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 Drift. 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!

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

Modify the semantic phase input token list to correspond to the set of tokens emitted by your new Drift Parser and the semantic phase output T-code list to include the new Drift T-codes used in the extensions listed below.

Extensions to the T-Code Machine Model

The Drift abstract machine has the following new T-code instructions to handle Drift strings. These instructions use the standard PT abstract machine expression stack \((\text{Stack})\) and internal registers \((L, R)\). The notation used below is the same as in appendix B1 of the PT report.

\[
\text{L} \leftarrow \text{Stack} \quad \text{(pop the expression stack into internal register \(L\))}
\]

\[
\text{Stack} \leftarrow (L + R) \quad \text{(push the sum of the values in internal registers \(L\) and \(R\) onto the expression stack)}
\]

\[
\text{StringStack} \leftarrow (\text{string}) \quad \text{(push a string onto the StringStack)}
\]

\[
\text{SR} \leftarrow \text{StringStack} \quad \text{SL} \leftarrow \text{StringStack} \quad \text{Stack} \leftarrow (\text{SR} :: \text{L} .. \text{R}) \quad \text{(create a substring)}
\]

\[
\text{SR} \leftarrow \text{StringStack} \quad \text{Stack} \leftarrow (# \text{SR}) \quad \text{(get length of string)}
\]

\[
R \leftarrow \text{Stack} \quad \text{Stack} \leftarrow \text{chr}(R) \quad \text{StringStack} \leftarrow \text{SR} \quad \text{(get character)}
\]

\[
\text{SR} \leftarrow \text{StringStack} \quad R \leftarrow \text{ord}(\text{SR}[1]) \quad \text{Stack} \leftarrow R \quad \text{(get ASCII value)}
\]

\[
\text{SR} \leftarrow \text{StringStack} \quad \text{SL} \leftarrow \text{StringStack} \quad \text{Stack} \leftarrow (\text{SR} == \text{SL}) \quad \text{(equality test)}
\]

\[
\text{tLiteralString} \quad \text{StringStack} \leftarrow (\text{string})
\]

\[
\text{tFetchString} \quad L \leftarrow \text{Stack}; \quad SR \leftarrow \text{Memory}[L]; \quad \text{StringStack} \leftarrow SR
\]

\[
\text{tAssignString} \quad SR \leftarrow \text{StringStack}; \quad L \leftarrow \text{Stack}; \quad \text{Memory}[L] \leftarrow SR
\]

\[
\text{tStoreParmString} \quad SL \leftarrow \text{StringStack}; \quad R \leftarrow \text{Stack}; \quad \text{Memory}[R] \leftarrow SL
\]

\[
\text{tSubscriptString} \quad R \leftarrow \text{Stack}; \quad L \leftarrow \text{Stack}; \quad \text{Stack} \leftarrow (L + R)
\]

\[
\text{tConcatenate} \quad SR \leftarrow \text{StringStack}; \quad SL \leftarrow \text{StringStack}; \quad \text{StringStack} \leftarrow (\text{SR} + \text{SL})
\]

\[
\text{tSubstring} \quad SR \leftarrow \text{StringStack}; \quad R \leftarrow \text{Stack}; \quad L \leftarrow \text{Stack}; \quad \text{StringStack} \leftarrow (\text{SR} :: \text{L} .. \text{R})
\]

\[
\text{tLength} \quad SR \leftarrow \text{StringStack}; \quad \text{Stack} \leftarrow (# \text{SR})
\]

\[
\text{tChr} \quad R \leftarrow \text{Stack}; \quad SR \leftarrow \text{chr}(R); \quad \text{StringStack} \leftarrow SR
\]

\[
\text{tOrd} \quad SR \leftarrow \text{StringStack}; \quad R \leftarrow \text{ord}(\text{SR}[1]); \quad \text{Stack} \leftarrow R
\]

\[
\text{tStringLength} \quad SR \leftarrow \text{StringStack}; \quad \text{SL} \leftarrow \text{StringStack}; \quad \text{Stack} \leftarrow (\text{SR} == \text{SL})
\]

1. Note that \(\text{tSubscriptString}\) implicitly implements scaling by the width of a string, in the same way \(\text{tSubscriptInteger}\) implements scaling by the size of an integer.

2. Note that equality and inequality \((==\text{ and }!=)\) are the only comparison operators defined on Drift strings.

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

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

\[
\text{tLoopBegin} \quad \text{null operation}
\]

\[
\text{tLoopBreakIf} \quad \text{null operation}
\]

\[
\text{tLoopTest} \quad (\text{code loc}) \quad R \leftarrow \text{Stack}; \quad \text{if } R \text{ then } \text{IP} := (\text{code loc})
\]

\[
\text{tLoopEnd} \quad (\text{code loc}) \quad \text{IP} := (\text{code loc})
\]

\[
\text{tCaseDefault} \quad \text{null operation}
\]

Modules

Add handling of the declaration of anonymous modules as specified in the Drift language specification. Specify a new symbol table entry kind \(\text{syModule}\) for modules.

The T-code implementation of modules does not involve any new T-codes. Just treat the module as if it were part of the main program, skipping around the declarations and executing the module statements as if they were the main program statements. The only tricky part is to continue execution after the statements (i.e., don’t return from them, just fall through to the rest of the program).

The name of a module should be a symbol declared in the enclosing module or main program scope. Although the module’s symbol cannot be used for any particular purpose, you should enter it in the symbol table so as not to allow redeclaration of the module name as something else.

The implementation of the module scope should be done like a procedure scope in PT, that is, each module should have its own local scope. However, the symbol table entries for public procedures (those declared with \(^\star\)) must be transferred to the enclosing module or main program scope when the end of the module is encountered so that they can be called from outside the module. Drift modules export their public procedures “unqualified” - that is, if module M has public procedure P, then it is called from outside the module simply as P, not as M.P.
The easiest way to implement this is to mark each public procedure in the symbol table with an attribute which says whether it is a public procedure or not (if you're careful this can be done by adding a new SymbolKind for public procedures, e.g. syPublicProcedure, without adding a new array for the attribute). Then add two new SymbolTable mechanism operations: oSymbolTblStripScope and oSymbolTblMergeScope. Call both of these instead of oSymbolTblPopScope when processing the end of a module.

oSymbolTblStripScope should go 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 nameSymbolTableRef for the identifier to the symbolTableNameLink for the symbol; see the comments in the implementation of the semantic operation oSymbolTblPopScope in semantic.pt for more information). This effectively makes the symbols inside the module invisible. Of course, we want the public procedures to remain visible, so it should not make the change for any symbol that is marked as a public procedure.

oSymbolTblMergeScope should simply merge the top two scopes in the symbol table by decreamenting the lexical level but not changing the symbol table top. This has the effect of putting all of the module's symbols into the enclosing module or main program's scope. However, if oSymbolTblStripScope has been called first, then only the public procedures will be visible, which is exactly the effect we want.

If you are ambitious, these two operations can be optimized to remove the inaccessible entries from the symbol table altogether and thus save storage. They could also be combined into a single operation if you wish.

The string Type

Remove handling of the char data type and the operations and traps for characters. Add handling of the string type. Storage allocation for strings should be in units of stringSize, which is 256 (for now). Handle string operations using the obvious translation from postfix to sequences of new T-code operations as defined above.

String const declarations should be treated as if they were variables which are subsequently assigned. That is, you should compile the declaration:

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

exactly as you would the sequence:

```
var littleString : string
littleString = "Hello mom"
```

Do 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!)

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

The elsif Clause

If you have completely handled elsif in your parser, then this is the payoff and there is nothing further to do. However, if you have chosen to defer handling of elsif to the semantic phase, then you must now change the handling of if statements in the semantic phase to handle elsif. This is actually not very difficult - modify the S/SL rules for if statements to handle elsif 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 elsif and another that uses the equivalent else ... if, and checking that the output T-code is the same.

The loop Statement

Remove handling of the PT repeat statements. Add handling of the loop statement as specified in the Drift language specification. The following template gives the T-code implementation of Drift loops.

```
loop
    . . .
    break if expression
        tLoopBreakIf
            (T-code for expression)
        tLoopTest
            exitlabel
    . . .
end
    . . .
    exitlabel: . . .
```

The rule to generate this code is just like the rule for the while loop in the PT semantic phase, except that a statement sequence is allowed before the exit test.

The switch Statement and default Clause

Change PT case statement handling to the Drift switch statement. Add handling of the Drift default clause. The following template gives the T-code implementation of Drift switch default clauses. Remember that they are optional.

```
switch expression
    . . .
    case value1 :
        label1:
            . . .
            tCaseSelect
                tablelabel
            . . .
        tCaseMerge
            endlabel
    . . .
    case value2 :
        label2:
            . . .
            tCaseMerge
                endlabel
        . . .
            (more alternatives)
            . . .
        tCaseMerge
            endlabel
    . . .
    default :
        tCaseElse
            . . .
        tCaseMerge
            endlabel
end
    . . .
    exitlabel: . . .
```

(file branch table)
The *CaseStmt* rule must be modified to handle both the case where there is no *default* clause (in which case it should do the same thing it does now) and the case where there is a *default* clause (indicated by *sDefault*) in which case it should generate the *tCaseElse* through *tCaseMerge* part shown above after the branch table.

**Statement Sequences**

Because we changed all statement sequences into *sBegin .. sEnd* in the semantic token stream output by the parser, statement sequences are now single statements as far as the semantic phase can tell. So all we need is to handle a single statement wherever Drift allows a statement sequence, making it exactly like PT - so nothing special to do for these.

**Functions (CISC 858 only! )**

Add handling of functions as specified in the 858 extensions to the Drift language specification. Specify new symbol table entry kinds for functions and public functions. Functions are only allowed to return the types integer, string and boolean.

The Drift implementation of functions is exactly like procedures with two minor changes: Following the parameters, you must accept the result type, change the function’s type stack entry to reflect that type and update the function’s symbol table entry with the new type. At the end of the function, accept the result expression using the Expression rule to generate code for it. Check that the type of the function and the type of the expression match and issue an error message if not. Use a new T-code, *tFunctionResult*, to mark the beginning of the function’s value expression.

Here is an example of the T-code sequence to implement functions:

```plaintext
func square (x: integer) : integer tSkipProc sqend
tLiteralAddress x tStoreParmInteger tParmEnd

return x * x tFunctionResult
tLiteralAddress x tFetchInteger
tLiteralAddress x tFetchInteger tMultiply

tProcedureEnd
sqend:
```

And here is an example of the T-code sequence to call a function:

```plaintext
z = square (y) + 1 tAssignmentStmt
tLiteralAddress z tCallBegin
tLiteralAddress y tFetchInteger tParmEnd
tCallEnd square tLiteralInteger 1 tAdd tAssignInteger
```
Phase 3 Changes Checklist

Phase 3 is much more challenging than the first two phases, so here is a checklist to help you to make sure you are addressing all the issues. Ask your TA if you find that you do not understand any of the items on the list. This is not necessarily a complete list, depending on the details of your solution.

Changes to semantic.ssl

1. Change the Input semantic token definitions in semantic.ssl to be the same as the Output semantic tokens in your parser.ssl.

2. Change all the T-codes in the Output section for Char operations to be String operations (e.g., tFetchChar becomes tFetchString). Change all uses of the Char T-codes in the S/SL source to use the String T-codes instead. In general, everywhere it presently says “Char” in the semantic.ssl S/SL source it should now say “String”. Remove the redundant tLiteralChar T-code. Add the new T-codes for tConcatenate, tSubstring, tLength and tStringEqual. Replace the old T-codes for the repeat operations with the new loop operations tLoopBegin, tLoopExit, tLoopTest and tLoopEnd and add the new tCaseElse T-code. For 858, add the new tFunctionResult T-code as well.

3. Add a definition for stringSize to the type Integer. The value is 256.

4. In type StdType, change stdChar to stdString.

5. Add the new oSymbolTblStripScope and oSymbolTblMergeScope operations to the SymbolTable mechanism.

6. Add a kind for syModule to the type SymbolKind. If you are using a special kind for public procedures, add a kind for them also (e.g. syPublicProcedure), otherwise add a public attribute in another way. 858 groups should add syPublicFunction as well.

7. In type TypeKind, change the type kind for char (tpChar) to be for string (tpString). Change all uses of the char type in the whole S/SL source to use the string type instead. Remember that strings are first class types in Drift, so they act like integers, not like packed arrays as in PT.

8. In type TrapKind, change the names of the traps for read and write char to be for string, and change their trap numbers to 107 for trReadString and 108 for trWriteString (which are the trap numbers I have assigned to them in the Drift runtime library). Remove the redundant extra trWriteString. Change all uses of the char traps in the S/SL to use the string traps instead.

9. Change the Program rule to handle Drift program headers (i.e. using sExtern) instead of PT program headers. Or alternatively, if your parser hides this change by emitting exactly the old sProgram token, then you won’t need to make this change.

10. Add a new rule ModuleDefinition to handle modules. A ModuleDefinition is much like Program except it has no extern parameters and no halt. At the end of the module it uses the new strip scope and merge scope symbol table operations to promote all public symbols to the enclosing scope.

11. Modify handling of constant definitions to allow only one per definition.

12. Modify handling of type definitions to allow only one per definition.

13. Modify handling of variable declarations to allow multiple identifiers declared using one type, but only one declaration per definition. You will have to push all the declared identifiers on the Symbol Stack and keep count of how many there are using the Count Stack, and then accept the type and use it to set the type and enter in the SymbolTable all the identifiers you pushed, one at a time. Remember to keep the stacks straight and clean up after you’re done. Watch out for the fact that the code to do this for one variable in PT swaps the type stack but forgets to swap it back after entering the variable type - this will have to be fixed when you do it for more than one variable.

(Note: If you’re finding this change too challenging, try just doing the single identifier case with no counting first.)

14. Change handling of procedure definitions to recognize public procedures and store them with the special public attribute or special symbol kind syPublicProcedure. This is tricky and will require you to copy some code in the procedure definition rule.

15. Remove the handling of repeat statements and add handling of the Drift general loop statement. Handling loop statements is just like while statements except that the T-codes are different and there is a statement allowed before the condition test.

16. Change case statement handling to handle the Drift switch statement. Add handling of the optional default clause of the Drift switch statement. The default clause is much like another case alternative, except emitted after the tCaseEnd.

17. Add handling of ternary (three-operand) operators (e.g. substring) to the Expression rule. Add a new TernaryOperator rule to handle substring operations. Look at the BinaryOperator rule as a model.

18. Change UnaryOperator rule to handle the string length operation as well. Be careful to get the type checking right.

19. Add handling of concatenation to the sAdd part of the BinaryOperator rule. Remember that strings are first class values in Drift, so string concatenation is much like integer addition in terms of what to do.

20. Strings are first class values in Drift, so we no longer need the tSkipString and tStringDescriptor stuff in the T-code for string literals. The T-code for a string literal in any context should simply be tLiteralString.

Example:

"hi" in PT used to generate:  
  tSkipString L  
S: 2  
L: tLiteralString S  
"hi" should now generate:  
  tLiteralString 2  
"hi"
21. Change handling of string constants to act like vars instead. For example,

```plaintext
const s = "foo"
```

should be handled as if it were:

```plaintext
var s: string
s = "foo"
```

that is, it should generate the T-code:

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

22. (858 ONLY) Add acceptance of function definitions. Function definitions are just like procedure definitions except that they have a result expression following their declarations and statements block.

23. (858 ONLY) Add acceptance of user function calls to the `FunctionOperand` rule. Function calls are just like procedure calls except that they yield a result expression.

**Changes to semantic.pt**

1. Change the semantic operations, type values, input tokens, T-code tokens to those generated in `semantic.def` when `semantic.ssl` is compiled by S/SL. (Just copy and paste the contents of `semantic.def` where indicated by comments in the `semantic.pt` source code.)

2. 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 reference `String` instead.

3. Change the predefined type "text" to reference `String` instead of `Char`.

4. Add cases for the new semantic operations `oSymbolTblStripScope` and `oSymbolTblMergeScope` to the SslWalker.

The implementation of `oSymbolTblStripScope` is like `oSymbolTblPopScope` except that it should not decrement the lexical level. That is, it just changes all the `identSymbolTblRefs` for the symbols in the top scope to their `symbolTblLink` values, and that's all. (This has the effect of removing them from visibility even though they are technically still in the table. A bit of a hack, but easy and correct.)

The implementation of `oSymbolTblMergeScope` is easy - it just has to decrement the lexical level without changing any ident links.

5. Change `oAllocateVariable` to handle allocation of `Strings` (size 256).

6. Change all the assertions that insist on the top of the `SymbolStack` being `syProcedure` to allow for `syPublicProcedure` as well. If you are in 858, these must be changed to allow `syFunction` and `syPublicFunction` also.