CISC/CMPE 458 Course Project Winter 2023
Phase 3. Semantic Analyzer
K. Jahed – February 2023

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 Quby. 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 OnQ. 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. So, for example, we will continue to use the PT T-codes for "procedures", even though Quby declares them using the keyword **def**. 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 Quby Parser and the semantic phase output T-code list to include the new Quby T-codes used in the extensions listed below.

Extensions to the T-Code Machine Model

The Quby abstract machine has the following new T-code instructions to handle Quby 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 Quby 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".

tLiteralString (string)[1]	*StringStack<— (string)*
tFetchString	*L<— Stack; SR <— Memory[L]; StringStack <— SR*
tAssignString	*SR <— StringStack; L<— Stack; Memory[L] <— SR*
tStoreParmString	*SL <— StringStack; R<— Stack; Memory[R] <— SL*
tSubscriptString	*R<—Stack; L<—Stack; Stack<—(L+R)*
tConcatenate	*SR <— StringStack; SL <— StringStack; StringStack <— (SL + SR)*
tIndex	*SR <— StringStack; SL <— StringStack; StringStack <— (SL ? SR)*
tSubstring	*SR <— StringStack; R <- Stack; L <— Stack; StringStack <— (SR $ L .. R)*
tLength	*SR <— StringStack; Stack <— (# SR)*
tChr	*R <— Stack; SR <— chr(R); StringStack <— SR*
tOrd	*SR <— StringStack; R <— ord(SR[1]); Stack <— R*
tStringEqual[2]	*SR <— StringStack; SL <— StringStack; Stack <— (SR == SL)*

[1] Note that *tSubscriptString* implicitly implements scaling by the width of a string, in the same way *tSubscriptInteger* implements scaling by the size of an integer.
[2] Note that equality and inequality (== and !=) are the only comparison operators defined on Quby 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 Quby abstract machine also has the following additional T-code instructions to handle Quby **do** loops and **case** statement **else** clauses:

tDoBegin	*null operation*
tDoBreakIf	*null operation*
tDoTest	*(address)*
tDoEnd	*(address)*
tCaseElse	*null operation*

Statements and Declarations

Unlike PT, Quby allows for the arbitrary mixing of declarations and statements. The easiest way to implement this is to merge the alternatives of the *Statement* rule into the *Block* rule and modify the *Statement* rule to push a new scope, call the *Block* rule, and finally pop the scope. Remember to remove handling of the PT **begin** statement, and accept the *sBegin* and *sEnd* token at the beginning and end of The *Block* rule instead.

Modules

Add handling of the declaration of anonymous modules as specified in the Quby language specification. Specify a new symbol table entry kind *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 (declared as **def *** in Quby) 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. Quby modules export their public procedures "unqualified" - that is, if module *M* has public procedure *P*, then it is called from outside the module in Quby 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 *identSymbolTableRef* for the identifier to the *symbolTableIdentLink* 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 decrementing 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.

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 1024 (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 process the declaration:

That is, you should process the declaration:

```
        val littleString = "Hello mom”
```

exactly as you would process 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!)

Quby 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*). Quby 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 **do** statement

Remove handling of the PT **repeat** statements. Add handling of the **do** statement as specified in the Quby language specification. The following template gives the T- code implementation of Quby loops. 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 **break if** part.

```
do                          dolabel:         tDoBegin
  ...                                        ...
  break if expression                        tDoBreakIf
                                             (T-code for expression)
                                             tDoTest     exitlabel
  ...                     .                  ...
end                                          tDoEnd dolabel
...                         exitlabel:       ...
```

The **case** Statement **else** clause

Change PT **case** statement to handle Quby case statement **else** clauses. The following template gives the T-code implementation of Quby case **else** clauses. Remember that they are optional.

```
case expression                              tCaseBegin
                                             (T-Code for expression)
                                              tCaseSelect tablelabel
  when value1 then          label1:
    ...                                      ...
                                             tCaseMerge endlabel
  when value2 then          label2:
    ...                                      ...
                                             tCaseMerge endlabel

  (... more
  alternatives ...)
                           tablelabel:
                                             (case branch table)
  else                                       tCaseElse
    ...                                      ...
                                             tCaseMerge endlabel
end                        endlabel:         ...
```

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

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. If you notice anything missing or incorrect, please post a question on OnQ.

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*, *tIndex* and *tStringEqual*. Remove the old T-codes for the while operations, and replace the old T-codes for the repeat operations with the new **do** loop operations *tDoBegin*, *tDoBreakIf*, tDoTest and *tDoEnd*. Add the new *tCaseElse* T-code.

3. Add a definition for *stringSize* to the type Integer. The value is 1024.

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.

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 Quby, 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 108 for *trReadString* and 109 for *trWriteString* (which are the trap numbers I have assigned to them in the Quby 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. Add a new rule *ModuleDefinition* to handle modules. A *ModuleDefinition* is much like Program except it has no program parameters and no halt. At the end of the module it uses the new *oSymbolTblStripScope* and *oSymbolTblMergeScope* symbol table operations to promote all public symbols to the enclosing scope.

10. Merge the alternatives of the *Statement* rule into the *Block* rule and modify the *Statement* rule to push a new scope, call the *Block* rule, then pop the 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 Quby general **do** loop statement. Handling do statements is just like **while** statements except that the T-codes are different and there is a statement allowed before the conditional exit.

16. Change **case** statement handling to handle the Quby optional **else** clause. The **else** 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. Add handling of the string index (?) operator to the *BinaryOperator* rule. Be careful to get the type checking right.

19. Add handling of string concatenation to the *sAdd* part of the *BinaryOperator* rule. Remember that strings are first class values in Quby, so string concatenation is just like integer addition in terms of what to do, except the T-codes are different.

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

21. Strings are first class values in Quby, 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*.

22. Change handling of string constants to act like vars instead.

<u>Changes to semantic.pt</u>
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 entire contents of semantic.def where indicated by the 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.) Unlike *oSymbolTblPopScope*, be careful not to change *symbolTableTop*, *typeTableTop* and *lexicLevelStackTop* in *oSymbolTblStripScope* since we're don't want to remove anything from the tables in this case.
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 1024).

6. Change all the assertions that insist on the top of the *SymbolStack* being *syProcedure* to allow for *syPublicProcedure* as well.