G53CMP Compilers: Coursework Part II

Autumn, Academic Year 2013/14

Henrik Nilsson
School of Computer Science
University of Nottingham

November 10, 2013

1 Introduction

This document details Part II of the assessed coursework for the module G53CMP Compilers. The weight of Part II is 15 % of the overall mark. (The weight of Part I was 10 %, for an overall coursework weight of 25 %.) As in Part I, the coursework is centred around the compiler HMTC, implemented in Haskell, for the small language MiniTriangle. However, this version of MiniTriangle has been extended with functions, procedures, and arrays, making it a much more interesting language. The compiler itself has also been extended to a complete compiler, including type checking and code generation. Finally, all the *front-end* extensions from Part I have been implemented, giving everyone a common starting point and allowing this version of HMTC to serve as a model solution for Part I, against which you can compare your own solution in the light of the feedback you have received on your work on Part I.

You are referred to the Part I description for pointers to resources for general background on Haskell, GHC, and the School's Haskell installation. You are also referred to Part I for details regarding assessment and feedback: exactly the same procedure and rules are used for Part II, including the compulsory oral examination and its role. You are reminded that this coursework is to be carried out *individually*. This means that while you are welcome to discuss the coursework with friends, on the forum, or the module team, in the end, you must solve the problems on your own and demonstrate that you have done so by being able to explain your solutions as well as their wider context.

In this second part of the coursework, you will continue the work on the extensions of the MiniTriangle language by extending the type checker and the code generator to work with the new or extended language constructs from Part I. As discussed above, the scanner and parser of the new version of the MiniTriangle compiler already extended to handle the new features from the first part. The type system for the version of MiniTriangle we are using is explained and specified (using typing rules) in Appendix C. You will need to familiarise yourself with this. However, the appendix goes into a lot more detail that you will actually need, so it is recommended to read it through, so go through it fairly quickly at first, and then use it as a reference.

The target language of the compiler is Triangle Abstract Machine (TAM) code. The TAM is a simple stack machine, a simplified version of the one introduced in Watt & Brown's book. The code you have been given also includes a TAM interpreter, and you can instruct the compiler to run the generated code directly via command line options: see Section 3 below. See Appendix D for a specification of the TAM instructions. The lecture slides do also go into a fair amount of detail. Of course, for the definitive account of what instructions there are and to understand exactly what each instruction does, you can always refer to the source code of the interpreter.

2 Submission

For practical information about deadlines, timetables for oral vivas, and so forth, see the main module web page and the coursework support page (linked from the main page). There you can also find links to other practical information; such as on electronic submission.

For Part II of the coursework, the following has to be submitted by the deadline:

- A brief written report as specified below.
- The complete source tree for the extended compiler.

Pay close attention to the report requirements. The report should be a self-contained record of what you have done, and you will lose marks if it does not contain all relevant information. Submission of source code is mainly to facilitate testing.

The submission is part physical, part electronic:

- *Physical:* hard copy of the report to the School office
- *Electronic:*

  - Electronic copy of the report (PDF). The file should be called `xyz99u-report-partII.pdf`, where `xyz99u` should be replaced by your School of CS user ID.
  - Archive of the source code hierarchy, either gzipped TAR or zip. The archive should be called
    * `xyz99u-src-partII.tgz` or
    * `xyz99u-src-partII.zip`,

    where `xyz99u` again should be replaced by your School of CS user ID. The archive should contain a single top-level directory containing all the other files.

The written report should be structured by task. For each task:

- *Brief* comments about the key idea of the solution and any subtle aspects; a few of sentences would normally suffice.
- Answers to any theoretical questions, such as any new or changed typing rules.
- All *added* or *modified* code, with enough context to make an incomplete definition easy to understand. (Thus, you should *not* include all code!)
- Anything extra that the task specifically asks for.

To exemplify the point about added and modified code, if you:

- have added a new function, then include the complete function definition, including the type signature;
- have extended a lengthy function with a few cases, then include the new cases (along with immediately surrounding cases to the extent needed to make it clear `where` the extension was made when necessary to make the order of pattern matching clear);
- have added a constructor to a datatype, include the definition and state the name of the extended type explicitly.

3 Getting Started

As in Part I, it is assumed in the following that you are going to use the Haskell system GHC on the School's Linux/Unix servers. There are alternatives; e.g. you could use the Haskell installation on the School's windows

machines, or you could work on your own machine. See part I for a discussion on this, including various caveats and, where applicable, workarounds.

First download the archive that contains the HMTC source code for Part II along with some MiniTriangle test programs. There are links from the G53CMP Compilers module web pages. To unpack the archive:

```
clyde$ tar zxvf G53CMP-CWPartII.tgz
```

The directory `SrcPartII` contains the source code for HMTC relevant to Part II and a further subdirectory containing some MiniTriangle test programs. A makefile (called `Makefile`) specifies how to build HMTC and its documentation. To compile HMTC, change directory to the HMTC source code directory and invoke GNU Make. Building the compiler is the default goal in the makefile, so no further arguments to GNU Make are needed:

```
clyde$ gmake
```

Note that the makefile is written specifically for GNU Make, so other versions of Make will likely not work. However, on a typical Linux system, `make` is just another name for `gmake`.

The HMTC source code is well-documented, and the source code comments have furthermore been formatted to allow the generation of separate, typeset, documentation by means of Haddock, the Haskell Documentation system. To create this documentation, invoke GNU Make with the goal `doc`:

```
clyde$ gmake doc
```

At present, hyper-linked, indexed, HTML documentation is built. All documentation files are created in the subdirectory `Doc`. To view the documentation, point your browser to the file `Doc/index.html`. Browsing through this documentation is an excellent way to get familiar with HMTC. Make sure you understand how the hyper-linking works and to check out the various indices.

When building the documentation you will likely see warnings from Haddock that it cannot find link destinations for standard Haskell types like `GHC.Types.Int`. This just means there will be no hyper-links to those types in the generated documentation. This is not a problem and those warning can thus be ignored.

Once HMTC has been compiled, it can be invoked to compile a MiniTriangle program into TAM code as follows:

```
clyde$ ./hmtc filename
```

where *filename* gives the path of the program to compile. For example

```
./hmtc MTTests/test2.mt
```

If there are no errors, code is generated and written into a file named like the source file but with suffix `.tam` (replacing any suffix `.mt`). However, you can ask the compiler to print the intermediate representation at various stages to get some more information. For example:

```
./hmtc --print-after-codegen MTTests/test2.mt
```

or even to stop after a specified phase:

```
./hmtc --stop-after-codegen MTTests/test2.mt
```

Alternatively, instead of writing the generated code to a file, it is possible to send it directly to the integrated TAM interpreter for execution: see below.

To get help with the HMTC command-line syntax, call HMTC with the `--help` option:

```
./hmtc --help
```

Important! Note that it is possible to ask HMTC to pretty-print the intermediate representation after one or more different compilation phases, and to instruct HMTC to *stop* after a specific phase. For example, if the type checker *has* been been extended to account for the new functionality, but *not yet* the code generator, it is useful to be able to stop after type checking but before code generation in order to test the type checker without causing errors such as "Non-exhaustive patterns".

At present, the only way to run the generated TAM code is via the TAM interpreter integrated with the compiler. You instruct the compiler to pass the generated code to the interpreter via the command line options `--run` and `--run-traced`. The latter variant turns on tracing, enabling you to see how the machine works.

HMTC can also compile a program provided through the standard input. Just invoke HMTC without any file name argument. This allows you to enter the program to compile directly, without first having to write it into a named file. Terminate the input by pressing CTRL-D immediately after a newline. For example:

```
clyde$ ./hmtc
let const x : Integer = 1 in putint(x)
CTRL-D
```

There is one caveat, though: after a program has been read from the standard input, the standard input stream is closed. This means that any further

attempt to read from the standard input, i.e. by directly or indirectly invoking the TAM instructions `GETINT` and `GETCHR` (see Appendix D), will fail. The solution is to write such programs to a source file and invoke HMTC on this file.

You can also test individual functions from within GHCi. In particular, there is a test function in the module `TypeChecker` called `testTypeChecker`. For example:

```
clyde$ ghci
Prelude> :load TypeChecker
TypeChecker> :type testTypeChecker
testTypeChecker :: String -> IO ()
TypeChecker> testTypeChecker "putint(1 + true)"
Diagnostics:
Error at line 1, column 12:
Expected a type compatible with "Integer", got "Boolean"

MTIR:
CmdCall <line 1, column 1>
  ExpExtRef "putint" : (Integer) -> Unit
  ExpApp <line 1, column 8>
    ExpExtRef "add" : (Integer, Integer) -> Integer
    ExpLitInt 1 : Integer
    ExpLitBool True : Boolean
  : Integer
```

The TAM interpreter can also be run directly from within GHCi by loading the module `TAMInterpreter` and then calling the function `runTAM` on a list of instructions:

```
runTAM :: Bool -> [TAMInst] -> IO ()
```

This is very helpful for getting familiar with how the TAM works and for developing and debugging TAM code; for example, for Task II.3 below.

However, in this case, in order to get easy access to the constructors for the TAM code, make sure you load the module in interpretive mode by prefixing the module name with a `*`:

```
:load *TAMInterpreter
```

This is because GHCi's behaviour regarding what definitions are in scope varies depending on whether the interpreted or compiled version (if the latter happens to be available) of a module is used. In interpreted mode, everything

that is in scope *inside* the module comes into scope at the GHCi prompt when the module is loaded, and that is what you most likely want in this case.

The first argument to `runTAM` is used to turn traced execution on or off. `True` turns tracing on which is very useful for seeing what the TAM does. For example:

```
runTAM True [LOADL 1, LOADL 2, ADD, PUTINT]
```

This results in the following output:

```
        LOADL       1                   [1]
        LOADL       2                   [2, 1]
        ADD                             [3]
3
        PUTINT                          []
TAM Halted!
```

For another example, here is a little TAM program that reads a number from the terminal, doubles it, and prints the result:

```
runTAM True [GETINT, LOADL 2, MUL, PUTINT]
```

The trace of the execution is as follows, where 21 is the number entered by the user:

```
Enter integer:
21
        GETINT                          [21]
        LOADL       2                   [2, 21]
        MUL                             [42]
42
        PUTINT                          []
TAM Halted!
```

For a final example, consider this buggy TAM program:

```
runTAM True [LOADL 3, LOADL 4, ADD, MUL, PUTINT]
```

Trace:

```
        LOADL       3                   [3]
        LOADL       4                   [4, 3]
        ADD                             [7]
Stack underflow!
        MUL                             [7]
```

For larger pieces of code, you might find it more convenient to implement your own Haskell module in which to develop you code, rather than doing everything interactively. For example:

```
module MyTAMCode where

import TAMCode
import TAMInterpreter

addAndPrint = [LOADL 1, LOADL 2, ADD, PUTINT]
```

This code can now be tested as follows:

```
:load MyTAMCode
runTAM True addAndPrint
```

See Appendix D for a specification of the TAM instructions. The lecture slides also go into a fair amount of detail, and the file `TAMCode.hs` gives a complete albeit brief description of all TAM instructions. Of course, to understand exactly what each instruction does, you can always refer to the source code of the interpreter in `TAMInterpreter.hs`, which should be fairly easy to follow.

Familiarise yourself with the complete version of HMTC for Part II. For instance, try out HMTC on various MiniTriangle examples. Go through the documentation in order to become familiar with the various modules.

4 Tasks

Part II of the coursework is concerned with extending the type checker and the code generator of HMTC to work with the new or extended MiniTriangle language constructs introduced in Part I. As discussed in the introduction, the version of the MiniTriangle compiler for Part II includes a type checker and a code generator for the *unextended* MiniTriangle language, as well as a scanner and parser *already extended* to handle the new features from the first part.

Task II.1 (Weight 15 %) In Part I of the coursework, you added the following new or extended constructs to the MiniTriangle language:

- `repeat-until`-loops
- conditional expressions

- extended syntax for the `if`-command (`elsif` and optional `else` branch)
- character literals

However, Part I was only concerned with extending the scanner and the parser. The next step is to extend the type checker. As a preparatory step for this, extend the MiniTriangle type system as detailed in Appendix C by adding new rules or modifying existing ones to cover the above language extensions.

In order to formulate concise typing rules for language constructs encompassing sequences of subterms, it is recommended to use vector notation where appropriate as discussed in Appendix C and the lectures. For example, the rule for the version of the `if`-command without the optional `else` might look something like:

$$\frac{\ldots \quad \ldots \quad \ldots \quad \ldots}{\ldots \vdash \texttt{if}\ e_1\ \texttt{then}\ c_1\ \texttt{elsif}\ \overline{e_2}\ \texttt{then}\ \overline{c_2}} \quad \text{(T-IF}\ldots\text{)}$$

We understand this to match an arbitrary number of `elsif-then`s, including none.

However, if you prefer, you can also formulate rules that deal with sequences by explicit induction (recursion) over the sequences. In that case, you will have also have to introduce additional syntactic categories for the sequences so as to make it clear what is what in the rules through naming conventions along the lines of the existing rules.

It should be relatively straightforward to formulate most if the rules. However, there is a caveat concerning the typing of conditional expressions. As has been discussed in the lectures, the "then branch" and the "else branch" both have to have some common type T. However, as discussed in Section C.6, due to subtyping and implicit dereferencing, an expression can sometimes have more than one type. For example, assume `x` : `Ref Integer` and `y` : `Ref Integer`, and consider a conditional expression b `? x : y` (where b is some Boolean expression). Thanks to the typing rule T-SOURCES (see Section C.7.2), the subexpressions `x` and `y` have both type `Ref Integer` and `Integer`. Thus we can pick either as their common type, meaning that the conditional expression b `? x : y` also can have either type `Ref Integer` or type `Integer`. As far as the type system as such goes, this would be fine.

However, the chosen strategy for *implementing* the type system in the form of the HMTC type checker is such that it is necessary to pick a specific type early (see Section C.8). In this case, this means that unless the typing rule for conditional expressions imposes additional constraints (besides the type of the two branches being the same) to narrow down the possible types,

it becomes difficult to implement the rule faithfully. The easiest solution in this case is to insist that the common type must not be a reference type in a similar manner as in the rule T-ASSIGN for assignment. However, you may wish to explore more flexible alternatives: it can of course be quite useful to have the possibility to select between two references.

You don't have to put a lot of effort into typesetting your typing rules. If you want, you can use a naming convention such as a suffix "s" to indicate vectors, rather than the overbar notation, for example. Neatly hand-written and scanned (photographed) rules are also perfectly fine.

For reference, the relevant productions for the extended *abstract* syntax are as follows:

Command	→	...	
	|	`repeat` *Command* `until` *Expression*	CmdRepeat
	|	`if` *Expression* `then` *Command*	CmdIf
		(`elsif` *Expression* `then` *Command*)*	
		(`else` *Command* | ϵ)	
Expression	→	...	
	|	*CharacterLiteral*	ExpLitChr
	|	*Expression* `?` *Expression* `:` *Expression*	ExpCond

Task II.2 (Weight 35 %) Now extend the actual type checker to handle the language extensions according to the extended type system you specified in Task II.1. This also means you will have to add a new type `Character`, and that you will need to extend the MiniTriangle standard environment correspondingly to make this type available to the programmer.

The back-end of the compiler is structured around a new abstract syntax representation called MTIR (MiniTriangle Intermediate Representation) that carries additional semantic information. The type checker, besides checking the types, translates the initial AST into the enriched MTIR. While the the new language constructs from Part I have been added to the initial abstract syntax AST (defined in the file `AST.hs`), they have (with the exception of literal characters) *not* been added to MTIR. Thus you will also have to extend MTIR (defined in `MTIR.hs`) to handle the new constructs.

You will have to modify the following files:

- `Type.hs`
- `MTStdEnv.hs`
- `MTIR.hs`

- `PPMTIR.hs`
- `TypeChecker.hs`

When you add the new character type to `Type.hs`, do not forget to also extend the Eq and Show instances. The changes to `MTIR.hs` and `PPMTIR.hs` are so straightforward and follows `AST.hs` and `PPAST.hs` so closely that it suffices to include only the added or modified code for these files in your report; that is, no explanations needed beyond making it clear where the code belongs.

It is strongly suggested that you implement one language construct at a time (i.e., make all relevant changes in all relevant files for one construct before you move on to the next) in the following order:

- character literals
- `repeat-until`-loops
- conditional expressions
- extended `if-then-else`-commands

As mentioned above, `MTIR` (and the pretty printing function) has already been extended to account for literal characters. When `MTIR` is modified to account for the extended `if-then-else` syntax, you may find that it is necessary to temporarily comment out the corresponding case in the code generator (file `CodeGenerator.hs`) in order to compile the compiler. You will return to extending the code generator in Task II.4 below.

When implementing the rule for checking conditional expressions, keep the discussion from Task II.1 about the caveat related to this rule in mind. If you followed the advice from that discussion, a hint is to use the function `infNonRefTpExp` where appropriate.

Note that the type checker is structured using the diagnostics monad to facilitate collection of error messages and warnings. Behind the scenes, all messages are still collected into a list in exactly the same way as we have seen explicitly in many of the lectures. However, the diagnostics monad hides the "plumbing", allowing code that needs to emit diagnostics to focus on the essence of the algorithm in question, here type checking. You don't need a detailed understanding of how the diagnostics monad is implemented to successfully complete this task: just use the provided monadic actions such as `require` and `emitErrD` in combination with the `do`-notation in pretty much the same way as if you were writing imperative code. (If you want to explore

alternative approaches to type checking the conditional expression, you may also want to be aware of the combinator `|||` for diagnostic computations.)

Hint: You may find the following function from the Haskell prelude useful for processing a list of things in a monadic context:

```
mapM :: (Monad m) => (a -> m b) -> [a] -> m [b]
```

The idea is that a monadic action on a single element is "lifted" to work on lists of elements. Or, if you prefer, it is like a loop over a list in which the monadic action is carried out on each list element, and each individual result gathered into a list of results.

Task II.3 (Weight 15 %) The target of the HMTC compiler is the Triangle Abstract Machine (TAM) and the target language is thus TAM code. The aim of this task is to gain familiarity with the TAM and TAM code through implementing some simple TAM programs. Additionally, the MiniTriangle standard library is going to be extended with a couple of subroutines to make it possible to read single characters from and write single characters to the terminal now that you have enriched the language with a character type and character literals.

See Appendix D and the lecture slides for explanations of the TAM instructions. See Section 3 above for hints on easy ways to write and test TAM code. When typesetting TAM code in your reports, you can either:

- cut and paste the Haskell representation of your code sequences into your report, preferably laid out with one instruction per line for ease of reading; or
- transliterate the Haskell representation to the slightly neater style used in the lectures and in the TAM specification.

Furthermore, for (a) and (b) below, in addition to writing up your answer in the report, you should add a module `MyTAMCode.hs` to the source tree that includes the *executable* TAM code you wrote for these subtasks. Name your definitions in a way that makes it clear which answer is which.

(a) Implement a TAM program that reads a number n from the terminal and then prints the numbers from 1 to n (in that order). Use a *loop*. Make sure no numbers are printed if $n < 1$.

(b) Implement a *recursive* TAM function `fac` to compute $n!$, the factorial of n. (That is, you should implement this function in TAM code; you are *not* asked to implement a new TAM instruction.) Arguments and

results should be passed and returned on the stack. The function should return 1 for any argument $n \leq 0$. Further, the function should be *pure*: no side effects on any global variables.

Then use this function in a TAM program that reads a number n from the terminal and prints fac(n).

(c) Extend the MiniTriangle standard library, and thus also the MiniTriangle standard environment, with a procedure `getchr` to read a single character from the terminal, and a procedure `putchr` to write a single character to the terminal. The procedure `getchr` should write the read character to a variable (of type `Character`) passed by reference, while the procedure `putchr` should expect a single argument (of type `Character`) passed by value. The MiniTriangle type signatures should thus be as follows:

Name	Type
`getchr`	(`Snk Character`) → `Void`
`putchr`	`Character` → `Void`

The TAM instructions for reading and writing characters are `GETCHR` and `PUTCHR`, respectively.

Make sure you test your code! Then incorporate the code into the MiniTriangle standard library that can be found in the file `LibMT.hs`, and modify the standard environment defined in `MTStdEnv.hs` accordingly. You will have to modify the following files:

- `LibMT.hs`
- `MTStdEnv.hs`

Task II.4 (Weight 35 %) Extend the code generator so that code is generated properly for the language extensions. The code generator has already been extended to handle character literals. Thus it remains to extend the code generator to generate code for:

- `repeat-until`-loop (recall the dynamic semantics: repeat body at least once until condition becomes true)
- conditional expressions
- extended `if-then-else`-command

In addition, the function `sizeOf` needs to be extended for the new type `Character` (just uncomment the relevant case). It is strongly suggested that you implement one language construct at a time in the order suggested above.

You will have to modify the following file:

- `CodeGenerator.hs`

Note that the code generator is structured using the code generator monad (`TAMCG = CG TAMInst ()`) to facilitate joining generated code fragments into a single sequence and generation of fresh names (i.e., names guaranteed to be distinct within the generated program) for use, for example, as labels. Behind the scenes, the generated instructions are still joined into a list in exactly the same way as we have seen error messages being explicitly gathered into lists in many of the lectures, and the name generation is based on threading through a counter in exactly the same way as we saw in the tree numbering example in the lecture on monads. However, the code generation monad hides the "plumbing", allowing code structured using this monad to focus on the essence of the code generation algorithm. You don't need a detailed understanding of how the code generation monad is implemented to successfully complete this task: just use the provided monadic actions such as `emit` and `newName` in combination with the `do`-notation in pretty much the same way as if you were writing imperative code.

Hint: You may find `mapM` from the Haskell prelude useful again, or, in case you don't care about the result from processing the list:

```
mapM_ :: (Monad m) => (a -> m b) -> [a] -> m ()
```

You can essentially understand `mapM_` as "for each element in a list, do the specified action on that element".

A MiniTriangle Grammars

This appendix contains the grammars that define the concrete and abstract syntax of the version of MiniTriangle used for Part II of the coursework. The concrete syntax is divided into two parts: lexical syntax and context-free syntax. The grammars are derived from the book by Watt & Brown. Compared to Part I, the language has been extended with procedures, functions, and arrays. However, for consistency with the typing rules of the type system (see appendix C), the grammars do not include the syntactic extensions from Part I (`repeat`-loop, extended `if`-command, etc.).

A.1 MiniTriangle Lexical Syntax

Non-terminals are typeset in italics, like *this*. Terminals are typeset in typewriter font, like `this`. Terminals whose spelling (the concrete character sequence) is different from what is shown in the grammar, such as names of special characters, are typeset in italics and underlined, like *this*. For simplicity, we resort to a slightly informal way of stating that the keywords are not valid identifiers.

Program	→	(*Token* | *Separator*)*
Token	→	*Keyword* | *Identifier* | *IntegerLiteral* | *Operator* | `,` | `;` | `:` | `:=` | `=` | `(` | `)` | `[` | `]` | *eot*
Keyword	→	`begin` | `const` | `do` | `else` | `end` | `fun` | `if` | `in` | `let` | `out` | `proc` | `then` | `var` | `while`
Identifier	→	*Letter* | *Identifier Letter* | *Identifier Digit* except *Keyword*
IntegerLiteral	→	*Digit* | *IntegerLiteral Digit*
Operator	→	`^` | `*` | `/` | `+` | `-` | `<` | `<=` | `==` | `!=` | `>=` | `>` | `&&` | `||` | `!`
Letter	→	`A` | `B` | ... | `Z` | `a` | `b` | ... | `z`
Digit	→	`0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`
Separator	→	*Comment* | *space* | *eol*
Comment	→	`//` (any character except *eol*)* *eol*

A.2 MiniTriangle Context-Free Syntax

Non-terminals are typeset in italics, like *this*. Terminals are typeset in typewriter font, like `this`. Terminals whose spelling (the concrete character sequence) is different from what is shown in the grammar are typeset in italics and underlined, like *Identifier* and *IntegerLiteral*. Their spelling is defined by the lexical grammar (where they are non-terminals!).

Program → *Command*

Commands → *Command*
| *Command* `;` *Commands*

Command → *VarExpression* `:=` *Expression*
| *VarExpression* `(` *Expressions* `)`
| `if` *Expression* `then` *Command*
`else` *Command*
| `while` *Expression* `do` *Command*
| `let` *Declarations* `in` *Command*
| `begin` *Commands* `end`

Expressions → ϵ
| $Expressions_1$

$Expressions_1$ → *Expression*
| *Expression* `,` $Expressions_1$

Expression → *PrimaryExpression*
| *Expression BinaryOperator Expression*

PrimaryExpression → *IntegerLiteral*
| *VarExpression*
| *UnaryOperator PrimaryExpression*
| *VarExpression* `(` *Expressions* `)`
| `[` *Expressions* `]`
| `(` *Expression* `)`

VarExpression → *Identifier*
| *VarExpression* `[` *Expression* `]`

BinaryOperator → `^` | `*` | `/` | `+` | `-` | `<` | `<=` | `==` | `!=` | `>=` | `>` | `&&` | `||`

UnaryOperator → `-` | `!`

Declarations → *Declaration*
| *Declaration* `;` *Declarations*

Declaration → `const` *Identifier* `:` *TypeDenoter* `=` *Expression*
| `var` *Identifier* `:` *TypeDenoter*
| `var` *Identifier* `:` *TypeDenoter* `:=` *Expression*
| `fun` *Identifier* `(` *ArgDecls* `)` `:` *TypeDenoter* `=` *Expression*
| `proc` *Identifier* `(` *ArgDecls* `)` *Command*

Declarations → ϵ
| $ArgDecls_1$

$ArgDecls_1$ → *ArgDecl*
| *ArgDecl* `,` $ArgDecls_1$

ArgDecl → *Identifier* `:` *TypeDenoter*
| `in` *Identifier* `:` *TypeDenoter*
| `out` *Identifier* `:` *TypeDenoter*
| `var` *Identifier* `:` *TypeDenoter*

TypeDenoter → *Identifier*
| *TypeDenoter* `[` *IntegerLiteral* `]`

Note that the productions for *Expression* makes the grammar as stated above ambiguous. Operator precedence and associativity for the *binary* operators as defined in the following table is used to disambiguate:

Operator	**Precedence**	**Associativity**
`^`	1	right
`* /`	2	left
`+ -`	3	left
`< <= == != >= >`	4	non
`&&`	5	left
`||`	6	left

A precedence level of 1 means the highest precedence, 2 means second highest, and so on.

A.3 MiniTriangle Abstract Syntax

This is the MiniTriangle abstract syntax. It captures the tree structure of MiniTriangle programs as concisely as possible. For example, note that there is only one non-terminal for expressions as opposed to three in the grammar for the concrete syntax. Such "extra" non-terminals are helpful for specifying the exact details of the concrete syntax, and sometimes to avoid ambiguity. But once a program has been successfully parsed, its structure has been determined, and such extra detail no longer serve any purpose. Another difference is that concrete unary and concrete (infix) binary operator application are subsumed by function application, as such operators *are* functions of one and two arguments, respectively. As a consequence, a single "variable" terminal *Name* replaces *Identifier* and *Operator*; i.e., *Name* = *Identifier* ∪ *Operator*.

The rightmost column gives the node labels for drawing abstract syntax trees. They are also used as the names of the data constructors of the datatypes for representing MiniTriangle programs in the compiler. Note that some elements of concrete syntax, such as keywords, do occur in the productions. They are there to make the connection between the concrete and abstract syntax clear, and to provide an *alternative* textual representation for the abstract syntax (e.g. for use in typing rules). However, these fragments of concrete syntax are *omitted* when drawing abstract syntax trees, as they are implied by the node labels and thus are superfluous. Also note that some of the productions make use of the EBNF *-notation for sequences. When drawing an abstract syntax tree, that means that the corresponding nodes will have a varying number of children.

Program	→	*Command*	Program
Command	→	*Expression* `:=` *Expression*	CmdAssign
	|	*Expression* `(` *Expression** `)`	CmdCall
	|	`begin` *Command** `end`	CmdSeq
	|	`if` *Expression* `then` *Command* `else` *Command*	CmdIf
	|	`while` *Expression* `do` *Command*	CmdWhile
	|	`let` *Declaration** `in` *Command*	CmdLet
Expression	→	*IntegerLiteral*	ExpLitInt
	|	*Name*	ExpVar
	|	*Expression* `(` *Expression** `)`	ExpApp
	|	`[` *Expression** `]`	ExpAry
	|	*Expression* `[` *Expression* `]`	ExpIx

Declaration	→	`const` *Name* `:` *TypeDenoter* `=` *Expression*	DeclConst
	|	`var` *Name* `:` *TypeDenoter* (`:=` *Expression* | ϵ)	DeclVar
	|	`fun` *Name* `(` *ArgDecl** `)` `:` *TypeDenoter* `=` *Expression*	DeclFun
	|	`proc` *Name* `(` *ArgDecl** `)` *Command*	DeclProc
ArgDecl	→	*ArgMode* *Name* `:` *TypeDenoter*	ArgDecl
ArgMode	→	ϵ	ByValue
	|	`in`	ByRefIn
	|	`out`	ByRefOut
	|	`var`	ByRefVar
TypeDenoter	→	*Name*	TDBaseType
	→	*TypeDenoter* `[` *IntegerLiteral* `]`	TDArray

B MiniTriangle Standard Environment

The MiniTriangle standard environment provides the following types, constants, and procedures:

Name	Type	Description
	Types	
`Boolean`	type	Boolean type; elements: `false`, `true`
`Integer`	type	Integer type; 32 bits
	Constants	
`false`	`Boolean`	The truth value **false**
`true`	`Boolean`	The truth value **true**
`minint`	`Integer`	The smallest represesentable integer
`maxint`	`Integer`	The largest representable integer
	Procedures	
`getint`	(`Snk Integer`) → `Void`	Read integer from the terminal
`putint`	`Integer` → `Void`	Write integer to the terminal
`skip`	() → `Void`	Do nothing

Additionally, the standard environment defines a number of functions and procedures used internally by the compiler, such as implementations of all operators and a procedure for reporting array indices out of bounds.

C MiniTriangle Type System

This appendix explains and specifies the type system for the HMTC version of MiniTriangle (henceforth just MiniTriangle). The notation essentially follows B.C. Pierce *Types and Programming Languages*, and the presentation is also inspired by that book. Naming conventions:

Symbol	Meaning/syntactic category
Γ	Type environment (or typing context)
S, T	*Type*
c	*Command*
e	*Expression*
d	*Declaration*
a	*ArgDecl*
x	Variable *Name*
p, f	Procedure/function *Name*
n	*IntegerLiteral*

In other words, we have $c \in \mathit{Command}$, $e \in \mathit{Expression}$, $x, p, f \in \mathit{Name}$, and so on. Subscripted and primed variants of these are also used with the same interpretation. For example, e_1, e_{X}, e' all stand for expressions; i.e., $e_1 \in \mathit{Expression}$, $e_{\mathrm{x}} \in \mathit{Expression}$, $e' \in \mathit{Expression}$.

The syntactic categories referred to above (*Command*, *Expression*, *Name*, etc.) are as specified by the MiniTriangle abstract syntax (appendix A.3), except for *Type* that is defined in the following. The syntactic category *TypeDenoter* in the abstract syntax corresponds to types that may occur as parts of declarations: at present the base types `Boolean` and `Integer` along with (nested) arrays of the base types. However, to specify the MiniTriangle type system we need a refined notion of type that can express the exact type of *any* typed MiniTriangle entity. *Type* is defined such such that $\mathit{TypeDenoter} \subseteq \mathit{Type}$. This simplifies the typing rules for declarations, but is not strictly speaking necessary.

The typing rules make use of vector notation for conciseness. For example, $\overline{T}$ is a sequence of zero or more types; i.e. $\overline{T} \in \mathit{Type}^*$. However, we will also use vector notation as a shorthand to avoid distracting repetition when the meaning is clear. For example, we take $\overline{e} : \overline{T}$ to be a shorthand for $\overline{e} = e_1, e_2, \ldots e_n$; $\overline{T} = T_1, T_2, \ldots T_n$; and $e_1 : T_1$, $e_2 : T_2$, $\ldots e_n : T_n$ for some $n \in \mathbb{N}$.

C.1 MiniTriangle Types

The syntax of types is defined by the following context-free grammar:

Type	→	*Types:*
	`Void`	*The empty type (procedure return type)*
	| `Boolean`	*The Boolean type*
	| `Integer`	*The Integer type*
	| `Src` *Type*	*Read-only variable reference (source)*
	| `Snk` *Type*	*Write-only variable reference (sink)*
	| `Ref` *Type*	*Variable reference*
	| $Type^* \rightarrow Type$	*Type of procedures and functions (arrow)*

The intended meaning of these types should be clear, except perhaps for reference types that will be explained later.

The set of types specified above is more general than what is needed for the present version of MiniTriangle. For example, the type syntax does not rule out long chains of reference types (such as `Ref` (`Ref` (`Ref Integer`))) or higher-order procedures and functions, neither of which is possible since the MiniTriangle grammar simply does not provide any way to express programs making use of such types (and nor are they necessarily supported by the later stages of the compiler or abstract machine). However, the above grammar is much simpler than a more precise account of the types that actually can occur, and it facilitates future generalisations.

C.2 Imperative Variables and Dereferencing

One feature common to most imperative languages is that dereferencing is implicit when variables are read. For example, consider the following C-like declarations:

```
int x;
int y;
```

and the code fragment:

```
y = x + 1;
```

The variables `x` and `y` are each really a *reference* to a memory location where an integer can be stored, and the code above actually says:

> *Fetch* the integer stored at the address `x` refers to. Add one to this integer. Store the result at the address `y` refers to.

However, note that whereas the addition (`+ 1`) and storing the result (`=`) are both operations that are explicitly mentioned in the code fragment, fetching is not: it is tacitly assumed that `x` stands for the *value* stored in the memory

location associated with `x`, *not* the address of this location. In contrast, `y`, on the left-hand side of the assignment, *does* stand for the address of the memory location associated with `y`.

Thus we see that it is the usage context of a variable occurrence that determines if this occurrence is to be understood as the address of the variable or the value stored there. Not needing to explicitly indicate when a value has to be fetched from memory is what is meant by implicit dereferencing.

C.3 MiniTriangle Reference Types

The fact that variables are references to memory locations is made explicit in the MiniTriangle type system. For example, consider a declaration of a variable `x` of type `Integer`:

```
let
    var x : Integer
in
    ...
```

The type attributed to `x` in the body of the `let`-construct becomes `Ref Integer`; that is, a reference to a memory location that can hold a value of type `Integer`, or "reference of type `Integer`" for short. References are always typed to make the type of the referenced value clear.

There are actually three kinds of references in MiniTriangle:

- `Ref` T: read/write reference; i.e., values of type T can be written to and read from the referenced memory location.
- `Src` T: *source*; i.e., read-only reference.
- `Snk` T: *sink*; i.e., write-only reference.

These three types are collectively referred to as "reference types" or just "references". We formalise this through the following predicate:

$$\text{reftype}(\texttt{Src}\ T) \quad (1)$$

$$\text{reftype}(\texttt{Snk}\ T) \quad (2)$$

$$\text{reftype}(\texttt{Ref}\ T) \quad (3)$$

Perhaps somewhat confusingly, we will also use "reference of type T" in the narrower sense of `Ref` T. Usually the context will make it clear what is meant; otherwise we will write "read/write reference" to distinguish from "source" and "sink".

Read/write references are used for variables, as illustrated by the example above, while sources are used for constants. For example, a definition:

```
const c : Integer = ...
```

results in the type `Src Integer` being attributed to `c`.

The three reference types are also used for passing arguments to procedures and functions by reference, allowing the specification of input/output, input, and output arguments. For example, the type of the procedure `getint` in the MiniTriangle standard library, which is used to input an integer from the terminal, is `Snk Integer` → `Void`, meaning that it needs to be passed a reference to a memory location to which the integer read from the terminal can be written.

However, as in most imperative languages, dereferencing is still implicit when values of reference type are used. The way this works is that the the MiniTriangle system is set up so that an entity of reference type *also* is considered to have the type of the referenced entity. To illustrate, when the integer variable `x` from the example above is used in an expression like `x + 1`, `x` has *both* type `Ref Integer` and `Integer`. As the type of the operator `+` is (`Integer`, `Integer`) → `Integer`, we can see that the first argument has to have type `Integer`. But as that is *one* of the possible types for `x`, the expression `x + 1` is well-typed in this case, with the overall type of the expression being `Integer`.

Behind the scenes, the type checker, in addition to checking that a program is well-formed in the sense defined by the MiniTriangle type system, inserts dereferencing operations to ensure that an expression of reference type actually gets the type it needs to have to fit with the usage context. To continue the example, the type checker will transform `x + 1` into something like `deref(x) + 1` in this case[1].

Should there be multiple levels of referencing, the type checker would insert two or more dereferencing operations as needed. For example, assume the type of `x` had been `Src` (`Ref Integer`) instead. Then `x + 1` would be transformed to `deref(deref(x)) + 1`. Ultimately, each `deref`-operation is translated into an instruction that reads the memory contents at the referenced location (which, in the case of nested referencing, will be another reference that may be further dereferenced in turn).

[1]Note that `deref` is a "hidden" language construct, only used internally by the compiler at present, not a function that can be used by the programmer.

C.4 Subtyping

One type S is said to be a subtype of another type T, written $S <: T$, if a value of type S can be used wherever a value of type T is expected. Object-oriented languages, like Java and C#, are important examples of languages with type systems based on subtyping. Recall that an object that is an instance of a class C can be used wherever instances of any of C's superclasses are expected. I.e. C is a subtype of C's superclasses.

There are other possibilities as well. For example, in a language that has a type `Nat` for natural numbers, `Nat` might be considered to be a subtype of an integer type `Int`, as any natural number is also an integer. In this case, it is clear that there is a close connection between subsets and subtypes. That is, at least at a conceptual level: internally, it is not necessarily the case that elements of a subtype (in this case `Nat`) have the same representation as the corresponding elements of the supertype (in this case the elements of `Int` that are natural numbers).

However, if it is possible to make representations coincide in an implementation, this is advantageous as it makes the implementation of subtyping both easier and more efficient as there is no need to convert between representations at runtime. Object-oriented languages are thus typically designed to make this possible. For example, in Java, there is no runtime overhead associated with viewing an instance of some class C as an instance of a superclass of C[2].

C.5 Subtyping in MiniTriangle

In MiniTriangle, the reference types naturally induces a subtyping relation as a read/write reference can be used in place of either a source (read-only) or a sink (write-only). The following inference rules define the MiniTriangle subtyping relation:

[2] *Downcasting* is a different matter. As there is no static guarantee that an entity that statically is known to have some type T at runtime actually has some more refined type S as dictated by the cast, with $S <: T$, a runtime check that ensures that the entity actually has the more refined type needs to be inserted to guarantee language safety.

$$T <: T \qquad (1)$$

$$\frac{S <: T}{\texttt{Src}\ S <: \texttt{Src}\ T} \qquad (2)$$

$$\frac{T <: S}{\texttt{Snk}\ S <: \texttt{Snk}\ T} \qquad (3)$$

$$\frac{S <: T \quad T <: S}{\texttt{Ref}\ S <: \texttt{Ref}\ T} \qquad (4)$$

$$\frac{S <: T}{\texttt{Ref}\ S <: \texttt{Src}\ T} \qquad (5)$$

$$\frac{T <: S}{\texttt{Ref}\ S <: \texttt{Snk}\ T} \qquad (6)$$

$$\frac{\overline{T} <: \overline{S} \quad S' <: T'}{\overline{S} \to S' <: \overline{T} \to T'} \qquad (7)$$

Rule (1) says that any type is a subtype of itself; i.e., subtyping is *reflexive*. In fact, subtyping is also *transitive*, which is essential, making it a *preorder*. However, transitivity is not manifest from these rules but has to be proved (by induction).

If we have a value of type `Src` S, then it can be dereferenced to obtain a value of type S. Clearly, if $S <: T$, then this value of type S can be used wherever a value of type T is expected. But then it follows that we can use a value of type `Src` S wherever a value of type `Src` T is expected; i.e., it should be the case that `Src` $S <:$ `Src` T. Rule (2) formalises this. Note that `Src` is *covariant*: it preserves the subtyping ordering.

By a similar line of reasoning, we expect a subtyping relationship to hold between `Snk` S and `Snk` T if S and T are related by subtyping. Rule (3) captures this. Note that this time, the requirement is that $T <: S$; i.e., `Snk` is a *contravariant* type constructor (reverses the subtyping ordering). Why? Well, if we are using a sink of type S at type `Snk` T, then any value of type T written into this sink could potentially be used at type S through some different route. But that will only work if we insist that T is a subtype of S.

Rule (4) is essentially a combination of rule (2) and (3) as a reference simultaneously is a source and a sink.

Rules (5) and (6) formalises that a reference is both a source and sink and thus can be used in place of either. Note that rule (5) is covariant while rule (6) is contravariant for the same reasons as above.

Rule (7), finally, formalises when a procedure or function type is a subtype of another. Note the covariant subtyping relationship between the result types and the contravariant one between the arguments. Intuitively, this is because returning a value is akin to a source, while passing arguments is akin to writing them to a sink.

All reference types in MiniTriangle (`Src` T, `Snk` T, `Ref` T) share a common representation: they are just pointers. The differentiation between sources, sinks, and read/write references only serve to keep track of read and write

"permissions". No change of representation is needed to use a value of some type T at one of T's supertypes, and thus no runtime overhead is incurred.

C.6 Implicit Dereferencing in MiniTriangle

As discussed in Appendix C.3, dereferencing is implicit in MiniTriangle. At the type level, this is manifested through an implicit coercion from a reference type to the type of the referenced entity, meaning that the type system ascribes more than one type to such entities. For example, an expression of type `Ref Integer` is implicitly coerced to `Integer` when necessary, meaning it can be typed at both types.

However, unlike subtyping, this is not just a matter of viewing an entity as having a more refined type, but these coercions actually involve a representational change: from a reference (pointer) to an entity to the referenced entity itself by following the reference. The MiniTriangle type checker therefore has to insert *explicit* dereferencing operations wherever the type system makes use of an implicit coercion from a reference type to the referenced type.

The predicates sources and sinks defined below account for this. If it is the case that sources(S, T) holds, this means that a value of type S can "source" a value of type T through zero or more dereferencing operations. For example, from a value of type `Ref` (`Src` (`Snk Boolean`)) we can obtain a value of type `Snk Boolean` by dereferencing twice. If sinks(S, T) holds, it means that it is possible to obtain a sink to which a value of type T can be written from a value of type S through zero or more dereferencing operations. For example, from a value of type `Ref` (`Snk Integer`) it is possible to obtain an integer sink by dereferening once.

$$\frac{S <: T}{\text{sources}(S,\ T)} \quad (1) \qquad\qquad \frac{T <: S}{\text{sinks}(\texttt{Snk}\ S,\ T)} \quad (1)$$

$$\frac{\text{sources}(S,\ T)}{\text{sources}(\texttt{Src}\ S,\ T)} \quad (2) \qquad\qquad \frac{T <: S}{\text{sinks}(\texttt{Ref}\ S,\ T)} \quad (2)$$

$$\frac{\text{sources}(S,\ T)}{\text{sources}(\texttt{Ref}\ S, T)} \quad (3) \qquad\qquad \frac{\text{sinks}(S,\ T)}{\text{sinks}(\texttt{Src}\ S,\ T)} \quad (3)$$

$$\frac{\text{sinks}(S,\ T)}{\text{sinks}(\texttt{Ref}\ S,\ T)} \quad (4)$$

Rules (2) and (3) of the definition of the predicate sources and rules (3) and (4) of the definition of the predicate sinks are justified by dereferencing. Thus, when the type checker carries out a typing derivation, it will have to

insert a dereferencing operation exactly once each time one of these rules are used[3]. By contrast, rule (1) of the definition of the predicate sources and rules (1) and (2) of the definition of the predicate sinks are justified by subtyping alone and no dereferencing operations are thus inserted in those cases. For example, if $S <: T$, then it is clearly possible to obtain a value of type T from a value of type S as the latter also has type T by virtue of subtyping.

C.7 MiniTriangle Typing Relations

We are now in a position to specify the MiniTriangle type system as such. This is done through the following typing relations:

$\Gamma \vdash c$	Command c is well-formed in type environment Γ
$\Gamma \vdash e : T$	Expression e has type T in type environment Γ
$\Gamma\ ;\ \Gamma_{\mathrm{B}} \vdash \overline{d} \mid \Gamma'$	Declarations $\overline{d}$ are well-formed in type environments Γ and Γ_{B}, extending the environment Γ to Γ'
$\Gamma \vdash \overline{a} \mid \Gamma'$	Argument declarations $\overline{a}$ are well-formed in type environment Γ, extending the environment to Γ'

A type environment associates names with types, allowing the type of a named entity to be found (if it is in scope). However, it is also necessary to keep track of the current scope level and the scope level at which a named entity has been declared. An environment is therefore taken to be a pair of scope level and a list of pairs of the form

$$x_{(n)} : T$$

This can be read "x at scope level n has type T". Additionally, for an environment to be well-formed, the scope levels of the names must not exceed the scope level of the environment. Thus, an environment Γ has the form:

$$\Gamma = (n;\ \langle x_{1(n_1)} : T_1,\ x_{2(n_2)} : T_2,\ \ldots\ x_{k(n_k)} : T_k \rangle)$$

where $n_i \leq n$ for $1 \leq i \leq k$.

To keep the typing rules clear and concise, we adopt some notation, conventions, and abbreviations in relation to environments.

- $\Gamma_{(n)}$: A subscript within brackets is used to refer to the scope level of an environment. If two environments with the same name but different scope levels occur in a rule, e.g. $\Gamma_{(n)}$ and $\Gamma_{(n+1)}$, it is understood that the environments are the same, except for the scope level.

[3]This correspondence is particularly clear in the definition of the function `sources` in the type checker. The definition is not much more than a transliteration of the rules defining the predicate sources, with the addition of code for inserting the dereferencing operations.

- $\Gamma(x)$: The type of x in Γ; if there are more than one entity named x in scope, then the type of one with the highest scope level.
- $x \in \Gamma$: True if $x_{(n)} : T$ for some n and T is in Γ.
- $x_{(n)} \in \Gamma$: True if $x_{(n)} : T$ for some T is in Γ.
- $\Gamma,\ x : T$: Given

  $$\Gamma = (n;\ \langle x_{1(n_1)} : T_1,\ x_{2(n_2)} : T_2,\ \ldots\ x_{k(n_k)} : T_k \rangle)$$

  $\Gamma,\ x : T$ is shorthand for the extended environment

  $$(n;\ \langle x_{1(n_1)} : T_1,\ x_{2(n_2)} : T_2,\ \ldots\ x_{k(n_k)} : T_k,\ x_{(n)} : T \rangle)$$

  and an additional side condition:

  $$x_{(n)} \notin \Gamma$$

  In other words, the environment Γ is extended with the information that an entity named x at the present scope level n has type T, but only if there is not already a declaration for x at this scope level.

C.7.1 Commands

The following inference rules define the typing relation specifying well-formed commands:

$$\frac{\Gamma \vdash e_{\mathrm{x}} : S \quad \Gamma \vdash e : T \quad \neg\mathrm{reftype}(T) \quad \mathrm{sinks}(S,\ T)}{\Gamma \vdash e_{\mathrm{x}}\ \texttt{:=}\ e} \qquad \text{(T-ASSIGN)}$$

$$\frac{\Gamma \vdash e_{\mathrm{p}} : \overline{T} \to \texttt{Void} \quad \Gamma \vdash \overline{e} : \overline{T}}{\Gamma \vdash e_{\mathrm{p}}\,\texttt{(}\overline{e}\texttt{)}} \qquad \text{(T-CALL)}$$

$$\frac{\Gamma \vdash \overline{c}}{\Gamma \vdash \texttt{begin}\ \overline{c}\ \texttt{end}} \qquad \text{(T-SEQ)}$$

$$\frac{\Gamma \vdash e : \texttt{Boolean} \quad \Gamma \vdash c_1 \quad \Gamma \vdash c_2}{\Gamma \vdash \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2} \qquad \text{(T-IF)}$$

$$\frac{\Gamma \vdash e : \texttt{Boolean} \quad \Gamma \vdash c}{\Gamma \vdash \texttt{while}\ e\ \texttt{do}\ c} \qquad \text{(T-WHILE)}$$

$$\frac{\Gamma_{(n+1)}\ ;\ \Gamma' \vdash \overline{d} \mid \Gamma' \quad \Gamma' \vdash c}{\Gamma_{(n)} \vdash \texttt{let}\ \overline{d}\ \texttt{in}\ c} \qquad \text{(T-LET)}$$

Most of the rules above are straightforward and fairly standard. However, T-ASSIGN and T-LET deserve some comments. The rule T-ASSIGN says that the expression e_{x} denoting the variable that will be written to must have a type S such that S can sink (possibly after implicit dereferencing) a value of type T, where T is the type of the expression yielding the value to be written. This should be fairly intuitive. The reason that the rule further insists that the type T must not be a reference type is to avoid ambiguity. Because an expression can have more than one type due to implicit dereferencing (and subtyping; see Appendix C.6), it could be that it would not be clear just how many dereferencing steps should be carried out otherwise.

For example, suppose e_{x} : `Ref (Snk Integer)` and e : `Ref Integer`. Note that sinks(`Ref (Snk Integer)`, `Ref Integer`) holds because `Ref (Integer)` <: `Snk (Integer)` (rule (2) in the definition of sinks). This typing corresponds to storing a reference (pointer) to an integer into the location referred to by the value of e_{x}.

However, there is another possibility. Note that e_{x} can source a integer sink (one dereferening operation). Thus we also have e_{x} : `Snk Integer`. Similarly, e can source an integer (one dereferencing operation). Thus we also have e : `Integer`. And clearly, sinks(`Snk Integer`, `Integer`) holds. This typing corresponds to fetching the integer referred to be e and storing in the location *indirectly* referred to be e_{x}. This is a very different semantics from above. By insisting that the assigned values are as "dereferenced as possible", we avoid this ambiguity (at the expense of losing flexibility not needed in the present version of MiniTriangle anyway), opting for the latter interpretation.

Regarding T-LET, note how the scope level is increased by one before extending the environment according to the declarations. Also note that Γ', the extended environment, is not only used for checking that the body of the `let`-command is well-formed, but also when checking that the declarations themselves are well-formed. The latter allows for recursive procedures and functions. See the rules for declarations below.

C.7.2 Expressions

The typing rules for expressions are as follows:

$$\frac{\Gamma \vdash e : S \quad \text{sources}(S,\ T)}{\Gamma \vdash e : T} \qquad \text{(T-SOURCES)}$$

$$\Gamma \vdash n : \texttt{Integer} \qquad \text{(T-LITINT)}$$

$$\frac{\Gamma(x) = T}{\Gamma \vdash x : T} \qquad \text{(T-VAR)}$$

$$\frac{\Gamma \vdash e_\text{f} : \overline{T} \to T' \quad \Gamma \vdash \overline{e} : \overline{T}}{\Gamma \vdash e_\text{f}\,(\overline{e}) : T'} \qquad \text{(T-APP)}$$

$$\frac{\Gamma \vdash \overline{e} : T \quad n = \text{length}(\overline{e})}{\Gamma \vdash \texttt{[}\overline{e}\texttt{]} : T\,\texttt{[}n\texttt{]}} \qquad \text{(T-ARY)}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_\text{a} : R\,(T\,\texttt{[}n\texttt{]}) \quad \Gamma \vdash e_\text{i} : \texttt{Integer} \\ R \in \{\texttt{Src}, \texttt{Snk}, \texttt{Ref}\}\end{array}}{\Gamma \vdash e_\text{a}\,\texttt{[}e_\text{i}\texttt{]} : R\,T} \qquad \text{(T-IX)}$$

The rules T-LITINT, T-VAR, T-APP are standard. For T-ARY, we take the common type T of the array elements to be arbitrary if $n = 0$. T-IX specifies that the array expression must denote a *reference* to an array, and that the result of indexing this array is a reference (of the same kind) to an individual element of that array. (This ensures arrays are accessed in place, rather then needlessly being copied to the stack first.) Finally, T-SOURCES is the rule that allows implicit dereferencing. It says that any expression that has type S in the environment Γ also has type T in the same environment if S can source T; i.e., if a value of type S can be dereferenced zero or more times to obtain a value of type T. See the definition of sources in Section C.6, and note how the sources predicate also takes care of subtyping thanks to rule (1) of its definition.

C.7.3 Declarations

Finally we turn to the typing rules for declarations. The relation to determine if a list of declarations is well-typed is defined inductively by simply recursing down this list, extending the environment in which the declarations are checked along the way, thus ensuring that each declared entity is brought into scope in the following declarations. Once the end of the list is reached, the final environment is returned as the one in which to check the body of a `let`-command; see rule T-LET above. However, to allow for (mutually) recursive functions and procedures, the bodies of these are checked in a separate "body" environment, Γ_B. The rule T-LET is set up so that Γ_B is the

same as the environment in which the body of the `let`-command is checked, meaning that *all* declared entities are in scope in the bodies of all functions and procedures.

$$\frac{\Gamma \vdash e : T \quad \Gamma,\, x : \texttt{Src}\ T \,;\, \Gamma_{\mathrm{B}} \vdash \overline{d} \mid \Gamma' \quad \Gamma \vdash \mathrm{wellinit}(n,\, e)}{\Gamma_{(n)} \,;\, \Gamma_{\mathrm{B}} \vdash \texttt{const}\ x\ \texttt{:}\ T\ \texttt{=}\ e\ \texttt{;}\ \overline{d} \mid \Gamma'} \qquad \text{(T-DECLCONST)}$$

$$\frac{\Gamma,\, x : \texttt{Ref}\ T \,;\, \Gamma_{\mathrm{B}} \vdash \overline{d} \mid \Gamma'}{\Gamma \,;\, \Gamma_{\mathrm{B}} \vdash \texttt{var}\ x\ \texttt{:}\ T\ \texttt{;}\ \overline{d} \mid \Gamma'} \qquad \text{(T-DECLVAR)}$$

$$\frac{\Gamma \vdash e : T \quad \Gamma,\, x : \texttt{Ref}\ T \,;\, \Gamma_{\mathrm{B}} \vdash \overline{d} \mid \Gamma' \quad \Gamma \vdash \mathrm{wellinit}(n,\, e)}{\Gamma_{(n)} \,;\, \Gamma_{\mathrm{B}} \vdash \texttt{const}\ x\ \texttt{:}\ T\ \texttt{:=}\ e\ \texttt{;}\ \overline{d} \mid \Gamma'} \qquad \text{(T-DECLINITVAR)}$$

$$\frac{\Gamma_{\mathrm{B(n+1)}} \vdash \overline{a} \mid \Gamma'_{\mathrm{B}} \quad \Gamma'_{\mathrm{B}} \vdash c \quad \Gamma,\, f : \mathrm{proctype}(\overline{a}) \,;\, \Gamma_{\mathrm{B}} \vdash \overline{d} \mid \Gamma'}{\Gamma \,;\, \Gamma_{\mathrm{B(n)}} \vdash \texttt{proc}\ p\texttt{(}\overline{a}\texttt{)}\ c\ \texttt{;}\ \overline{d} \mid \Gamma'} \qquad \text{(T-DECLPROC)}$$

$$\frac{\Gamma_{\mathrm{B(n+1)}} \vdash \overline{a} \mid \Gamma'_{\mathrm{B}} \quad \Gamma'_{\mathrm{B}} \vdash e : T \quad \Gamma,\, f : \mathrm{funtype}(\overline{a},\, T) \,;\, \Gamma_{\mathrm{B}} \vdash \overline{d} \mid \Gamma'}{\Gamma \,;\, \Gamma_{\mathrm{B(n)}} \vdash \texttt{fun}\ f\texttt{(}\overline{a}\texttt{)}\ \texttt{:}\ T\ \texttt{=}\ e\ \texttt{;}\ \overline{d} \mid \Gamma'} \qquad \text{(T-DECLFUN)}$$

$$\Gamma \,;\, \Gamma_{\mathrm{B}} \vdash \epsilon \mid \Gamma \qquad \text{(T-DECLEMPTY)}$$

There are five main cases: constant definition, uninitialised variable declaration, initialised variable declaration, procedure declaration, and function declaration. Note how the declared type of the constant or variable is checked against the defining or initialisation expression in case there is one. Further, note how the type of the declared entity is added to the environment as `Src` T in the case of a constant and `Ref` T in the case of a variable. Thus in all cases, the type of the declared entity is a reference type, making it clear that a dereferencing operation is needed to get the actual value. This is because both constants and variables are stored on the stack, with only their addresses (offsets with respect to the stack base or local base), not their values, known at compile time. The type of a constant is `Src` T as constants can only be read (after their initial definition), while the type of variables is `Ref` T as they can be both read and written. Finally, note how Γ_B, rather than Γ, as discussed above, is extended by the declarations of formal arguments to Γ'_B, and then used for checking the well-formedness of the bodies of declared procedures and functions.

The types of declared procedures and functions are computed by the following auxiliary functions:

$$\begin{aligned}
\text{proctype}(\overline{a}) &= \text{argtypes}(\overline{a}) \to \texttt{Void} \\
\text{funtype}(\overline{a},\ T) &= \text{argtypes}(\overline{a}) \to T \\
\\
\text{argtypes}(x : T,\ \overline{a}) &= T,\ \text{argtypes}(\overline{a}) \\
\text{argtypes}(\texttt{in}\ x : T,\ \overline{a}) &= \texttt{Src}\ T,\ \text{argtypes}(\overline{a}) \\
\text{argtypes}(\texttt{out}\ x : T,\ \overline{a}) &= \texttt{Snk}\ T,\ \text{argtypes}(\overline{a}) \\
\text{argtypes}(\texttt{var}\ x : T,\ \overline{a}) &= \texttt{Ref}\ T,\ \text{argtypes}(\overline{a}) \\
\text{argtypes}(\epsilon) &= \epsilon
\end{aligned}$$

The defining expressions for constants and the initialisation expressions for variables must be "well-initialised", meaning that they must not use functions defined in the same `let`-block. This make a straightforward implementation of constant and variable allocation and initialisation possible. (The type system as such would work fine without it.) The relation $\Gamma \vdash \text{wellinit}(n,\ e)$ formalises this requirement:

$$\Gamma \vdash \text{wellinit}(\_,\ n) \qquad \text{(WI-LITINT)}$$

$$\Gamma \vdash \text{wellinit}(\_,\ x) \qquad \text{(WI-VAR)}$$

$$\frac{f_{(n)} \notin \Gamma \quad \Gamma \vdash \text{wellinit}(n,\ \overline{e})}{\Gamma \vdash \text{wellinit}(n,\ f\,\texttt{(}\overline{e}\texttt{)})} \qquad \text{(WI-APP)}$$

$$\frac{\Gamma \vdash \text{wellinit}(n,\ \overline{e})}{\Gamma \vdash \text{wellinit}(n,\ \texttt{[}\overline{e}\texttt{]})} \qquad \text{(WI-ARY)}$$

$$\frac{\Gamma \vdash \text{wellinit}(n,\ e_\text{a}) \quad \Gamma \vdash \text{wellinit}(n,\ e_\text{i})}{\Gamma \vdash \text{wellinit}(n,\ e_\text{a}\,\texttt{[}e_\text{i}\texttt{]})} \qquad \text{(WI-IX)}$$

Note: if an unknown function is being called (although not currently possible), then that would conservatively not be considered well-initialised.

Finally, the relation for checking argument declarations and extending the given environment has a similar structure to that for checking declarations. Note how all formal arguments are treated as constants (sources).

$$\frac{\Gamma,\, x : \texttt{Src}\ T \ \vdash\ \overline{d} \mid \Gamma'}{\Gamma \ \vdash\ x : T \ ;\ \overline{d} \mid \Gamma'} \qquad \text{(T-DECLARG)}$$

$$\frac{\Gamma,\, x : \texttt{Src}\ (\texttt{Src}\ T) \ \vdash\ \overline{d} \mid \Gamma'}{\Gamma \ \vdash\ \texttt{in}\ x : T \ ;\ \overline{d} \mid \Gamma'} \qquad \text{(T-DECLINARG)}$$

$$\frac{\Gamma,\, x : \texttt{Src}\ (\texttt{Snk}\ T) \ \vdash\ \overline{d} \mid \Gamma'}{\Gamma \ \vdash\ \texttt{out}\ x : T \ ;\ \overline{d} \mid \Gamma'} \qquad \text{(T-DECLOUTARG)}$$

$$\frac{\Gamma,\, x : \texttt{Src}\ (\texttt{Ref}\ T) \ \vdash\ \overline{d} \mid \Gamma'}{\Gamma \ \vdash\ \texttt{var}\ x : T \ ;\ \overline{d} \mid \Gamma'} \qquad \text{(T-DECLVARARG)}$$

$$\Gamma \ \vdash\ \epsilon \mid \Gamma \qquad \text{(T-DECLARGEMPTY)}$$

C.8 Notes on MiniTriangle Type Checker Implementation

As far as languages and type systems go, MiniTriangle is a small language and its type system is fairly simple. Yet, turning the specification of its type system into a type checker is not entirely straightforward. The main issue is that the typing rules are not algorithmic; i.e., they can't directly be read as specifying a function, where the inputs uniquely determine the output, but just a relation that can be one to many. As a case in point, consider the rule T-SOURCES that allows a single expression to have many possible types. This was deliberate: the typing rules have been written for clarity and ease of understanding, keeping details about exactly *how* to go about checking types from obscuring the specification of the type system.

There are a number of approaches for handling this. A principled one would be to allow for many possible types, which then gradually would be narrowed down to, hopefully, only a single type once all constraints have been taken into account.

The MiniTriangle type system has, however, been designed to permit a simpler approach. By carefully considering what is known and unknown in each rule, each rule can be turned into one, or, in case what is knowns and not differ in different contexts, more functions, with the knowns as inputs and the unknowns as outputs, uniquely determined by the inputs. As a case in point, the relation $\Gamma \ \vdash\ e : T$ is implemented as a number of functions, including `chkTpExp` to be used to check that an expression e has a known type T in a known environment Γ (environments are always known), and `infTpExp` that infers the "least dereferenced" type for e, if well-typed, in the known environment Γ.

The notion of scope level is refined into major and minor scope levels in the implementation. The major scope level is incremented for bodies of procedures and functions. This is because a new activation record is created when a function or procedure is called. The difference in major scope level between the context in which a variable reference occurs and the scope level of the referenced variable determines how many static links that have to be followed to get to the correct activation record (see the lecture notes). In contrast, variables declared by a `let`-command gets allocated in the current activation record. Therefore only the minor scope levels needs incrementing.

D Triangle Abstract Machine Instructions

Meta variable	Meaning
a	Address: one of the forms specified by table below when part of an instruction, specific stack address when on the stack
b	Boolean value (false = 0 or true = 1)
ca	Code address; address to routine in the code seqment
d	Displacement; i.e., offset w.r.t. address in register or on the stack
l	Label name
m, n	Integer
x, y	Any kind of stack data
x^n	Vector of n items, in this case any kind

Address form	Description
`[SB +` d`]` `[SB -` d`]`	Address given by contents of register `SB` (Stack Base) +/− displacement d
`[LB +` d`]` `[LB -` d`]`	Address given by contents of register `LB` (Local Base) +/− displacement d
`[ST +` d`]` `[ST -` d`]`	Address given by contents of register `ST` (Stack Top) +/− displacement d

Instruction	Stack effect	Description
Label		
`LABEL` l	—	Pseudo instruction: symbolic location
Load and store		
`LOADL` n	$\ldots \Rightarrow n, \ldots$	Push literal integer n onto stack
`LOADCA` l	$\ldots \Rightarrow \text{addr}(l), \ldots$	Push address of label l (code segment) onto stack
`LOAD` a	$\ldots \Rightarrow [a], \ldots$	Push contents at address a onto stack
`LOADA` a	$\ldots \Rightarrow a, \ldots$	Push address a onto stack
`LOADI` d	$a, \ldots \Rightarrow [a+d], \ldots$	Load indirectly; push contents at address $a+d$ onto stack
`STORE` a	$n, \ldots \Rightarrow \ldots$	Pop value n from stack and store at address a
`STOREI` d	$a, n, \ldots \Rightarrow \ldots$	Store indirectly; store n at address $a+d$

Instruction	Stack effect	Description
	Block operations	
`LOADLB` m n	$\ldots \Rightarrow m^n, \ldots$	Push block of n literal integers m onto stack
`LOADIB` n	$a, \ldots \Rightarrow [a+(n-1)], \ldots, [a+0], \ldots$	Load block of size n indirectly
`STOREIB` n	$a, x^n, \ldots \Rightarrow \ldots$	Store block of size n indirectly
`POP` m n	$x^m, y^n, \ldots \Rightarrow x^m, \ldots$	Pop n values below top m values
	Arithmetic operations	
`ADD`	$n_2, n_1, \ldots \Rightarrow n_1 + n_2, \ldots$	Add n_1 and n_2, replacing n_1 and n_2 with the sum
`SUB`	$n_2, n_1, \ldots \Rightarrow n_1 - n_2, \ldots$	Subtract n_2 from n_1, replacing n_1 and n_2 with the difference
`MUL`	$n_2, n_1, \ldots \Rightarrow n_1 \cdot n_2, \ldots$	Multiply n_1 by n_2, replacing n_1 and n_2 with the product
`DIV`	$n_2, n_1, \ldots \Rightarrow n_1/n_2, \ldots$	Divide n_1 by n_2, replacing n_1 and n_2 with the (integer) quotient
`NEG`	$n, \ldots \Rightarrow -n, \ldots$	Negate n, replacing n with the result
	Comparison & logical operations (false $= 0$, true $= 1$)	
`LSS`	$n_2, n_1, \ldots \Rightarrow n_1 < n_2, \ldots$	Check if n_1 is smaller than n_2, replacing n_1 and n_2 with the Boolean result
`EQL`	$n_2, n_1, \ldots \Rightarrow n_1 = n_2, \ldots$	Check if n_1 is equal to n_2, replacing n_1 and n_2 with the Boolean result
`GTR`	$n_2, n_1, \ldots \Rightarrow n_1 > n_2, \ldots$	Check if n_1 is greater than n_2, replacing n_1 and n_2 with the Boolean result
`AND`	$b_2, b_1, \ldots \Rightarrow b_1 \wedge b_2, \ldots$	Logical conjunction of b_1 and b_2, replacing b_1 and b_2 with the Boolean result
`OR`	$b_2, b_1, \ldots \Rightarrow b_1 \vee b_2, \ldots$	Logical disjunction of b_1 and b_2, replacing b_1 and b_2 with the Boolean result
`NOT`	$b, \ldots \Rightarrow \neg b, \ldots$	Logical negation of b, replacing b with the result

Instruction	Stack effect	Description
	Control transfer	
`JUMP` l	—	Jump unconditionally to location identified by label l
`JUMPIFZ` l	$n, \ldots \Rightarrow \ldots$	Jump to location identified by label l if $n = 0$ (i.e., n is false)
`JUMPIFNZ` l	$n, \ldots \Rightarrow \ldots$	Jump to location identified by label l if $n \neq 0$ (i.e., n is true)
`CALL` l	$\ldots \Rightarrow pc+1, lb, 0, \ldots$	Call global subroutine at location identified by location l, setting up activation record by pushing static link (0 for global level), dynamic link (value of `LB`), and return address (`PC`+1, address of instruction after the call instruction) onto the stack
`CALLI`	$ca, sl, \ldots \Rightarrow$ $pc+1, lb, sl, \ldots$	Call subroutine indirectly; address of routine (ca) and static link to use (sl) on top of the stack; activation record as for `CALL`
`RETURN` m n	$x^m, pc, lb, 0, y^n \ldots$ $\Rightarrow x^m, \ldots$	Return from subroutine, replacing activation record by result and restoring `LB`
	Input/Output	
`PUTINT`	$n, \ldots \Rightarrow \ldots$	Print n to the terminal as a decimal integer
`PUTCHR`	$n, \ldots \Rightarrow \ldots$	Print the character with character code n to the terminal
`GETINT`	$\ldots \Rightarrow n, \ldots$	Read decimal integer n from the terminal and push onto the stack
`GETCHR`	$\ldots \Rightarrow n, \ldots$	Read character from the terminal and push its character code n onto the stack
	TAM Control	
`HALT`	—	Stop execution and halt the machine