CISC 458
Tutorial 6

Starting Phase 3

Revised by Ahmed Harby from Previous Versions

Thursday, March 2, 2023

Introduction

Advice

START EARLY

- Why? The semantic analyzer is complicated and it will take time to understand how it works.
- **Important**: Make sure you know how it works before you make changes (except for trivial things like updating tokens).

Files to Change: Breakdown

- Your task is to modify the files in `ptsrc/semantic`
  - `semantic.ssl`
    - Has definitions of all the rules as to how semantic analysis should work.
  - `semantic.pt`
    - Has definitions of all the tables, semantic mechanisms, and error messages.
- So, if you see `oTypeStkPop` (a semantic mechanism operation) called in `semantic.ssl`, the actual implementation is in `semantic.pt`

Semantic Operations

- Semantic operations are the words starting with "o".
- Declare (and use) new semantic operations in `semantic.ssl`, then write the actual code in `semantic.pt` under procedure `SslWalker`.
- **Optional**: For testing and debugging purposes, you might want to add a semantic operation that prints out the contents of stacks and tables.

Tracing the Semantic Phase Output

Purpose of Semantic Phase

- Recall:
  - The semantic phase in a compiler converts the language constructs in the source code to a lower-level intermediate representation.
  - Still higher level than assembler, and still not specific to a given machine or architecture.
  - The output stream consists of emitted tokens that we will refer to as *T-code instructions* or *T-code tokens*.
    - The entire output stream is referred to as *T-Code*.

New `ssltrace` Command

```
ssltrace "ptc -o3 -t3 -L lib/pt test.pt" lib/
    pt/semantic.def -e
```

- `-t3` trace the output of the semantic analyzer
- `-o3` execute only parser, scanner, and semantic analyzer
- `-L` is the direct path to your library
- `lib/pt/semantic.def` is where `ssltrace` should look for the tokens
- `-e` gives the trace of emitted tokens.

- **Note**: The `-m` option of `ssltrace` can be useful in this phase to trace the output of a specific mechanism.

Compiling the Semantic Analyzer

- Compile with "`make`" in the `ptsrc/semantic/` directory and not "`make`" in the `ptsrc/` directory
  - `make` in the `ptsrc/` directory **will require** a complete `Quby` compiler!
- **Note**: Don't run `make semantic` in the `ptsrc/` directory; the `MakeFile` in that directory does not recognize it.

Semantic Phase Trace Example

Trace Example Program

In this section, we will be looking at the trace of the following PT program, using the default PT compiler.

```
program Foo (output);
begin
end.
```

Trace Example Program

The trace is redacted for presentation purposes using the command

```
ssltrace "ptc -o3 -t3 -L lib/pt test.pt" lib/
    pt/semantic.def | egrep "^ *[\.o%]"
```

The regex (`egrep`) matches semantic operations (starting with 'o') and output operations (starting with '.' or '%').

- **Note**: Use `ssltrace` and its flags instead when tracing the semantic analyzer! The above `egrep` can remove important things like assertion failures, crashes and error messages.

Trace Example Output

```
oSymbolTblPushScope
oCountPush(three)
 oSymbolStkPushLocalIdentifier
 oSymbolStkSetKind(syVariable)
 oSymbolStkLinkToStandardType(stdText)
 oTypeStkPushSymbol
 oValuePush(two)
  oAllocateAlignOnWord
  oSymbolStkEnterDataAddress
  .tLiteralInteger
  oEmitValue
  % value emitted 2
  .tLiteralAddress
```

Trace Example Output

```
    oEmitDataAddress
    % value emitted 0
    .tFileDescriptor
    oAllocateDescriptor
  oSymbolTblEnter
  oSymbolStkPop
  oTypeStkPop
  oValuePop
oCountPop
oSymbolTblPopScope
.tTrapBegin
.tTrap
oEmitTrapKind(trHalt)
  % value emitted 0
```

Trace Example Explained

> `oSymbolTblPushScope`

Function: Push a new symbol table scope to differentiate global from predeclared

Progress: *Start processing file descriptors (program parameters)*

> `oCountPush(three)`

Function: Setup for the first 'custom' file descriptor (1 = input, 2 = output) if needed, so that we will start counting at 3.

Progress: *Ready to process "output"*

Trace Example Explained

> > `oSymbolStkPushLocalIdentifier`

Function: Process the ‘output’ identifier and push it into the symbol stack

> > `oSymbolStkSetKind(syVariable)`

Function: Set the kind of the ‘output’ symbol to a variable

> > `oSymbolStkLinkToStandardType(stdText)`

Function: Set type reference to text (file contains text data)

Trace Example Explained

> > `oTypeStkPushSymbol`

Function: Push an entry onto the type stack, copying its attributes from the symbol type reference (text in this case)

> > `oValuePush(two)`

Function: Push the value of the identifier (2 = output)

Progress: *Allocate "output" using the top entry of the symbol stack and the type stack*

Trace Example Explained

`> > > oAllocateAlignOnWord`

Function: Align the free-space pointer to a word (4 bytes) boundary

`> > > oSymbolStkEnterDataAddress`

Function: Set address field of top of symbol stack to free-space pointer

`> > > .tLiteralInteger`

`> > > oEmitValue % value emitted 2`

Function: T-Code instruction that an integer value follows and is emitted (value is 2)

Trace Example Explained

> > > `.tLiteralAddress`

> > > `oEmitDataAddress` % value emitted 0

Function: T-Code instruction that an address follows and is emitted (value is 0). Of course we start at the beginning = 0.

> > > `.tFileDescriptor`

Function: T-Code instruction for a complete file descriptor consisting of the integer value and its target address.

> > > `oAllocateDescriptor`

Function: Move the free-space pointer up by the size of the file descriptor.

Progress: *Done allocation for "output"*

Trace Example Explained

> > `oSymbolTblEnter`

Function: Make a new symbol table entry from the top entry of the symbol stack.

Progress: *"output" is in the symbol table, so now we need to clean up the stacks*

> > `oSymbolStkPop`

Function: Pop the 'output' entry off the top of the symbol stack

> > `oTypeStkPop`

Function: Pop the 'output' entry off the top of the type stack.

> > `oValuePop`

Function: Pop the 2 off the top of the value stack.

Progress: *"output" has been allocated completely*

Trace Example Explained

> `oCountPop`

Function: Done processing program parameters (files), so pop the counter for custom file descriptors (unused in this case).

Progress: *Ready to process body of program, which is non-existant in this case*

> `oSymbolTblPopScope`

Function: Done the program, pop the global scope off of the symbol stack.

Progress: *Halts the program*

> `.tTrap`

> `oEmitTrapKind(trHalt)` % value emitted 0

Function: T-code instruction that a halt trap (instruction) follows and is emitted (0 = proper exit).

Progress: *Halts the program*

Changes Needed in Phase 3

Definitions

- Start off slow and do things you already know how to do:
  - Add the new semantic output tokens to `semantic.ssl` in the same order you added them in `parser.ssl`
- Check that the order of your tokens are in sync!
- **Reminder**: Copy the contents of `semantic.def` into `semantic.pt` where indicated by the `===` comments!

Definitions

- Add the new T-code instructions to handle `Quby` string operations
  - `tFetchString`
  - `tConcatenate`
  - `tSubstring`
  - `tLength`
  - `tIndex`
  - `tStringEqual`
- Add a definition for stringSize to the type Integer. The value is 1024.
- 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`
  - `tDoEnd`
- Add the new `tCaseElse` T-code .

Programs

- The output by the `Quby` parser implemented in Phase 2 is similar to the PT Pascal one
  - Nothing to do!
- Test using the null `Quby` program

```
using output
```

Blocks

- **Remember**: `Quby` allows both declarations and statements to be intermixed
  - Merge the alternatives of the `Statement` rule into the `Block` rule
- Move `sBegin` to the beginning of the rule
- Remove handling of the `begin` statement which is not in `Quby`.
- Replace the old `Statement` rule with one that pushes a new scope, calls the `Block` rule, then pops the scope.
- Make sure to test these changes with a program that uses a variety of declarations and statements to make sure things are still working.

Constant and variable declarations

`Constant` and `Variable` declarations

- Modify handling of constant definitions to allow only one per definition.
- Modify handling of type definitions to allow only one per definition.
- Modify handling of variable declarations to allow multiple identifiers declared using one type, but only one declaration per definition.
  - Push all the declared identifiers on the Symbol Stack and keep count of how many there are using the Count Stack.
  - Enter in the `SymbolTable` all the identifiers one at a time.
  - **Remember** to keep the stacks straight and clean up after you're done.

Statements

The `do` Statement

- Remove handling of the PT repeat statements.
- Add handling of the do statement as specified in the Quby language specification.
- 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.

The `case` Statement else clause

- Modify `PT` `case` statement to handle the default alternative `else`
  - Reuse the existing `CaseStmt` rule 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).
    - Emit `tCaseElse` through `tCaseMerge` after the table.
    - The else clause is much like another case alternative, except emitted after the tCaseEnd.

The `elseif` Clause

- This is where your group's decision on `elseif` in Phase 2 affects the amount of work needed in Phase 3.
  - If you handled it completely in Phase 2 (treating an `elseif` as an `if` within the `else` block), there is nothing more to do.
  - If you outputted an `sElseIf`, you will need to handle the `elseif` as if you received the equivalent sequence of a nested `if` within an `else`.
    - Make sure you get exactly the equivalent T-code. Test by making two programs one with `elseif` and another with `else { if ...}` and compare the output T-code.

Modules

Modules

- The T-code implementation of modules does not involve any new T-codes.
- Add a new rule ModuleDefinition to handle modules.
- The only tricky part is to continue execution after the statements.
- **Remember**: Quby modules export their public procedures "unqualified" (e.g. if module M has public procedure P, then it is called from outside the module in Quby simply as P, not as M.P).
- Add two new `SymbolTable` mechanism operations: `oSymbolTblStripScope` and `oSymbolTblMergeScope`
  - Call both of these instead of oSymbolTblPopScope when processing the end of a module.

Strings

Strings

- Re-purpose types and T-code names (e.g., `pidChar`, `tpChar`, `tFetchChar`) to handle `Quby` strings instead of PT chars
- Remove the tSkipString and tStringDescriptor stuff in the T-code for string literals.

String Allocation

- Define `stringSize` as 1024 bytes under type `Integer` in `semantic.ssl`, which also contains `byteSize` and `wordSize`.
- In type `StdType`, change `stdChar` to stdString.
- 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.

String Operations

- Remember the new string operations are added in `Quby`!
  - `sSubstring` – substring operation
  - `sConcatenate` – string concatenation
  - `sLength` – string length
  - `sIndex` – string Index
  - `sStringEqual` – string equality
- Some pointers
  - Strings are first class values. Operations on integers can often be used as templates.
  - All changes for this should be to the `S/SL` `UnaryOperator` and `BinaryOperator` rules.
  - For all operations, make sure appropriate type checking is done
    - **Hint**: Use `oTypeStkChooseKind` and the error `#eTypeMismatch`.

Suggestions

Basic Suggestions

- Make use of the checklist provided in the assignment description to make sure you don't miss anything.
  - If you notice anything missing from the checklist, please post it on the forum to help everyone.
- As always, ask questions on the forum.
  - Please contact one of us if you find yourself working more than 12 hours on a problem, although I'd advise doing it much earlier.
  - If your group has abandoned you to a problem you believe is much more difficult than the others, please let us know.

Basic Suggestions

- Refer to `semantic.ssl` for an explanation of the various semantic mechanisms and operations in comment form.
  - **Optional**: Add comments to any semantic operations you introduce in a similar manner.
- if you don't trust your phase 2 code, you can use the (path) to the solution on OnQ as a base for Phase 3.
  - This should be a last resort, and you should note it in your documentation. We'd prefer you not do this if your `parser` code is fine.

Advanced Suggestions

- Trace simple `PT` programs using the default `PT` compiler to understand how the semantic analyzer works, especially how the stacks/tables are used.
  - Do this as a group, which means communicating to each other on how some mechanisms may work.
  - Don't withhold how a mechanism might work from others if you split up the work as a group! *It's not cool ¿:{*

Advanced Suggestions

- Before splitting the work, identify which items are dependent on others
  - The work is difficult to split up properly; look at the checklist for help
  - No perfect way to split up the work. It depends on the individuals of the group.
  - At the very least, change and update the tokens before splitting up the work, and make sure the semantic phase at least compiles.
  - Testing should be done as a group; or each member creates own test cases to test the changes they are responsible for.
- Testing is your friend.

Most Important Suggestions

START NOW!
DO NOT START LATER!

Next Tutorial

- More on Phase 3