CISC/CMPE 458 Course Project Winter 2023
Phase 4. Code Generator
K. Jahed – March 2023

In this phase you will undertake the modifications to the Coder phase of the PT Pascal compiler to turn it into a code generator for Quby. These changes will be less extensive than those undertaken in phase 3, but still challenging. The most difficult part of this phase will be understanding the Intel x86 target machine and the fundamental mechanisms of the Coder, particularly the Operand and Emit mechanisms.

Because these mechanisms are simpler and smaller than those of the Semantic phase, and because you are now completely familiar with S/SL, this phase will be easier to understand. Most of the changes will involve copying or mimicking parts of existing PT coder rules. The following suggestions are provided to guide you in your modifications to the code generator.

Suggestions for implementing Phase 4

Rather than detail the actual changes to be made in this phase, I will simply suggest the Intel x86 code sequences that you should use to implement the new T-Code abstract machine instructions. Once you understand the workings of the Coder, implementation of these sequences should not be difficult, since it is very similar to sequences already generated by the Coder.

A good way to work on the Coder phase is to study the Intel x86 code that is emitted by the PT compiler for some simple PT Pascal programs. You can use the compile command "ptc -S prog.pt" to see the Intel x86 assembly language output in the file "prog.s". You can also find the assembly code emitted by my own Quby compiler for the Quby examples in the examples directory mentioned above.

The following templates assume that you continue to represent the PT Expression Stack (ES) in Intel x86 registers, as the PT Coder presently does, and that the *StringStack* is represented by the addresses of the corresponding string values, in Intel x86 registers also (see details below).

Code Templates for Quby Modules
This is where the pay-off for all the work you put into your semantic phase comes in. Quby modules are a scoping concept only - and since you have already implemented all scoping restrictions in the semantic phase, there are no module T-codes to implement!

Code Templates for Quby Do Loops
The implementation of Quby loops is similar to the implementation of PT **while** loops. The following are suggested Intel x86 code templates for the Quby **do** loop T-codes.

T-Code Instruction	*Intel x86 Code Implementation*
tDoBegin	bN:
(statements)	*(code for statements)*
tDoBreakIf	*(no code)*
(expression)	*(code for expression)*
tDoTest (exit code loc)	je fN
(statements)	*(code for statements)*
tDoEnd (do code loc)	jmp bN
	fN:

Look at the S/SL rule that handles **while** loops, and study the PT Coder generated assembly language output using the -S compiler flag, to figure out how to generate this code. The **do** statement is just like the **while** statement, except that the exit condition is reversed and statements can appear before **break if** as well as after.

Code Templates for Quby Case Else
The implementation of Quby **else** clause of **case** statements is a relatively simple addition to the existing handling of PT case statements. Everything remains the same, except that the rule *EmitDefaultCaseAbort* must check for a *tCaseElse* and emit the code for the **else** clause statements as if they were a *CaseVariant* (but without a case label) in place of the case abort

trap call. (If there is no *tCaseElse*, it should still emit the case abort trap call as before.)

Code Templates for Quby String

The Intel x86 machine, being a register-based computer, makes it tricky to handle long values such as Quby strings (which are 1024 bytes) conveniently, so we are going to use a quite different way to represent the *StringStack* on the Intel x86 machine. The way we will do it is to represent each string value by its address.

String variables will be accessed using their address, and results of string operations will be represented by the address of the string result, stored in a temporary register just as if it were an integer or other simple value. (This way of doing things is exactly what Java and other object-oriented languages do to for objects.) The result itself will be allocated in memory by the string operation library routines, which are implemented as a new set of Quby trap routines that we simply call. (We could optimize string operations by generating custom inline code for each one, but this way is simpler.)

The following set of new Quby trap routines has been provided to help you implement strings. For each string operation, rather than generate direct code for the operation, we will generate code to call the corresponding trap routine. Like the PT Pascal trap routines, all of these new trap routines take their parameters on the Intel x86 stack (in reverse order) and leave their result in the standard Intel x86 result register, %eax.

Quby Trap Code	*Routine Name*
pttrap101	PTXAssignString
pttrap102	PTXChrString
pttrap103	PTXConcatenate
pttrap104	PTXSubstring
pttrap105	PTXLength
pttrap106	PTXStringIndex
pttrap107	PTXStringEqual
pttrap108	PTXReadString
pttrap109	PTXWriteString

The following Intel x86 code templates for Quby T-code string operations use these new traps. In the templates below, %T and %X represent a temporary register used by the template, and can be any temporary register not in use at the time.

The example templates assume that all string operands are string variables, and so the "load address" instruction (*lea*) is used to get the address of the string. When the operand is not a string variable, appropriate other code must be used to get the address of the string value into the register or on the stack as shown. In particular, if the string is the result of a previous string operation, represented by its address in a temporary register, then its address can be moved directly instead of using a load address. Effectively, all strings, whether variables or results of string expressions, are always passed to routines as addresses by reference.

All of the string operation routines assume that every string is ended with a zero byte. The zero byte, automatically added to the text of literal strings by the *.asciz* Intel x86 assembler directive shown below, is used to mark the end of Quby varying-length strings.

Note that all templates push (save) all four x86 general purpose registers (%eax, %ebx, %ecx %edx) before each call to an external routine (e.g., trap routine) and restore them on return. This is generated by coder.ssl's *@SaveTempRegsToStack* and *@RestoreTempRegsFromStack* S/SL rules, which must be called before and after the code for each external call so that the external routine can use all the registers without fear of destroying any in use by the Quby program.
The code generated by these routines looks like this:

```
@SaveTempRegsToStack        pushl  %eax
                            pushl  %ebx
                            pushl  %ecx
                            pushl  %edx

@RestoreTempRegsFromStack   popl  %edx
                            popl  %ecx
                            popl  %ebx
                            popl  %eax
```

In order to save space in the following templates and make it clear what the actual template is, the places where this code will appear in the generated code are replaced by the generating rule names in the templates that follow. Note that arguments are passed to traps in reverse order (i.e., last argument first), and that an add to %esp is needed to pop all arguments from the stack after the call.

T-Code Instruction — *Intel x86 Code Implementation*

3 tLiteralString "string" — (declare "string" in the data area, then get its address in a temporary register %T)

```
        .data           (assemble in data segment)
slabel: .asciz "string" (.asczis adds a 0 byte)
        .text           (switch back to code segment)
        lea  slabel,%T
```

4 tFetchString — (get address of the string variable s1 in a temporary register %T)

```
lea  s1,%t
```

5 tAssignString — (where s1 is the target string variable, s2 is the assigned string value)

```
@SaveTempRegsToStack
lea  s2,%T          (%T is any temp reg)
pushl  %T
lea  s1,%T
pushl  %T
call  pttrap101
addl  $8,%esp       (pop arguments)
@RestoreTempRegsFromStack
```

6 tStoreString — (where s1 is address of formal parameter string variable, s2 is the address of the argument string expression)

7 tSubscriptString — (same as *tAssignString* above)

8 tConcatenate — (**exactly** like *tSubscriptInteger*, but scaling by 1024 instead of 4)

(takes two strings s1, s2 and leaves the address of the string result of their concatenation in temporary register %X)

```
@SaveTempRegsToStack
lea  s2,%T          (%T is any temp reg)
pushl  %T
lea  s1,%T
pushl  %T
call  pttrap103
addl  $8,%esp       (pop arguments)
movl  %eax,%esi     (save result in %esi)*
@RestoreTempRegsFromStack
movl  %esi,%X       (force result to temp)*
```

T-Code Instruction	*Intel x86 Code Implementation*
tSubstring	(takes string s1, lower index i1 and upper index i2; returns the address of the substring result in temporary register %X) @SaveTempRegsToStack pushl i2 pushl i1 lea s1,%T (%T is any temp reg) pushl %T call pttrap104 addl $12,%esp (pop arguments) movl %eax,%esi (save result in %esi)* @RestoreTempRegsFromStack movl %esi,%X (force result to temp)*
tIndex	(take two strings s1, s2 and returns the integer index of the first instance of s2 in s1 in temporary register %X) @SaveTempRegsToStack lea s2,%T (%T is any temp reg) pushl %T lea s1,%T pushl %T call pttrap106 addl $8,%esp (pop arguments) movl %eax,%esi (save result in %esi)* @RestoreTempRegsFromStack movl %esi,%X (force result to temp)*

T-Code Instruction	*Intel x86 Code Implementation*
tLength	(takes string s1 and returns its integer length in temp register %X) @SaveTempRegsToStack lea s1,%T (%T is any temp reg) pushl %T call pttrap105 addl $4,%esp (pop arguments) movl %eax,%esi (save result in %esi)* @RestoreTempRegsFromStack movl %esi,%X (force result to temp)*
tChr	(takes integer i1, converts its value to a character and returns the address of the string result in temporary register %X) @SaveTempRegsToStack pushl i1 call pttrap102 addl $4,%esp (pop arguments) movl %eax,%esi (save result in %esi)* @RestoreTempRegsFromStack movl %esi,%X (force result to temp)*

tOrd	(Note: Remove the optimizing rule *OperandChrAssignPopPop*, there are no good optimizations of this for strings.)
	
	(takes string s1; converts the first character of s1 into an integer in temporary register %X)
	
	lea s1,%T (%T is any temp reg)
	movl $0,%X
	movb (%T),%X

T-Code Instruction	*Intel x86 Code Implementation*
tStringEqual	(takes two strings s1 and s2 and returns a boolean result in temporary register %X)

```
@SaveTempRegsToStack
lea  s2,%T              (%T is any temp reg)
pushl  %T
lea  s1,%T
pushl  %T
call  pttrap107
addl  $8,%esp           (pop arguments)
movl  %eax,%esi         (save result in %esi)*
@RestoreTempRegsFromStack
movl  %esi,%X           (force result to
                         temp)*
```

tTrap trReadString	(takes string variable s1 and file descriptor f1; reads input into s1)
tTrap trWriteString	(takes string s1, width i1 and file descriptor f1; writes s1 to output)

(Both traps will be handled automatically by the existing code generator logic if you get handling of string expressions and variables right, so there is nothing new to do.)

*__Note:__ Because the concatenate, substring, length, index, chr and equal trap routines pttrap103, pttrap104, ptttrap106, pttrap105, pttrap102 and pttrap107 are functions, they return their result in register %eax. Restoring the temporary registers would destroy it, so we temporarily put it in a scratch register (%esi) until after the restore. See the PT coder's *OperandEofFunction* rule for how to generate this code.

Phase 4 Changes Checklist

Phase 4 is more challenging than the first two phases, but easier than phase 3. A lot of it is just copying and reusing existing parts of PT coder rules. 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.

Changes to coder.ssl

1. Add the input T-codes for tFetchString, tAssignString and all the other new ones produced by your semantic phase to the Input section, and remove any for Char, repeat and other ones that were deleted. Make sure that the list is identical to the Output T-code tokens in your semantic.ssl.

2. Add the trap codes for the new Quby run time monitor string operation traps - make sure that the values are the ones listed in the phase 4 handout (i.e., trAssignString = 101, trWriteString = 109, and so on). Make sure that your semantic phase is also using the correct trap numbers (old trWriteString was 8, it's now 109).

3. Add a string kind to the DataKinds. Value doesn't matter, 3 works.

4. Remove the oOperandPushChar semantic operation. Add the oOperandPushString operation.

5. Because we have no distinction between statements and declarations in Quby, statement T-codes get mixed in with declaration initialization T-codes, so the old assumption in the rule *Block* that statements follow declarations no longer holds. Modify it to accept all the statement T-codes directly in its main declaration-handling loop by copying all the alternatives in the *Statements* rule directly into it instead of calling *Statements* afterwards. Conversely, wherever statements can appear in Quby, declarations can, so replace the entire body of the *Statements* rule with a simple call to the new *Block* rule.

6. Remove handling of tLiteralChar and tStoreChar whereever they occur. Add handling of tLiteralString and tStoreString instead. Remove handling of tStringDescriptor and tSkipString whereever they occur. The new tLiteralString can be handled just like tSkipString used to be, except that no tStringDescriptor T-code is accepted. Remove handling of Char operations tAssignChar and tSubscriptChar. Don't remove the rules they call (because they are also used to handle Booleans).

A done by whoever is doing tAssignString

B done by whoever is doing tSubscriptString

3-16

7. Add handling of String operations *tAssignString*, *tStoreString*, *tSubscriptString*, *tConcatenate* etc. Remember that the operand length for string operations is Data Kind string. DO NOT make coalescence optimized cases (ones that look for immediate assignment) for String operations - you have enough to do without optimizing as well.

A 8. Add handling of String parameters in the *Routine* rule - instead of using *EmitMove*, they must use your new *OperandAssignStringPopPop* rule.

B 9. Add subscripting of String arrays. Subscripting of strings is EXACTLY like subscripting of Integer arrays except that the scaling factor is 1024 (2^10) instead of 4 (2^2). (Remember that you can multiply by 1024 by shifting left 10 bits.)

10. Add the new rules to be used by your string operations, *OperandAssignStringPopPop*, *OperandConcatenatePop*, *OperandSubstringPopPop*, *OperandIndexPop*, *OperandLength*, *OperandChr*, *OperandOrd* and *OperandStringEqualPop*. DO NOT make optimized versions such as *OperandConcatenateAssignPopPopPop*. The string operations all call trap routines at run time, so the optimizations are irrelevant. Remove the optimized cases for *tChr* and *tOrd*, which are now String operations. To implement Ord, for which we have no trap routine, simply force the address of the string into a temporary, change the mode to *mTempIndirect* and the length to byte, then force to a temporary to get the result integer (word). See the template in the assignment above.

11. Strings are passed to the string traps by address always. So to pass a String argument to the String operation traps, just force the address of the String to a temporary (*OperandForceAddresIntoTemp*), force the result to the stack (*OperandForceToStack*), then pop and free the temporary (*OperandPopAndFreeTemp*). The String result of a String trap routine is returned as the address of the result in the result register. To represent that, simply push the result register (*mResultReg*) on the Operand Stack and generate code to copy it into a scratch register (*mScratchReg1*) before restoring the temporary registers (*RestoreTempRegsFromStack*), then force the scratch register to a temporary (*OperandForceIntoTemp*). Make sure that you generate code to call the string traps exactly as other traps are called - see *OperandEofFunction* for an example.

eax

esi

force contents of esi into temp

12. Since strings are always represented by their addresses, *OperandForceIntoTemp* must be changed to have an alternative for mString that calls *OperandForceAddressIntoTemp* instead. Otherwise items of size string act like those of size word (because they are addresses), so *OperandForceIntoTemp* must be modified to allow size string as well as word.

1 13. Add handling of **do** statements by copying the *WhileStmt* rule to a new *DoStmt* rule to handle Quby do statements, and delete *RepeatStmt*. Remember that Quby do loops are just like PT while statements except that statements may appear between the beginning of the loop and the **break if** condition.

2 14. Modify the *EmitDefaultCaseAbort* rule to check for an **else** clause (beginning with the *tCaseElse* T-code) before emitting the trap call. If there is an **else** clause, handle it just like a *CaseVariant* (but without a case label), otherwise emit the abort trap just as before.

Changes to coder.pt

1 ✓ token updates

1. Change the semantic operations, type values, input tokens, output tokens, etc. to those generated in coder.def when coder.ssl is compiled by S/SL.

2. Make sure that the *firstCompoundToken* and *lastCompoundToken* values correspond exactly to the first and last T-code that have associated values generated by the semantic phase (these must be in a contiguous range). Make sure that the value of *tEndOfFile* and *lastInputToken* is exactly one more than the *lastOutputToken* T-code generated by your semantic phase.

3. In *OperandFoldManifestSubscript*, add a case to scale the subscript by 1024 if the element size is string (in the same way it scales by 4 if the element size is word). (You only need to do this if you chose to optimize manifest subscripts of string arrays in your S/SL.)

4. Remove all references and alternatives for *tStringDescriptor* and *tSkipString* wherever they occur. Change the Assert statements that reference *tStringDescriptor* to say *tLiteralString* instead.

5. Remove the case alternative for the *oOperandPushChar* semantic operation, and change the case alternative for the *oOperandPushStringDescriptor* semantic operation to be *oOperandPushString*.

6. Remove the call to *AssertAllTempsAreFree* in the *oEmitMergeSourceCoordinate* semantic operation. Now that we have string traps, it's going to fail.