantic.ssl that syProcedure is accepted in semantic.ssl, and change all the assertions in semantic.pt that insist on the top of the SymbolStack being syProcedure to allow for syPublicProcedure as well.

To transfer public procedures to the enclosing scope, instead of popping the module's scope from the symbol table, we will strip out everything that is not public and then merge the module's scope into the enclosing one. To do this, we will add two new SymbolTable mechanism operations, oSymbolTblStripScope and oSymbolTblMergeScope, and use both of them together instead of oSymbolTblPopScope when processing the end of a module definition.

oSymbolTblStripScope looks through the top scope in the symbol table, setting the symbol table reference for each symbol's identifier index to the one at the next lower lexical level (i.e, set the identSymbolTableRef for the identifier to the symbolTable IdentLink for the symbol; see the comments in the implementation of the semantic
The following template gives the T-code implementation of Qust conditional exit.

```
tWhilePreBreak simply reuse the existing PT T-codes for before the conditional exit. So to save ourselves work in the code generator, we will statement, since we wrapped them in while, handle the Qust Remove handling of the PT case statement to handle the Qust match statements with the optional default alternative (I _ => ) instead. Qust match statements are exactly like PT case statements, and we reused the existing PT case statement semantic tokens to represent them in the parser output, so you can just reuse the existing CaseStmt S/SL rule, and simply extend it to look for an sCaseOtherwise and statement before the sCaseEnd. Handle the statement of the otherwise clause as if it were another alternative, but with no label.

The Qust match statement is very similar to the existing PT case statement, so we will reuse all the PT case statement T-codes for it to save work in the code generator. The following template gives the T-code implementation of Qust match statement default alternatives, which are required in Qust.

```

The S/SL rule to generate this code is exactly like the rule for the while loop in the PT semantic phase, except that a Statement is allowed before the break if part, and the exit condition must be inverted (with tNot).

Make these changes, rebuild the phase, then test a program with loop statements in it before proceeding.

Step 7: The Match Statement and Default Alternative

Change the handling of the PT case statement to handle the Qust match statements with the optional default alternative (I _ => ) instead. Qust match statements are exactly like PT case statements, and we reused the existing PT case statement semantic tokens to represent them in the parser output, so you can just reuse the existing CaseStmt S/SL rule, and simply extend it to look for an sCaseOtherwise and statement before the sCaseEnd. Handle the statement of the otherwise clause as if it were another alternative, but with no label.

The Qust match statement is very similar to the existing PT case statement, so we will reuse all the PT case statement T-codes for it to save work in the code generator. The following template gives the T-code implementation of Qust match statement default alternatives, which are required in Qust.

```

The CaseStmt rule must be modified to require the default alternative (indicated by sCaseOtherwise) and generate the tCaseOtherwise through tCaseMerge part shown above after tCaseEnd and branch table. The default alternative is much like any other case alternative, except that it is emitted after the tCaseEnd.

Make these changes, rebuild the phase, then test a program with match statements in it before proceeding.
Step 8: The Else If Clause
If you have completely handled else if in your Parser, then congratulations! This is the payoff and there is nothing further to do. However, if you have chosen to defer handling of else if to the semantic phase, then you must now change the handling of if statements in the semantic phase to handle else if. This is actually not very difficult - modify the S/SL rules for if statements to handle else if clauses exactly as if the equivalent else (if ...) had been received from the parser. Be careful that you get exactly the equivalent T-code - this is easy to test by making two test programs, one that uses else if and another that uses the equivalent else (if ...), and checking that the output T-code is the same.

Step 9: Mutable and Immutable Variables
Add handling of mutable variables (sMutable) in variable declarations. Change the symbol kind of mutable variables to the new symbol kind syMutableVariable, and modify semantic.ssl to add syMutableVariable as accepted everywhere that syVariable is accepted. Since in Qust mutable variables cannot be assigned, add a check in AssignmentStatement to make sure that the target of an assignment is a mutable variable, and if not give an error message.

Modify the ActualParameters rule to check for sMutable, which is an error if it appears on value parameters (since they can't be assigned to) and is required on all var parameters (so emit an error if it is missing). In VarActual, check that the actual parameter is a mutable variable (i.e., of kind syMutableVariable), and if not give an error message.

To test these changes, make a test program with both mutable and immutable (regular, non-mut) variables in it, and try assigning to each. Make a Qust function with both value (non-mut) and var (mut) parameters, and try passing mutable and immutable variables to it to see that you are catching the errors.

Step 10: The String Type
Replace the handling of the char data type and operations with the str type and the corresponding operations and traps of Qust strings. If you haven't already done so, begin by changing all the T-codes in the Output section for Char operations to be String operations (e.g., tFetchChar becomes tFetchString).

In semantic.ssl, change all uses of the Char T-codes to use the String T-codes instead. In general, everywhere it presently says "Char" in semantic.ssl it should now say "String". Storage allocation for strings is in units of stringSize, which is 1024. Add a definition for stringSize to the type Integer, and in type StdType, change stdChar to stdString. In type TypeKind, change the type kind for char (tpChar) to be for string (tpString), and change all uses of the Char type in the whole S/SL source to use the String type instead.

In semantic.pt, change the predefined type for Char to be a predefined type for String in the predefined type table entries and their initialization. Change all references to the Char type ref in the program to refer to String instead. Change the predefined type pidChar to pidString, and modify the predefined type pidText to reference String instead of Char in procedure Initialize. Modify the implementation of the cAllocateVariable semantic operation to handle allocation of Strings (size 1024, stringSize), and don't forget about string arrays.

Replace the PT character array handling of string literals (sStringLiteral) in rule Expression entirely. Strings are first class values in Qust, so we no longer need the tSkipString and tStringDescriptor stuff in the T-code for string literals. Instead, just emit a simple tLiteralString, much like tLiteralInteger for integers. The only difference is that tLiteralString requires the string length before the string itself in the T-code output.

Example:

```
"hi" in PT used to generate: tSkipString L S: 2 "hi"
```

Because we don't have any place in the symbol table to store string literals, string const declarations in rule ConstDefinitions should be treated as if they were variables. You should process the declaration:

```
const littleString = "Hello mom";
```

exactly as you would process the sequence:

```
let littleString : str;
littleString = "Hello mom";
```

which should generate the T-code:

```
tAssignBegin
tLiteralAddress s
tLiteralString 3 "foo"
tAssignString
```

That is, you should set littleString's symbol kind to an syVariable linked to standard type stdString, allocate storage for it, and generate the T-code sequence for a string assignment of the literal string value to it, exactly as assignment statements are handled in the AssignmentStmt rule (although obviously you don't have to check the types). Finally, you should set its symbol kind to syConstant, to remember that it cannot be assigned to in Qust.

Step 11: String Operations and Traps
Add handling of the string unary operation sLength to UnaryOperator, and the string operations concatenate (sAdd with string operands), string equality and inequality (sEq and sNE with string operands), repetition (sMultiply with string first operand), and substring (sSubstring with a string first operand and integer second and third ones) to BinaryOperator. Handle string operations using the obvious translation from postfix semantic tokens to sequences of the new StringStack T-code operations as defined on the first page.
Remember that strings are first class values in Qust, so for example, string concatenation is just like integer addition in terms of what to do, except the T-codes are different. Remember that \texttt{substring} actually takes three operands.

\textbf{Examples:}

\begin{verbatim}
"hello" / 4 : 5 sStringLiteral "hello" tLiteralString 5 "hello"
sInteger 4 tLiteralInteger 4 sInteger 5 tLiteralInteger 5 sSubstring tSubstring
"hel" + "lo" sStringLiteral "hel" tLiteralString 3 "hel"
sStringLiteral "lo" tLiteralString 2 "lo" sAdd tConcatenate
?
"hello" sStringLiteral "hello" tLiteralString 5 "hello"
S = "hello"; sAssignmentStmt sLength tLength
S = "hello"; sAssignmentStmt sIdentifier S tLiteralAddress S sStringLiteral "hello" tLiteralString 5 "hello" sExpnEnd tAssignString
\end{verbatim}

Qust strings can only be compared for equality (==) and inequality (!=) using the \texttt{tStringEqual} T-code operation (there is no \texttt{tStringNotEqual} operation, but I'm sure you can figure out how to handle inequality using \texttt{tStringEqual}). Qust strings cannot be compared for ordering (i.e., they can't be compared for $>$, $<$, $\geq$ or $\leq$).

Remove handling of \texttt{tpChar} entirely from \texttt{CompareRelationalOperandTypes} so an error will be flagged if the operands are strings.

Change the operands of the \texttt{rtChr} and \texttt{rtOrd} character traps to string (i.e., using \texttt{tpString} in place of \texttt{tpChar}). In the input/output procedures, change \texttt{AssignProcedure} to expect \texttt{tpString} as a parameter instead of the old \texttt{PT} character array, and remove handling of old \texttt{PT} char arrays entirely from \texttt{WriteText}. Replace the old \texttt{SymbolStkPushDefaultCharConstant} rule with \texttt{SymbolStkPushDefaultStringConstant}.

Change the names of the traps \texttt{trReadChar} and \texttt{trWriteChar} in type \texttt{TrapKinds} to be for \texttt{trReadString} and \texttt{trWriteString}, and change their trap numbers to 108 for \texttt{trReadString} and 109 for \texttt{trWriteString} (which are the trap numbers I have assigned to them in the Qust runtime library). Remove the redundant extra \texttt{trWriteString}. Change all uses of the \texttt{Char} input/output traps in \texttt{semantic.ssl} to use the new \texttt{String} traps instead.

\textbf{Note:} Do \textbf{not} try to perform any compile-time optimization of expressions involving strings. (There are hundreds of such possible optimizations and you could spend the rest of the term implementing them!)