

Implementation of Programming Languages

IMPL — COM00013I

Practical exercises

Jeremy Jacob^{*}

Version 2

^{*} Jeremy.Jacob@york.ac.uk

Contents

| | |
|---|-----------|
| 1. Introduction | 7 |
| 1.1. Structure of the practical sessions | 7 |
| 1.2. Structure of a compiler | 7 |
| 2. Lexical scanning and parsing | 8 |
| 2.1. Introduction | 8 |
| 2.2. Generating lexical scanners from regular expressions | 8 |
| 2.3. Generating parsers from grammars | 12 |
| 3. Syntax-directed programming; type-checking | 19 |
| 3.1. Syntax-directed programming | 19 |
| 3.2. Type checking | 23 |
| 3.3. Pre-compilation optimisation | 29 |
| 4. Interpreters: operational semantics | 30 |
| 5. Catch-up week | 33 |
| 6. Code generation for an unlimited register machine | 34 |
| 7. Optimisation | 36 |
| 8. Register allocation | 37 |
| 9. Catch-up week | 38 |
| 10. Extension: Blocks and procedures | 39 |
| 11. Extension: Lexer combinators | 40 |
| 12. Extension: Compiling for a graphical language | 44 |
| 12.1. Introduction | 44 |
| 12.2. A special purpose language | 44 |
| 12.2.1. The high-level language | 44 |

Contents

| | |
|--|-----------|
| 12.2.2. A low-level language | 47 |
| 12.3. A compiler for a special purpose language | 49 |
| 12.3.1. Preliminaries | 49 |
| 12.3.2. Compilation | 50 |
| 12.4. Optimising a compiler for a special purpose language | 51 |
| 12.4.1. Set up | 51 |
| 12.4.2. String rewriting | 52 |
| 12.4.3. Conditional rewriting | 53 |
| 12.4.4. An optimising compiler | 54 |
| A. The language M | 56 |
| A.1. Introduction | 56 |
| A.2. Syntax | 56 |
| A.2.1. Lexical categories | 56 |
| A.2.2. Syntactic categories other than instructions | 56 |
| A.2.3. The syntactic category of instructions | 57 |
| A.3. Static semantics | 58 |
| A.4. Dynamic semantics | 58 |
| B. The language N | 61 |
| B.1. Introduction | 61 |
| B.2. Syntax | 61 |
| B.3. Static semantics | 62 |
| B.3.1. Miscellaneous | 62 |
| B.3.2. Typing | 63 |
| B.3.3. Expression simplification | 65 |
| B.4. Dynamic semantics | 66 |
| B.4.1. Notation | 66 |
| B.4.2. Programs | 68 |
| B.4.3. Statements | 68 |
| C. Rules for compiling N to M | 70 |
| C.1. Rules for an unbounded register machine | 70 |
| C.1.1. Assumptions | 70 |
| C.1.2. Rules | 70 |
| D. Optimisation | 77 |
| D.1. Constant propagation | 77 |
| D.1.1. Purpose | 77 |
| D.1.2. Algorithm | 77 |

Contents

| | |
|--------------------------------------|----|
| D.2. Dead-code elimination | 78 |
| D.2.1. Purpose | 78 |
| D.2.2. Algorithm | 78 |

List of Figures

| | |
|---|----|
| 2.1. An example of lexing and parsing | 9 |
| 2.2. Disambiguation of grammars | 15 |
| 3.1. Proof rules for conjunction | 24 |
| 3.2. A proof of commutativity of conjunction | 24 |
| 3.3. Syntax of a small expression language | 26 |
| 3.4. Typing rules for a small language of expressions | 27 |
| 3.5. A proof in the small type system. To fit the proof on the page, it has been split into a lemma $(a; b, c \vdash b < c \implies \perp : B)$ and the main proof. | 27 |

List of Tables

| | |
|---|----|
| 2.1. A grammar for integer expressions | 13 |
| 2.2. A grammar of Lispkit Lisp | 14 |
| 2.3. A grammar of a trivial language | 16 |
| 2.4. A grammar of a trivial typed language | 17 |
| 2.5. A grammar of Dijkstra's Guarded Command Language | 18 |

1. Introduction

1.1. Structure of the practical sessions

The purpose of these practical exercises for IMPL are:

- to recapitulate and reinforce the theory; and
- to give you practice in programming the various algorithms needed for compilation (these run the gamut from simple algorithms over lists through to algorithms that calculate fixed points over directed graphs).

You should aim to complete the exercises in Chapter n in Week $n + 1$; there are two catch-up weeks, as well as Week 10 for slippage. There is an extension chapter that you may like to use as part of revision, as well as an advanced extension chapter. I recommend that you spend about 7 hours a week outside of the lecture and compulsory practical session. As each credit is meant to take 10 hours of effort, this leaves you with approximately 20 hours of revision time. Some exercises are marked with an asterisk; these are extension exercises.

The practical sessions, both compulsory and optional, will work best if you attempt the exercises *before* the session, and bring any questions on the material, both theory and practice, to the session.

1.2. Structure of a compiler

A compiler is (usually) designed as a pipeline, each component of the pipeline having its own theory (or theories). Some parts of the pipeline are small and simple (for example, the first, lexical scanning), others large and complex (for example, optimisation through liveness and reachability analysis). Hence some weeks will cover more than one topic, while some topics will cover more than one week.

2. Lexical scanning and parsing

2.1. Introduction

The first two stages of the compiler pipeline are:

lexical scanning (a.k.a. “lexing” or “scanning”) which converts a list of characters (a.k.a. a string) either to a *list of lexical tokens* or to an error message, and

parsing which converts a list of lexical tokens either to *an abstract syntax tree* or to an error message.

The abstract syntax tree of a valid program is a convenient way to represent the program, and is the basis of all further analysis and synthesis. See Figure 2.1 for an example.

Lexical scanning and parsing are solved problems, in the sense that programs exist that compile a description of lexical tokens into a program that turns a string into a list of tokens, and that compile a description of a grammar into a program that parses strings of lexical tokens into an abstract syntax tree.

We are going to use one program, BNFC¹, that does *both* jobs.

Ex. 2.1 Visit the BNFC website, bnfc.digitalgrammars.com, and make sure that you can find the documentation.

[You can download the tool from the website, for common architectures.]



Ex. 2.2 Read Chapter 2 of Ranta [1].

[Hint: See the EARL on the VLE, or the draft from the link on the IMPL website.]



2.2. Generating lexical scanners from regular expressions

Lexical scanning recognises substrings of characters, that form ‘words’ (including the ‘word’ which is a comment) and indicates its recognition of a type of word by

¹<http://bnfc.digitalgrammars.com/>

2. Lexical scanning and parsing

"if $x < y$ then $b := b - 1$ else $b := b + 1$ end"

↓ lexical scanning

[IF, VAR "x", LT, VAR "y", THEN, VAR "b", ASS, VAR "b", MINUS, NUM 1,
ELSE, VAR "b", ASS, VAR "b", PLUS, NUM 1, END]

↓ parsing

(Ite(Lt(Var x)(Var y))(Ass (Var b)(Minus(Var b)1))(Ass (Var b)(Plus(Var b)1)))

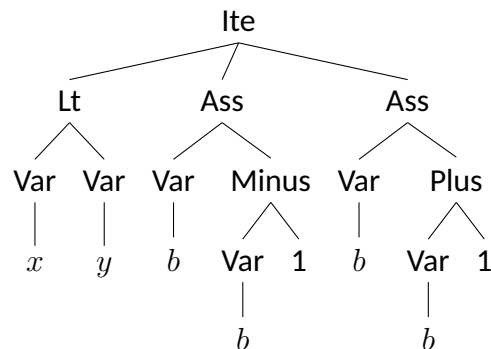


Figure 2.1.: An example of lexing and parsing. A string, its conversion to a list of lexical tokens, and thence to an abstract syntax tree (AST) are shown. The AST is presented in both a linearised form and graphical form.

2. Lexical scanning and parsing

issuing an appropriate token (or nothing in the case of a comment). Each type of word can be considered a string in a given language. It is enough to use regular languages. These can be described by regular expressions which are converted into finite state machines that recognise sentences by a lexical generator.


BNFC provides us with a concrete syntax for regular expressions, that is similar to “mathematical syntax”.

Ex. 2.3 Revise your MFCS (part B) notes on regular languages. □

Ex. 2.4 Read the BNFC documentation for lexer definitions². □

Ex. 2.5 ³

1. Take a copy of the file [INCOMPLETE_Element.cf](#)⁴ and rename it to [Element.cf](#). The file contains four missing regular expressions; each has been replaced by a question mark, ‘?’ (The second part of the file is the description of a grammar that allows any sequence of white-space separated tokens as an input.)

[Hint: Put the file in a directory of its own, as the BNFC compiler generates a *lot* of files.] 

2. Replace the questions marks as follows:


Cident Capitalised identifiers: the first letter must be upper case. There may be any number, including zero, of following characters in the classes: upper case letters, lower case letters, underscores, and single quotes

TeX \TeX commands: the first character must be a backslash ‘\’, and then any non-empty sequence of letters.

Weird A ‘weird’⁵ begins and ends with a percent symbol, ‘%’, and in between has either a non-empty sequence of letters or a non-empty sequence of digits.

Subscript (Approximations to) \TeX subscripts: the first letter must be an underscore; then either a single letter or an open-brace ‘{’ followed by a non-empty sequence of letters and digits and finally a close-brace, ‘}’.

3. Start up a terminal and change directory to the one in which you have placed your copy of [Element.cf](#).

[Hint: The usual Unix command to change directory is `cd <name—of—directory>.`] 

4. Compile the file by running:

`bnfc -m Element.cf`

If errors are reported, fix them and recompile until there are no errors. [Hint: If you 

²<https://bnfc.readthedocs.org/en/latest/lbnf.html#lexer>

³The instructions assume that you are running a Unix variant, such as Linux.

⁴http://www-module.cs.york.ac.uk/impl/Practical2/INCOMPLETE_Element.cf

⁵This was originally ‘word’, but because of a change to a Haskell library it caused a name-clash. A ‘weird’ is a more appropriate name for such an odd concept!

2. Lexical scanning and parsing

cannot follow the error messages, or if you can but cannot see how to fix the errors ask a demonstrator.]

When there are no errors you need to run:

```
make
```

This is the stage that creates lots of files.

5. You can now test the lexer on single tokens by running, for example:

```
echo "Abc" | ./TestElement
```

or by putting the string “Abc” in a file called `testEl` and running

```
./TestElement testEl
```

Try the following strings: “Abc”, “\maketitle”, “%aBc%”, “%543%”, “_a”, “_{a1b2c}”, all of which should generate tokens.

Try the following strings, all of which should generate lexical errors: “\8”, “%abc”, “%a6c%”, “{a1b2c}”

The generated program `TestElement` shows the result of parsing as well as lexical scanning. You can see the raw lexical tokens by running:

```
ghci LexElement.hs
```

This leaves you running an interpreted version of a programming language called Haskell. You should see a prompt “*LexElement>”.

If you type in a command such as:

```
tokens "Abc_\maketitle_%aBc%_%543%_a_{a1b2c}_\8_%abc_%a6c%_{a1b2c}"
```

you will get a list of tokens (note the double-backslash to represent a single backslash.). In the above case, the first group should all start “PT”; the first time a lexical error is found an error token is returned, and nothing further. Successfully scanned tokens are tagged with their position and type, error tokens just have a position.

[Hint: To quit `ghci` and return to the shell-level in your terminal give the command “:q”.]



Ex. 2.6* BNFC does not directly generate the FSM recogniser for tokens. Instead it generates a driver file for a lexer generator, which is then run by the `make` command. The lexer generator is called “alex”⁶. Its driver file for `Element` is called ‘`LexElement.x`’. The driver file is a mix of alex commands and utilities defined in Haskell.⁷ The key alex commands are preceded by a line `:—` and continue until the next open-brace (`{`) on a line of its own.

⁶<https://www.haskell.org/alex/>

⁷Other popular lexer generators, such as “flex” for C and C++, use a similar structure and command language.

2. Lexical scanning and parsing

Each line defines a token, and consists of a regular expression followed by an action (in braces) to take on a match.

Look at the relevant part of the driver file. Can you follow the definitions?

[Hint: abbreviations start with a dollar (\$) and are defined just before the line ':-'.]

The result of running `alex` on `LexElement.x` is in the file `LexElement.hs`. Can you see how the tokens are encoded? □

There are extension exercises in chapter 11, in which you may investigate generating lexers in a different way, by defining and using *lexer combinators*.

2.3. Generating parsers from grammars

As tokens are usually sentences in regular languages, programs are usually sentences in context-free languages. These are described by grammars, rather than regular expressions. The grammars are usually expressed in a notation called BNF (we will use a variant called labelled BNF, or LBNF). A grammar is converted to a push-down automaton to recognise sentences in a regular language.

Ex. 2.7 Revise your MFCS (part B) notes on context-free languages. □

Ex. 2.8 Read the BNFC documentation for parser definitions⁸. □

Ex. 2.9 We can see what the parser makes of our `Element` example (Exercise 2.5). Run:

```
ghci ParElement.hs
```

You now need to run a command of the form:

```
pListElement (myLexer "Abc_\maketitle_%aBc%_%543%_a_{a1b2c}")
```

(Note the double-backslash as a quoted single backslash.) The output is a linearised tree of tokens. In this case the tree is a list of tokens. □

The next pair of exercises are about a grammar to represent simple integer expressions.

Ex. 2.10

1. Take a copy of '`IntExp.cf`' and ensure that you understand it. [Hint: The token type `Integer` is predefined by BNFC, and is described in the documentation.]
2. Ensure yourself that the grammar described in '`IntExp.cf`' corresponds to the more usual presentation in Table 2.1.

⁸<https://bnfc.readthedocs.org/en/latest/lbnf.html>

2. Lexical scanning and parsing

$$I ::= I \text{ "+" } I \mid I \text{ "*" } I \mid \text{"NEG"} I \mid \text{Integer} \mid \text{"(" } I \text{ ")"}$$
 Integer expressions

Unary negation ("NEG") is the most tightly binding operator and addition the least tightly binding.

Both binary operators are right-associative.

Table 2.1.: A grammar for simple integer expressions in a standard BNF style. The lexical category "Integer" is standard, and omitted.

3. Compile it with BNFC (see Exercise 2.5).
4. Try it on a few examples, such as "`34+(16*2)`", "`(34+16)*2`", and "`34+16*2`". Experiment with a few others, including some strings that are not integer expressions, such as "`* 4 5`".
[Hint: This is most easily done by running `./ TestIntExp`, as we did `./TestElement` in Exercise 2.5].
5. Examine the driver file, `ParIntExp.y`, that BNFC prepares for the parser generator "happy". Can you follow what is going on?

□

Ex. 2.11

1. Extend the definition of integer expressions to include *variables*; do this in a file called `IntExpV`. Variables have the same grammatical status as integers, but their lexical structure is different.
[Hint: BNFC has a predefined token for variable names, 'Ident'.]
2. Try your new parser on expressions such as: "`x+(16*2)`", "`(x+16)*y`", "`apples+oranges_per_crate*crates`", "`34+(16*2)`", "`(34+16)*2`", and "`34+16*2`". (The last three are deliberately variable-free, and the same as those from Exercise 2.10.)

□

Ex. 2.12 Lispkit Lisp⁹ has a particularly simple syntax: everything is an S-expression. An S-expression is either null, written `()`, or an atom, or has two parts, the 'car' and the 'cdr', written `(car.cdr)`; both the car and the cdr are S-expressions. Atoms are either identifiers or integers. See Table 2.2.

Create a parser for Lispkit Lisp using BNFC.

[Hint: The first syntactic category defined in a file is the one parsed by the generated program `Test ...`.]

□

⁹A language similar to Scheme, but much, much smaller

2. Lexical scanning and parsing

| | |
|--|---------------|
| $S ::= "()" \mid A \mid "(" S "." S ")"$ | S-expressions |
| $A ::= \text{Integer} \mid \text{Ident}$ | atoms |

Table 2.2.: A grammar of Lispkit Lisp in BNF. The lexical categories “Integer” and “Ident” are standard, and omitted.

Ex. 2.13* Lispkit Lisp allows ‘list syntax’ syntactic sugar. For example, “ $(a\ b\ c)$ ” is shorthand for “ $(a.(b.(c.())))$ ”.

We can introduce a new grammatical category, $L ::= S\ L \mid S$ and add a production to S to capture this: $S ::= (L)$. Add this syntax to your definition of Lispkit grammars.

[Hint: BNFC has some features to help define syntax which is a list of items, although it is not necessary to use them. Look up the ‘[terminator](#)’ and ‘[separator](#)’ keywords in the BNFC documentation.] □

Ex. 2.14 Read Sections 3.5–3.9 of Ranta [1]. □

Two grammars may accept the same language but behave differently due to their structure. We will investigate this.

Ex. 2.15 Consider the two grammars **A** and **B**, of trivial expressions, in BNFC syntax:

```

AAdd. A ::= A "+" A ;
AMul. A ::= A "*" A ;
ANmb. A ::= Integer ;
ABrk. A ::= "(" A ")" ;

BAdd. B ::= B "+" B1 ;
B01. B ::= B1 ;
BMul. B1 ::= B1 "*" B2 ;
B12. B1 ::= B2 ;
BNmb. B2 ::= Integer ;
BBrk. B2 ::= "(" B ")" ;

```

Code these up, and try the result on “ 3_+4_*5 ” and “ 5_*4_+3 ”. What is the difference? [Hint: Draw the four abstract syntax trees, and evaluate them.] □

Grammar **A** in Exercise 2.15 is *ambiguous*: at least one string in its language has two different parses, that possibly give rise to different meanings. Grammar **B** defines exactly the same language, but has a unique parse for every string in the language. There is an algorithm to transform one into the other, and it would be nice if BNFC implemented it, but it does not: hence we must do it by hand; see Figure 2.2.

2. Lexical scanning and parsing

Inputs

- the original grammar,
- a binding priority for each operator (a number, higher means more tightly binding), and
- an associativity (left or right) for each infix operator.

Algorithm

1. Suppose infix operator ' $\&$ ' has priority p . Replace " $L. A ::= A \& A ;$ " by either
 - " $L. Ap ::= Ap \& A(p+1) ;$ " if ' $\&$ ' is left-associative, or
 - " $L. Ap ::= A(p+1) \& Ap ;$ " if ' $\&$ ' is right associative.
2. Suppose prefix (postfix) operator ' $\$$ ' has priority p . Replace " $L. A ::= \$ A ;$ " (" $L. A ::= A \$;$ ") by
 - " $L. Ap ::= \$ Ap ;$ " (" $L. Ap ::= Ap \$;$ ").
3. Constants and variables are treated as zero-place operators, with the highest possible priority. Suppose h is the highest priority assigned to an operator (infix, prefix or postfix), then replace $L. A ::= C ;$ by
 - " $L. A(h+1) ::= C ;$ ".
4. Replace bracketing " $L. A ::= "(" A ")" ;$ " by
 - " $L. A(h+1) ::= "(" A ")" ;$ ".
5. Link the levels: for each p in the range 0 to h add
 - " $Lp. Ap ::= A(p+1)$ " for some suitable label Lp .

BNFC shorthands

- BNFC treats A and $A0$ as synonyms, so it is usual to drop the ' 0 '. This has the advantage of the new grammar having the same name as the original.
- Some of the labels just add clutter to the abstract syntax trees generated, in particular, the labels for Steps 4 and 5. BNFC allows us to omit the labels, by using the pseudo-label ' $_$ '. With this simplification, Steps 4 and 5 introduce productions of the form:
 - " $_. A(h+1) ::= "(" A ")" ;$ " and
 - " $_. Ap ::= A(p+1) ;$ ", for each p in the range 0 to h .
- BNFC has a special syntax for adding the above unlabelled lines: "**coercions** $A (h+1) ;$ ".

Figure 2.2.: Disambiguation of grammars, given bindings and associativities

2. Lexical scanning and parsing

$S ::= \text{Identifier} \text{ ":=" } \text{IntExp} \mid \text{ "put" } \text{Ident} \mid S \text{ ";" } S$ statement

The words “put” and “NEG” are the only keywords of the language and may not be used as identifiers.

Table 2.3.: A grammar of a trivial language. The grammar of integer expressions, ‘IntExp’, is that of Exercise 2.11; the lexical category ‘Identifier’ is standard, and omitted.


Ex. 2.16 Take advantage of the [coercions](#) syntax to shorten the presentation of grammar B from Exercise 2.15. ☐

Ex. 2.17 Check that I got the transformation from A to B right (modulo change of name to B) in Exercise 2.15. ☐

In the exercises below you will need to disambiguate grammars.

Ex. 2.18 Consider a trivial language which just has assignment, a print statement and sequential composition. Its grammar is given in Table 2.3.


Create a parser for this language using BNFC.

[Hint: To get the grammar for integer expressions, just cut-and-paste your answer for Exercise 2.11.] 

☐

Ex. 2.19 Consider a second trivial language that has two types, assignment, a pair of print statements and sequential composition. A grammar is given in Table 2.4. [In the program “ $a, b : c : \dots$ ” the variables a and b have integer type, while c has Boolean type. Assignment statements must be homogeneous; the print statement “puti” can only handle integer variables, while “putb” can only handle Boolean variables.]

Create a parser for the trivial typed language, using BNFC.

[Hint: Parsing cannot be used to check types in general, so do not attempt to solve that problem.] ☐ 

Ex. 2.20* The original version of FORTRAN had a typing rule that could be checked by a parser: variable names that began with a letter in the range ‘I’..‘M’ had integer type, all others had float type.

Create a parser for a version of the trivial typed language that uses the type rule: variable names beginning with a letter ‘a’ or ‘b’ are Booleans and anything else is an Integer. The parser should check types.

[Hint: You will need to define your own lexical tokens to represent Boolean and Integer variables. You will need to replace V by two separate productions, one each for Boolean values and Integer values.] ☐ 

Ex. 2.21 Edsger W. Dijkstra designed a small, highly regular, imperative language [2]. It is usually known as Dijkstra’s Guarded Command Language (GCL)¹⁰, because the *guarded*

¹⁰It has also been called “LEWD”, the *Language* of EWD, and “NEWD”, the *Notation* of EWD.

2. Lexical scanning and parsing

| | |
|--|-------------------------------------|
| $P ::= I \text{ ";" } I \text{ ";" } S$ | programs |
| $S ::= \text{Identifier} \text{ ":=" } V \mid U \text{ Identifier} \mid S \text{ ";" } S \mid "(" S ")"$ | statements |
| $U ::= \text{"puti"} \mid \text{"putb"}$ | print statements |
| $V ::= \text{Identifier} \mid \text{Integer} \mid \text{"TRUE"} \mid \text{"FALSE"}$ | values |
| $I ::= [\text{Identifier}]^* \text{,}$ | comma-separated list of identifiers |

The language has keywords "puti", "putb", "TRUE", and "FALSE", which may not be used as identifiers.

Sequential composition (";") and identifier lists are right-associative.

Table 2.4.: A grammar of a trivial typed language. The lexical classes "Identifier" and "Integer" are standard, and omitted.

command is a characteristic feature. Guarded commands help to generalise both if-statements and while-loops¹¹. The language is also characterised by having a *parallel assignment* statement, where expressions are evaluated in the initial state. For example, we may write¹²:

$$x, y, z := z + 1, y + x, x * z$$

A syntax of GCL is given in Table 2.5. Create a parser for it using BNFC. You will need to choose ASCII representations for the symbols: " \square ", " \rightarrow ", and " \Rightarrow ".

[Hint: Again, you may wish to use BNFC's facilities for expressing lists of items.]

The grammar given has a very restricted definition of propositions. However, every propositional operator can be encoded as a combination of implication and the constant "false".

Table 2.5 says that the lengths of the lists of identifiers and expressions in a parallel assignment should be the same. Do not attempt to enforce this in your parser. \square

Ex. 2.22* It is possible to use syntax checking to check whether a parallel assignment is well-formed (equal numbers of identifiers and expressions), although *this turns out to be a bad idea* (see Exercise 2.23).

We can replace the production for assignment:

$$S ::= I \text{ ":=" } E$$

¹¹We will look at the *semantics* (meaning) later; now we are only interested in *syntax*.

¹²Again, at present we are only interested in syntax and postpone discussion of semantics.

2. Lexical scanning and parsing

| | |
|---|-------------------------------|
| $S ::= \text{"skip"} \mid I ::= E \mid \text{"if"} G \text{"fi"} \mid \text{"do"} G \text{"od"} \mid S ; S$ | statement |
| $G ::= P \rightarrow S \mid G \square G$ | guarded command |
| $P ::= \text{"FALSE"} \mid P \Rightarrow P \mid R$ | predicate |
| $R ::= \text{IntExpV} = \text{IntExpV} \mid \text{IntExpV} < \text{IntExpV}$ | relations |
| $I ::= [\text{Identifier}]^+_{,,}$ | non-empty list of identifiers |
| $E ::= [\text{IntExpV}]^+_{,,}$ | non-empty list of expressions |

The language has keywords: "skip", "if", "fi", "do", "od", "FALSE", and "NEG", none of which may be used as identifiers.

Sequential composition (";"), guarded command composition ("□"), identifier and expression lists are all right-associative.

In the construct $E := I$, E and I should have the same length.

Table 2.5.: A grammar of Dijkstra's Guarded Command Language (GCL). The grammar of integer expressions, 'IntExpV', is that of Exercise 2.11. The lexical category 'Identifier' is standard, and omitted.

by

$$S ::= A$$

$$A ::= \text{Identifier} ::= \text{IntExp} \mid \text{Identifier} , A , \text{IntExp}$$

This syntax demands that identifiers and expressions are balanced.

Amend a copy of your description of the syntax of Dijkstra's GCL so that it checks if assignment statements are well-formed. □

Ex. 2.23* To see the problem with making the balance of multiple assignment part of parsing, consider the statement that creates a state with x bound to 0, y bound to 1 and z bound to 2:

$$x, y, z := 0, 1, 2$$

Parse the statement with both the parser of Exercise 2.21 and of Exercise 2.22 and then draw the two resultant abstract syntax trees.

Can you see the problem? □

Ex. 2.24 Throughout these exercises we will build a compiler to a small machine language, M, from a small imperative language, N. Descriptions of the languages can be found in Appendices A and B. In particular descriptions of the syntax of the two languages can be found in section A.2 and section B.2.

Generate lexers and parsers for M and N using BNFC. □

3. Syntax-directed programming; type-checking

3.1. Syntax-directed programming

The first two phases of analysis by a compiler (lexing and parsing) leave us with an abstract syntax tree (or error messages). The best way of writing code for processing an abstract syntax tree is syntax-directed programming.

In syntax directed programming a datum is known to have one of a variety of syntactic forms. A function is defined in terms of a clause per syntactic form, the clause giving an expression whose value is to be returned. In practice, syntactic forms may be grouped in *patterns*. Here is an example of counting the number of items in a list, in the syntax of Haskell.

```
len [] = 0
len (x:xs) = 1 + len xs
```

The two syntactic forms of a list used here are:

1. the empty list, `[]`, and
2. a non-empty list, `(x:xs)` with a first element, `x` (the head of the list), and the remainder of the list, `xs` (the tail of the list).

In the case of:

the empty list the expression is `0`.

The non empty list the expression is one more than the length of the tail.¹

We are not limited to two patterns for lists. The pattern `"(x:y:xs)"` matches lists of at least length 2, and `"[x]"` matches a list of length exactly one. Further we can write patterns that include the structure of the elements of the lists. For example,

¹The pattern for the non-empty list can be improved slightly: as the expression does not use the value of the head we do not need to give it a name; instead we use an underscore in the pattern:
`len (_:xs) = 1 + len xs`

3. Syntax-directed programming; type-checking

`"[x,_,0]"` matches a list of length three, whose last element is zero, and whose second element is not needed in the associated expression; `"(x:xs):xss"` matches a non-empty list of lists whose first element is a non-empty list. Patterns allow us to avoid writing code for navigating through a data-structure.

Haskell is an ideal language for syntax-directed programming. Helpfully, BNFC generates Haskell to make it easy to integrate the lexer and parser with the remainder of the compiler code. An excellent, and short, text on Haskell programming is by Hutton [3], [4].

Ex. 3.1 Look in the directory where you created `IntExpV.cf` (Exercise 2.11) and processed it with `bnfc`. Among the many files you should find one called `AbsIntExpV.hs`. This file defines a module, also called `AbsIntExpV`, that contains the definition of a data type, `"IntExp"`, whose values are abstract syntax trees representing integer expressions that may contain variables.

A syntax tree can be of five types, one for each clause of the concrete syntax. Each type has a constructor which matches the name of the corresponding *label* in `IntExpV.cf`, and represents a tree node. Following each constructor is a number of type expressions (in this case just names) that tell us the number and types of the branches at that node.

There is also a type called `"Ident"` which is a copy of the `String` type, tagged with `"Ident"`, to distinguish it from other uses of strings.²

As part of their definition both types are furnished (`"deriving"`) with the `"obvious"` definitions of equality (`Eq`), ordering (`Ord`), and conversion to (`Show`) and from (`Read`) strings.

Draw as trees the following values of type `IntExp` and then convert them to their original concrete syntax:

```
Add (Neg (Var (Ident "x"))) (Mul (Nmb 1) (Var (Ident "y")))
Mul (Add (Nmb 3) (Nmb 4)) (Add (Nmb 0) (Neg (Nmb 1)))
```

□

Ex. 3.2

1. Take a copy of `INCOMPLETE_EvalIntExp.hs` and rename it to `EvalIntExp.hs`.
2. Open your renamed copy in a plain text editor/development tool³ You will see some expressions replaced by the special value `"error \"TBD\""`; eventually you will replace these.

The file loads two modules (`'AbsIntExpV`, created by BNFC, and `Data.Map` from the standard library). It then defines a new type synonym for `stores`: data structures that

²The type `"Ident"` could have been declared using `"data"`, but it is more efficient to use `"newtype"` in this special circumstance.

³There are many such tools under Linux: among them are: 'eclipse', 'emacs', and 'gedit'.

3. Syntax-directed programming; type-checking

record the value of variables, and two functions, one to evaluate variable-free expressions (`eval`) and one to evaluate any expression (`evalv`).

The function `eval` is defined in terms of five clauses, one for each possible type of node in the abstract syntax tree. This pattern is quite common, although sometimes a node needs to be represented by several clauses, and sometimes clauses can be coalesced.

The first clause of `eval`,

```
eval (Add a b) = eval a + eval b
```

evaluates trees with an 'Add' node at the root; it does this by summing the evaluations of the two sub-trees, 'a' and 'b'. The third clause is similar, and uses the special unary negation function, '`negate`', to avoid ambiguity. The fifth clause raises an exception, as this function should not be applied to expressions containing variables.

3. Complete the definition of '`eval`', by replacing the two occurrences of "`error "TBD"`" with suitable expressions.
4. Run your Haskell code in an interpreter. Either do this using the facilities of your chosen development environment or by running

```
ghci EvalIntExp
```

in a terminal. If you have made any lexical, syntax or type errors `ghci` will report them; you should correct your code and repeat until there are no errors.⁴

5. Try running a few examples in the interpreter. Include among them the two examples from Exercise 3.1.

□

We are now going to consider expressions with variables in them. The values of the variables are kept in a *store*, which is implemented as a finite function (or map). We can create store values in numerous ways. The first uses a combination of:

```
empty    -- the empty store
singleton i n -- a store that only associates identifier i with integer n
insert i n s -- a store like s, that also associates i with n
delete i s -- delete identifier i from store s
s \ t -- delete all elements of store t from store s
union s t -- a store with all the identifiers of s & t; in the case of a clash, s is preferred
```

The second converts from a list representation:

⁴Ask for help if you do not understand error messages!

3. Syntax-directed programming; type-checking

`fromList [(i, m), (j, n)]` — a store that associates *i* with *m* and *j* with *n*

Values can be accessed by using

`none s` — True if *s* empty, False otherwise
`member i s` — True if identifier *i* exists in *s*, False otherwise
`s ! i` — the value associated with identifier *i*, or an exception if none
`lookup i s` — "Just *n*" if "member *i s*", "Nothing" otherwise.

A full list of functions for `Data.Map` can be found at

<http://hackage.haskell.org/package/containers-0.4.0.0/docs/Data-Map.html>.

Ex. 3.3 Use the interpreter to write expressions that evaluate to:

1. a store that associates identifier "x" with 3, and nothing else;
2. a store that associates the identifier "x" with 0, the identifier "y" with 27, the identifier "z" with 16, and nothing else;
3. the value associated with the identifier named "x" in the store of Part 1, or an exception if there is not one; and
4. the value associated with the identifier named "y" in the store of Part 1, or an exception if there is not one.

□

Ex. 3.4 We now turn our attention to the second function in `EvalIntExp.hs`, `evalv`. This function takes two parameters: the first is a store, and the second the expression (which may contain variables) to be evaluated.

The definition of `evalv` is similar to that of `eval`, except that now a variable only causes an exception if it is not present in the store. If it is present in the store, it is replaced by its associated value.

1. Complete the definition of `evalv`.
2. Add definitions to your file for the second store of Exercise 3.3. The skeleton is:

```
xyz :: Store
xyz = error "TBD"
```

[Hint: If you wish, you can add definitions for the two expressions of Exercise 3.1, too.]

[Hint: It would be cleaner to create a new module, in a new file, that imported `EvalIntExp` and defined the test values.]

3. Use `evalv` to evaluate the two expressions of Exercise 3.1 with the second store of Exercise 3.3.

3. Syntax-directed programming; type-checking

□

Ex. 3.5 Finally in this section we build a calculator for variable-free expressions.

1. Create a module, in its own file, called “`Calculator`”.
2. The file should import both “`ParIntExpV`”, for access to lexing (`myLexer`) and parsing (`plntExpV`) functions, and “`EvalIntExpV`”, for access to the evaluation function (`eval`) and “`ErrM`”, for access to error types (`Err a`, for arbitrary type `a`).
3. The parser may return an error. Add the following error handler to your code:

```
handleErr :: Err a -> a -- simple handling of error messages
handleErr (Ok t) = t
handleErr (Bad s) = error s
```

4. Now define a calculator as a pipeline:

```
calc :: String -> Integer -- variable-free expressions
calc = eval . handleErr . plntExp . myLexer
```

5. Experiment with applying `calc` to strings, such as “`65+3*2`”, that contain concrete expressions. Include the two concrete strings of Exercise 3.1 (one should give an error). Also include strings with lexing errors and syntax errors.

You should see that BNFC does not generate lexers and parsers that give helpful error messages.

□

3.2. Type checking

Type-checking has two uses:

Analysis To reject lexically and syntactically correct code which is still nonsense (for example, $x := 5 + 'a'$).

Code generation For correct code, to give hints to the compiler (for example, about the space needed to store a temporary result).

Type-checking is rather like evaluating an expression, but a true/false answer is given (and also a helpful error message).

Type checking rules are often described using “Natural Deduction Inference Rules”, originally designed to describe logics. A proof is a tree with inference rules as branches. In the case of type systems the proofs are proofs of type-correctness.

3. Syntax-directed programming; type-checking

$$\begin{array}{c} \wedge\text{-I} \frac{\Gamma \vdash P \quad \Delta \vdash Q}{\Gamma, \Delta \vdash P \wedge Q} \\ \\ \wedge\text{-EL} \frac{\Gamma \vdash P \wedge Q}{\Gamma \vdash P} \qquad \wedge\text{-ER} \frac{\Gamma \vdash P \wedge Q}{\Gamma \vdash Q} \end{array}$$

Figure 3.1.: Proof rules for conjunction

$$\wedge\text{-ER} \frac{\Gamma \vdash P \wedge Q}{\Gamma \vdash Q} \quad \wedge\text{-EL} \frac{\Gamma \vdash P \wedge Q}{\Gamma \vdash P} \quad \wedge\text{-I} \frac{\Gamma \vdash Q \quad \Gamma \vdash P}{\Gamma \vdash Q \wedge P}$$

This proof can be summarised as a theorem:

$$\wedge\text{-commutativity} \frac{\Gamma \vdash P \wedge Q}{\Gamma \vdash Q \wedge P}$$

Figure 3.2.: A proof of commutativity of conjunction

In Figure 3.1 are the classical proof rules for conjunction; in Figure 3.2 is a proof of the commutativity of conjunction.

The general shape of a rule is:

$$\text{Name of rule} \frac{\text{Antecedent}_0 \quad \dots \quad \text{Antecedent}_n \quad [\text{Side conditions}]}{\text{Consequent}}$$

It relates the truth of the consequent to that of the antecedents and side condition. There may be no side conditions. There may be no antecedents; in this case we call the rule an *axiom*.

We can understand the rule in a *forward* manner:

if the antecedents are true, and the side conditions are true then the consequent is true,

or in a *backward* manner:

if we wish to prove the antecedent then a way to do it is to prove the antecedents and establish the side conditions.

The backward reading is often much more useful, as we know what we would like to prove, and then try and construct a proof.

3. Syntax-directed programming; type-checking

Each antecedent or consequent is a *judgement*. A judgement is composed of two parts separated by a “turnstile”, ‘ \vdash ’:

$$\Gamma \vdash C$$

It has meaning “the conclusion, C , is true in the context Γ ”. An example from the world of type-checking is:

$$x : \text{Int}, y : \text{Int}, z : \text{Int} \vdash y + z : \text{Int}$$

Here the context is a collection of simple statements about types of variables, and the conclusion a statement about the type of an expression. An example of an inference rule about types is:

$$+I \frac{\Gamma \vdash x : \text{Int} \quad \Gamma \vdash y : \text{Int}}{\Gamma \vdash x + y : \text{Int}}$$

This rule, called “ $+I$ ”, states that to show the (complex) expression $x + y$ has type Int in context Γ it is enough to show that the two simpler subexpressions, x and y also have type Int in context Γ .

It is easy to implement this rule in Haskell. First we introduce some types to represent types and contexts:

```
data TypeName = ... | IntType | ... deriving (Eq,Show)
type Context = [Typing]
```

Next, we define a function, `tx` say, that has this rule as a clause. The expression “`tx c e t`” is true exactly when expression e has type t in context c .

```
tx :: Context -> Expression -> TypeName -> Bool
...
tx c (Plus x y) IntType = tx c x IntType && tx c y IntType -- +I
...
```

Note how this has the structure:

```
Consequent = Antecedent0 && Antecedent1 -- Name-of-clause
```

The other clauses reflect the other proof rules, also in a syntax directed fashion. Side conditions are treated exactly like antecedents. They differ in that they are *not* judgements, but “extra-logical” statements. For example, in Figure 3.4, the rule ‘IdentB’ says that we can conclude that the expression v has type B if v is a variable found in the first list of variables names, and ‘IdentI’ says that we can conclude it has type I if found in the second list. “Finding a variable in a list” is *not* a statement

3. Syntax-directed programming; type-checking

| | |
|--|------------------|
| $p ::= "(i; i; e)"$ | top-level |
| $i ::= [\text{Identifier}]^*, "$ | identifier lists |
| $e ::= e \implies e \mid e + e \mid e r e$ | expressions |
| $\quad \mid "\perp" \mid \text{Integer} \mid \text{Identifier} \mid "(e)"$ | |
| $r ::= "=" \mid "<"$ | relations |

Addition (+) binds most tightly, implication (\implies) binds least tightly with the relational operators between them. Implication and addition are right-associative.

Figure 3.3.: Syntax of a small expression language

about an expression having a type, and so is a side-condition rather than an antecedent.

Consider the small language of variable-containing Boolean expressions with syntax given in Figure 3.3. In the example $(a, b; c, d, e; a \implies c = d + e)$ the expression $a \implies c = d + e$ is in terms of Boolean variables a and b and integer variables c , d and e , and is type-correct. On the other hand, $(a, b; c, d, e; c \implies a)$ is not type-correct.

Typing rules are given in Figure 3.4. A judgement $b; n \vdash e : T$ says that in the context of a list of Boolean variables, b and a list of integer variables n , expression e has type T . There are only two types, B (Boolean) and I (integer).

Four of the rules have side-conditions. The rule ‘top’ requires the lists of Boolean and integer variables to be distinct. The rule ‘int’ requires the expression n to be an integer constant. The rules ‘identB’ and ‘identI’ require the expression v to be a variable found in the first or second lists, respectively, in the context.

A proof of the type-correctness of $(a; b, c; (b < c \implies \perp) \implies a)$ is given in Figure 3.5.

Ex. 3.6 Consider the language of typed expressions defined in Figure 3.3 and Figure 3.4.

1. Prove $(a, b; c, d, e; a \implies c = d + e) : B$.
2. What happens if you try and prove $(a, b; c, d, e; c \implies a) : B$?
3. Create a BNFC driver file for the language, [Small.cf](#).
4. Create a Haskell typechecker for the parsed language, [TypeSmall.hs](#), by taking a copy of [INCOMPLETE_TypeSmall.hs](#), removing the ‘INCOMPLETE_’ from its name, and completing it.
5. Take a copy of [CompSmall.hs](#) and use it to test your type-checker.

3. Syntax-directed programming; type-checking

$$\begin{array}{c}
\text{top} \frac{b; n \vdash e : B}{\vdash (b; n; e) : B} [b \cap n = \emptyset] \\
\\
\text{implies} \frac{\Gamma \vdash x : B \quad \Gamma \vdash y : B}{\Gamma \vdash x \Rightarrow y : B} \qquad \text{plus} \frac{\Gamma \vdash x : I \quad \Gamma \vdash y : I}{\Gamma \vdash x + y : I} \\
\text{rel} \frac{\Gamma \vdash x : I \quad \Gamma \vdash y : I}{\Gamma \vdash x \text{ r } y : B} \\
\\
\text{false} \frac{}{\Gamma \vdash \perp : B} \qquad \text{int} \frac{}{\Gamma \vdash n : I} [n \in \mathbb{Z}] \\
\text{identB} \frac{}{b; n \vdash v : B} [v \in b] \qquad \text{identI} \frac{}{b; n \vdash v : I} [v \in n] \\
\\
\text{bracketB} \frac{\Gamma \vdash e : B}{\Gamma \vdash (e) : B} \qquad \text{bracketI} \frac{\Gamma \vdash e : I}{\Gamma \vdash (e) : I}
\end{array}$$

Figure 3.4.: Typing rules for a small language of expressions

Lemma:

$$\begin{array}{c}
\text{identI} \frac{}{a; b, c \vdash b : I} [b \in \{b, c\}] \quad \text{identI} \frac{}{a; b, c \vdash c : I} [c \in \{b, c\}] \\
\text{rel} \frac{}{a; b, c \vdash b < c : B} \qquad \text{false} \frac{}{a; b, c \vdash \perp : B} \\
\text{implies} \frac{}{a; b, c \vdash b < c \Rightarrow \perp : B}
\end{array}$$

Main proof:

$$\begin{array}{c}
\text{Lemma} \frac{}{a; b, c \vdash b < c \Rightarrow \perp : B} \quad \text{identB} \frac{}{a; b, c \vdash a : B} [a \in \{a\}] \\
\text{implies} \frac{}{a; b, c \vdash (b < c \Rightarrow \perp) \Rightarrow a : B} \\
\text{top} \frac{}{\vdash (a; b, c; (b < c \Rightarrow \perp) \Rightarrow a) : B} [\{a\} \cap \{b, c\} = \emptyset]
\end{array}$$

Figure 3.5.: A proof in the small type system. To fit the proof on the page, it has been split into a lemma $(a; b, c \vdash b < c \Rightarrow \perp : B)$ and the main proof.

3. Syntax-directed programming; type-checking

```
typecheckN :: Stm -> Err Stm -- wrapper for tn
tn :: Stm -> Result -- top level
type Result = String -- rename String, for documentation
tp :: [Dec] -> [Proc] -> [Proc] -> Result -- procedure declarations
ts :: [Dec] -> [Proc] -> Stm -> Result -- statements
te :: [Dec] -> [Proc] -> Exp -> Type -> Result -- expressions
```

Listing 3.1: Declarations for a typechecker for N

□

Ex. 3.7* The error messages given by the type-checker in Exercise 3.6 are not very helpful. Take a copy of the code in a file named `TypeSmallG.hs`. Do not forget to update the module name in the code!

Modify the code so that it reports if a variable is used in an incorrect context. For example, type-checking `(a;b;a=>b)` and `(a;c;a=>b)` should both return 'Non-Boolean variable "b" used as Boolean'.

[Hint: Two clauses need to be altered, two need to be inserted and one may be deleted.]

□



Ex. 3.8* We can improve the error messages even more by replacing a Boolean answer by a string. The empty string encodes 'True', anything else encodes 'False' (it helps if the string is terminated by a newline character, `'\n'`). Conjunction of Booleans is replaced by concatenation of strings.

Take another copy of `TypeSmall`, rename it to `TypeSmallS` and make the appropriate modifications.

□

A further improvement is possible, by using an advanced feature of the BNFC lexer to attach line and row numbers to tokens; these line and row numbers can then be used as part of the error message. If you are interested, see the BNFC documentation for the `position` keyword.

Ex. 3.9 Build a type checker for N, based on subsection B.3.2. The code should go in a module called `TypeN`, and define a top-level function `typecheckN` that takes an abstract-syntax representation of an N program and returns `True` exactly when the N program is type-correct.

[Hint: Each syntactic class that needs checking should have its own type-checking function. You will need at least the functions in Listing 3.1. The context, a list of variable declarations and a list of procedure declarations are the first two parameters. The value to be checked is the third parameter. In `'te ds ps e t'`, the fourth parameter, `'t'` is the type that `'e'` is checked against.]

□



Ex. 3.10* Modify your type checker for N in the manner of Exercise 3.8 so that it returns a string, containing all the type-error messages generated by type-checking.

3. Syntax-directed programming; type-checking

[Hint: To print out variable names, expressions, and so on, load the module '[PrintN.hs](#)' and use the function '[printTree](#)', which can be applied to values of any of the syntactic classes defined in [N.cf.](#)]

□

Ex. 3.11 The rule 'NproclisT' enforces a definition-before-use discipline for procedures. Modify the rule to relax this requirement. □

3.3. Pre-compilation optimisation

Once code has been shown to have no errors it may be simplified. For example, an assignment such as ' $y := (x+2) * (3+4)$ ' may be replaced by ' $y := (x+2) * 7$ '. Simplification rules for N expressions are given in subsection B.3.3.

Ex. 3.12 Why is there no rule for a constant true guard in an iteration, in subsection B.3.3? Rules such as associativity and commutativity of operators should not be implemented in a basic simplifier. Why not? □

Ex. 3.13 Take a copy of [INCOMPLETE_SimplifyN.hs](#) and complete it to give a top-down simplifier for N. □

Ex. 3.14 Try the simplifier on

```
{x: Int | |x:=5*(1+0); print x}
```

What is the problem with top-down simplifiers? □

Ex. 3.15* Solve the problem revealed by Exercise 3.14, by creating a bottom-up simplifier. Given a data type definition ' $\text{data } T = A \mid C \ T \ T$ ', a top-down function ' $f :: T \rightarrow T$ ' with a collection of clauses that analyse the structure of a constructor, for example:

```
f (C A u) = g (f u)
f (C t A) = h (f t)
f (C t u) = C (f t) (f u)
```

can be re-written to be bottom-up by changing the clauses to be:

```
f (C t u) = c (f t) (f u)
where
  c A u' = g u'
  c t' A = h t'
  c t' u' = C t' u'
```

□

4. Interpreters: operational semantics

There are two kinds of interpreters:

- those intended to give an efficient implementation of the language, and
- those intended to efficiently give the meaning of the language.

We will only be concerned with the latter kind, which are direct implementations of an *operational semantics* of the language.¹

Operational semantics come in two forms:

Big-step also called “Natural Semantics”. The rules encode reasoning about the execution of the whole program.

This is the simpler of the two, but cannot handle some programming language constructs well, such as jumps and concurrency. It is adequate for N, but not M.

Small-step also called “Structural Operational Semantics”. The rules encode reasoning about an atomic step of the program.

This is more complex, but can handle jumps and interleaving concurrency. It is adequate for M.

Just like type rules, operational semantics rules are traditionally presented as Natural Deduction rules.

A big-step judgement is about a heterogeneous relation between program/start-state pairs and end-state/output pairs. The context of the judgement is the same as for type rules: the types of variables and types/definitions of procedures. In the case of type rules the definition part of the procedure is irrelevant (as long as it is type correct), but it is relevant for operational semantics! In N we do not need the types of variables in the context, but some languages may. See section B.4 for a big-step semantics of N.

¹There are many ways of describing the semantics (“meaning”) of a language: operational semantics is traditionally used by compiler-writers as it shows how to execute each construct. One problem with operational semantics is that the meaning of a non-terminating program is an infinite object, which in our case will be a non-terminating interpretation.

4. Interpreters: operational semantics

A small-step judgement is about a homogeneous relation over program/start-state/output-buffer triples. Contexts are the program being executed. See section A.4 for a small-step semantics of M.

Note Ranta’s representation of the semantics [1, §5] is organised differently, and uses different symbols, but the underlying concepts are the same.

An important use for the semantics-based interpreters is *property-based testing* for a compiler. Suppose the semantics of N is given by the function

‘ $\text{semN}::\text{String} \rightarrow [\text{Integer}]$ ’, that for M by ‘ $\text{semM}::\text{String} \rightarrow [\text{Integer}]$ ’ and we have a candidate compiler

‘ $\text{compN2M}::\text{String} \rightarrow \text{String}$ ’. Then the function:

```
testc :: String -> Bool
testc n = semN n == (semM . compN2M) n
```

captures the correctness criterion for a compiler. We can arrange for `testc` to be applied to a representative sample of candidate programs, using a utility such as QuickCheck² or SmallCheck³. Using these tools is outside the scope of this module.

Ex. 4.1 Use the big-step semantics of N to prove that:

$$\vdash \{x : \text{Int}; y : \text{Int} \mid \mid x := 1; y := 2; \text{print } x + y\} \Downarrow\Downarrow\langle 3 \rangle$$

□

Ex. 4.2 Use the big-step semantics of N to prove that:

$$\vdash \{x : \text{Int} \mid \mid x := 0; \text{while } x < 1 \text{ do } x := x + 1; \text{print } x \text{ end}\} \Downarrow\Downarrow\langle 1 \rangle$$

□

Ex. 4.3 What happens if you try and construct a proof for:

$$\vdash \{ \mid \mid \text{while true do skip end} \} \Downarrow\Downarrow\epsilon$$

□

Ex. 4.4 Let p be the program:

```
movi x 1
movi y 2
add z x y
prn z
hlt
```

Construct a proof of $\vdash p \Rightarrow \langle 3 \rangle$.

□

²<https://en.wikipedia.org/wiki/QuickCheck>

³<https://www.cs.york.ac.uk/fp/smallcheck/>

4. Interpreters: operational semantics

Ex. 4.5 What happens if we try and prove something about a program such as:

```
here : jmp here
      hlt
```

□

As with type-checking we can create code from an operational semantics; in this case it builds an interpreter.

Suppose we want to implement a relation \approx , that is actually a function. (All of our relations will be functions, as our language is deterministic.) First choose a name for function, extended to take the context as a parameter, say f . A proof rule such as

$$\text{Name} \frac{\Gamma \vdash a \approx b \quad \Gamma \vdash b \approx c}{\Gamma \vdash a \approx h\ c} [g]$$

is converted into a clause of the definition of f :

```
f gamma a | g = h c -- Name
  where
    b = f gamma a
    c = f gamma b
```

The consequent is the main clause. A side-condition which is a Boolean becomes a guard, a side-condition which calculates a value should be treated as an antecedent. An antecedent becomes a clause of the “where” block *in reverse*. The name becomes a comment.

Ex. 4.6 Create a module, `SemN`, that animates the big-step semantics for N. To do this, take a copy of `INCOMPLETE_SemN.hs`⁴ and replace all the expressions “`error "TBD"`”.

[Hint: To test your module, create a module that loads it, as well as the parser and type-checker. Define a function that applies the semantic function to the result of lexical scanning, parsing and type-checking a string. You will need to deal with errors in the analysis.] □



Ex. 4.7* Create a module, `SemM`, that animates the small-step semantics for M.

Note that this will need to calculate the maximal transitive closure of \longrightarrow for each input program (that is, until a halt instruction is reached). [Hint: The standard prelude function `until` is useful for this.]



[Hint: BNFC generates code that uses `Integer` values. However, many standard Haskell functions use `Int`. Generic versions of these functions, that will accept an `Integer` exist in the library module `Data.List` with names such as “`genericIndex`” and “`genericLength`”; consult `Hoogλe`.] □



Ex. 4.8* N only has Integer and Boolean variables, and can only print Integers.

Modify the semantics so that it is possible to print a Boolean variable. This should *not* require an extra print statement. The value ‘True’ should be printed as ‘1’ and the value ‘False’ as ‘0’. □

□

⁴http://www-module.cs.york.ac.uk/impl/Practical2/INCOMPLETE_SemN.hs

5. Catch-up week

The lecture in Week 5 is replaced by a formative assessment.

6. Code generation for an unlimited register machine

Ex. 6.1 Show how to compile the statement $x := y + z$, for integer variables x, y and z , first using the simplistic rules and then the more sophisticated rules. Which gives the more efficient code?

You should present the compilations as proof trees.

[Hint: In order to complete the right-hand side of \rightsquigarrow and \hookrightarrow , build a tree with just the left hand-sides, until leaves are reached, and then work back down the tree.] □

Ex. 6.2 In Appendix C there are no rules for assigning Boolean expressions. Provide them, assuming that 0 encodes \perp (false) and all other values encode \top (true). For example,

$$\text{NMAssF} \frac{}{b := \text{false} \rightsquigarrow \boxed{\text{movi } b \ 0}}$$

[Note: we are encoding Booleans as machine Integers. Life would be easier if there were M instructions for Booleans.] □

Ex. 6.3* An incorrect answer for ‘NMAssOr’ is:

$$\text{NMAssOrBAD} \frac{x \hookrightarrow (\boxed{c}, v) \quad y \hookrightarrow (\boxed{d}, w)}{b := x \text{ or } y \rightsquigarrow \boxed{\begin{array}{l} c \\ d \\ \text{add } b \ v \ w \end{array}}} [v \text{ fresh}, w \text{ fresh}]$$

There is a similar incorrect answer for ‘NMAssAnd’, using ‘mul’.

Why are these not solutions?

[Hint: Consider the difference between numbers and the machine approximation to numbers.] □

Ex. 6.4 In Appendix C there is no rule ‘NMbr=’ for calculating a jump dependant on $a = b$. Provide one. □

Ex. 6.5 Provide the rules for \nrightarrow , which are dual to the rules for \rightarrow . □

6. Code generation for an unlimited register machine

Ex. 6.6 Consider a language, D, which is a subset N. A D program consists of a single print statement of a variable-free Integer expression. The legal strings of D are given by the grammar:

$$\begin{aligned} P &::= \text{"print"} E \\ E &::= \text{Integer} \mid \text{"neg"} E \mid E \text{" * " } E \mid E \text{" + " } E \end{aligned}$$

Write a compiler from D to M, using the relevant rules in Appendix C. You may ignore the context in the sequents, as it consists of variables and procedures in scope: D has neither. [Hint: As every legal D program is also a legal N program you may use the N front-end to scan, parse and type-check D. If you wish you may add a further function to the pipeline to check the restriction, or even build a whole new front end, but that is not the point of this exercise.¹ You should not put a simplifier into the pipeline (why?).]

[Hint: Use the standard pattern for converting proof rules to Haskell function definitions.] ☐

Ex. 6.7 Consider a language, F, which is a subset of N. An F program consists of either a print statement or a block. The block may declare a single Integer variable, but not a procedure or Boolean variable. The statement of the block is a possibly empty sequence of assignments followed by a print statement. The grammar of F is:

$$\begin{aligned} P &::= \text{"print"} E \mid \text{"{" Identifier " : " Int " ; " S "} \\ S &::= \text{"print"} E \mid \text{Identifier " := " } E \text{" ; " } S \\ E &::= \text{Integer} \mid \text{"neg"} E \mid E \text{" * " } E \mid E \text{" + " } E \mid \text{Identifier} \end{aligned}$$

Create a compiler for N. The phases in the pipeline prior to code generation may be identical to those for N; you may choose to construct a more restrictive parser, although this is not the point of this exercise.

[Hint: As with Exercise 6.6 no context is needed in a sequent, because there are no procedures, and no sub-blocks. As there may be at most one block, and it is outermost it is compiled in an empty context; hence it is enough to compile the statement embedded in the block.] ☐

Ex. 6.8 MiniN is a version of N with no procedures or nested blocks (a single, outermost block is allowed: this is the only way to declare variables.), but with all other features. Implement a compiler from MiniN to M. ☐

¹A better version of D dispenses with the keyword "print", but that would force the building of a new scanner and parser.

7. Optimisation

Ex. 7.1 Give an M program which has a `movi r9 0` instruction and a following `add r0 r0 r9` instruction, but where the constant cannot be propagated. □

Ex. 7.2 Give a code fragment that has a `movi r9 0` instruction and a following use of the register, to where the constant can be propagated, but the `movi r9 0` instruction does not become dead. □

Ex. 7.3 Take a copy of the files [CFG.hs](http://www-module.cs.york.ac.uk/impl/Practical2/CFG.hs)¹ (which defines a structure for control flow graphs and a function to reverse them), [FixedPoint.hs](http://www-module.cs.york.ac.uk/impl/Practical2/FixedPoint.hs)² (which defines a function for calculating fixed points), [INCOMPLETE_TablesM.hs](http://www-module.cs.york.ac.uk/impl/Practical2/INCOMPLETE_TablesM.hs)³ (which defines various tables to define functions of instructions) and [INCOMPLETE_OptimiseM.hs](http://www-module.cs.york.ac.uk/impl/Practical2/INCOMPLETE_OptimiseM.hs)⁴ (which defines part of the structures needed to implement constant propagation and dead code removal).

Complete [TablesM.hs](http://www-module.cs.york.ac.uk/impl/Practical2/TablesM.hs) and [Optimise.hs](http://www-module.cs.york.ac.uk/impl/Practical2/Optimise.hs), and add constant propagation and dead code elimination to the back end of the compiler.

[Hint: In [TablesM.hs](http://www-module.cs.york.ac.uk/impl/Practical2/TablesM.hs) you will find a full table embedded in the control-flow-graph construction algorithm, and an outline table which is partially filled in; the remaining tables do not have an outline, but are similar to the other two tables.] □

Ex. 7.4 Every label introduced is attached to a no-op instruction. We can use peephole optimisation to get rid of redundant ‘nop’ instructions.

The optimisation uses a peephole of size two:

$$\begin{array}{|c|} \hline L_0 : \dots : L_m : \text{nop} \\ M_0 : \dots : M_n : \text{instr} \\ \hline \end{array} \longrightarrow \begin{array}{|c|} \hline L_0 : \dots : L_m : M_0 : \dots : M_n : \text{instr} \\ \hline \end{array}$$

All labels attached to a no-op instruction are moved to the following instruction.

Implement a function ‘[removeNopM::M → M](#)’ that implements the rule. It should live in a module called ‘[PeepholeM](#)’.

Add the module to the compiler and the function to the back end.

You should now have an optimising compiler. □

¹<http://www-module.cs.york.ac.uk/impl/Practical2/CFG.hs>

²<http://www-module.cs.york.ac.uk/impl/Practical2/FixedPoint.hs>

³http://www-module.cs.york.ac.uk/impl/Practical2/INCOMPLETE_TablesM.hs

⁴http://www-module.cs.york.ac.uk/impl/Practical2/INCOMPLETE_OptimiseM.hs

8. Register allocation

Ex. 8.1 Take a copy of '[INCOMPLETE_RegisterAllocM.hs](#)'¹ and complete it by replacing all the occurrences of "**error** \"TBD\"".

[Hint: You will need to program a formula of the pattern $\bigcup_{x \in xs} f\ x$. This can be achieved by '**foldr union [] [f x | x <- xs]**'.]



Ex. 8.2* The code presented is not as efficient as it could be, as it converts virtual registers to physical/spillable registers more than once. Create a version that only does this once.

[Hint: The version in [INCOMPLETE_RegisterAllocM.hs](#) computes it at least three times, as it computes accessed spilt registers, assigned spilt register and registers in the instruction separately. By modifying [TableM.hs/repreg](#) to return a list of instructions, and more cases per instruction, you can create faster code.]



¹http://www-module.cs.york.ac.uk/impl/Practical2/INCOMPLETE_RegisterAllocM.hs

9. Catch-up week

The lecture in Week 9 is replaced by a formative assessment.

10. Extension: Blocks and procedures

Ex. 10.1* Extend the compiler to deal with blocks. ☐

Ex. 10.2* Extend the compiler to deal with dynamically scoped procedures. As part of this you will need to modify [TablesM.hs](#), the file that documents certain basic properties of M. In particular, the function `cfg`, which creates a control flow graph, is undefined for the `jmp` instruction, but needs to be defined. One possibility is to define:

`Jmp → [0..numLines−1]`

That is, a jump might be to *any* line in the code. Unfortunately, this is inefficient, and does not allow as much improvement as possible. However, code derived by compiling N cannot generate a jump to any location, but only to just after calls to a procedure. Implement this better definition. ☐

Ex. 10.3* Extend the compiler to deal with statically scoped procedures.

[Hint: The function `fresh` can be used to generate new procedure names.] ☐



11. Extension: Lexer combinators

The BNFC approach to lexing is *syntactic*: a text representing a regular expression is compiled into code (the function `'tokens'` in module `Lex\ldots`, renamed to `'myLexer'` in module `Par\ldots`). The advantage is that programs (in particular BNFC) can analyse the text.

Another approach is *semantic*, where the regular expression is represented directly as code. In this chapter you will build some *lexer combinators*. These are higher-order functions that take as input (functions representing) regular expressions and return a (function representing a) bigger regular expression. You will also need some atomic regular expressions to get things going; you will also build these.

The idea is to represent RegExps as a Haskell type:

```
type RegExp a = [a] -> (Maybe [a], [a])
```

The first component tells us if a successful lexing has occurred, and if so what token was found. The second component is the string that remains to be processed; if a successful lexing has occurred then it is the original string with the token removed, otherwise it is the original string.

To take a concrete example, we can represent the regular expression that is the single character x , denoting the language $\{ "x" \}$ by

```
re_x :: RegExp Char
re_x ('x':t) = (Just "x", t) -- 'x':t does start with 'x'
re_x s      = (Nothing, s)  -- nothing else starts with 'x'
```

so, for example:

```
re_x "xylem" == (Just "x", "ylem" )
re_x ""      == (Nothing, ""      )
re_x "phloem" == (Nothing, "phloem")
```

More formally, the meaning of an instance, $r :: \text{RegExp } A$, is described as follows. Suppose $r \text{ s} == (m, t)$, then:

1. if s starts with a sentence, q say, of the language represented by r then $m == \text{Just } q$ and $s == q ++ t$,

11. Extension: Lexer combinators

- otherwise, `m==Nothing` and `s==t`.

Ex. 11.1*

- Start a new Haskell module called `LexerCombinator` that imports `Data.Maybe`.
- Declare the type `RegExp`.
- Given `r :: RegExp a`, we could apply it once to a list, `s :: [a]`, to get the first instance of `r` in `s`, if one exists. It is more convenient to repeatedly apply `r` to `s` to break it into pieces. This is done by the following incomplete function:

```
chop :: (Eq a, Show a) => RegExp a -> RegExp a -> [a] -> [[a]]
-- chop regexpWhitespace regexpTokens source -> listOfTokens
chop w r = rrw
  where
    rrw s | s' == [] = [] -- nothing left to process
          | isJust m = fromJust m : rrw t -- regexp found
          | otherwise = error ("\\n\\tlexing _error:_ " ++ show (take 8 t))
    where
      (_, s') = ? -- swallow white space
      (m, t)  = ? -- grab next lexeme
```

Complete the definition.

- We could declare a large number of functions along the lines of `re_x`, but this would be tedious. Instead we can declare a function that, given a symbol in our chosen alphabet gives back the `RegExp` that recognises that symbol.

Define `symb :: Eq a => a -> RegExp a` to do this. In particular, `symb 'x'` will have exactly the same behaviour as `re_x`.

It is also useful have the dual function, `anybut :: Eq a => a -> RegExp a` that accepts any member of the alphabet *except* its argument. Implement this.

- As well as single characters, we need a representation of the regular expression describing the empty language, `nil :: RegExp a`. Define `nil`.
- It is useful to have a regular expression that accepts any character, `any :: RegExp a`. Implement `any`.
- The regular expression combinators are sequential and optional composition. We would normally write `r.s` or `rs` to represent “an instance of `RegExp r` followed by an instance of `RegExp s`”; in Haskell we will call the operator `./.`, as both a single dot and juxtaposition have reserved meanings. Similarly, we would normally write `r | s` to represent “either an instance of `RegExp r` or an instance of `RegExp s`”; in Haskell we will call the operator `/|/`, as a vertical bar has a reserved meaning.

Complete the following definitions:

11. Extension: Lexer combinators

```
-- tell Haskell the syntax of two new infix, left-associative, operators:
infixl 5 /|/ -- less tightly binding
infixl 6 /./ -- more tightly binding
```

```
-- declare their types
(/|/), (/./) :: RegExp a -> RegExp a -> RegExp a
```

```
-- and define their meanings
```

```
(r /|/ s) t | isJust m = ?
              | otherwise = ?
  where split@(m, _) = r t
```

```
(r /./ s) t | ? = ?
              | otherwise = ?
  where (m, u) = r t
        (n, v) = s u
```

Note that choice is asymmetric, giving priority to its left-hand argument.

[Hint: The function **isJust** is in module **Data.Maybe**. Another useful function is **fromJust**.]



8. The final operator is iteration, usually written r^* . We will write **star** *r*. Complete the definition:

```
star :: RegExp a -> RegExp a
star r = ?
```

9. The above functions are enough, but it is useful to introduce a few more, all written in terms of the above.

Complete the following definitions, that mimic r^+ (**plus** *r*) and $[abcd]$ (**anyof** [*a*, *b*, *c*, *d*]):

```
plus :: Eq a => RegExp a -> RegExp a
plus r = ? -- Hint:
```

```
anyof :: Eq a => [a] -> RegExp a
anyof = ? -- Hint: use a fold
```

10. Define, using the RegExp combinators:

```
space, comment, whitespace :: RegExp Char
spacechar = ? -- a single space, newline or tab character
comment = ? -- any sequence inside curly braces "{...}"
whitespace = ? -- any non empty sequence of spaces characters and comments
```

11. Extension: Lexer combinators

```
digit , numeral, upper, lower, letter , variable :: RegExp Char
digit = anyof "0123456789"
upper = ? -- upper case letters
lower = ? -- lower case letters
open = symb '('
close = ?
numeral = ? -- a non-empty sequence of digits
letter = ? -- either upper or lower case
variable = ? -- start with a lower case letter , and then any
              -- combination of letters , digits and underscores
```

```
chopper :: String -> [String]
chopper = chop whitespace (open /|/ close /|/ numeral /|/ variable )
```

Try `chopper` on a variety of strings, including those with lexing errors (for example, containing the character `'&'`, or having an illegal character at the start of a variable).

□

Exercise 11.1 gave us tools to define regular expressions, and to use the regular expressions to chop a list of symbols into words; it did not give us the tools to turn those words into tokens. In the next exercise we will build a tool to do this.

Ex. 11.2* Our strategy is to create a function that turns words into tokens, and to map this function over the list of words obtained by lexing a list of symbols. Note that this is entirely orthogonal to splitting into words.

Introduce:

```
data Token = OPEN | CLOSE | DO | NUM Integer | VAR String deriving Show
```

Complete the following definitions:

```
string2token :: String -> Token
string2token "(" = ?
string2token ")" = ?
string2token "do" = DO -- a keyword
string2token s@(c:_) | ? = NUM (read s)
string2token s = ? -- anything else must be a variable
```

```
tokeniser :: String -> [Token]
tokeniser = map string2token . chopper
```

[Hint: The function `read` exists for any type in class `Read` — it is the inverse of `show`. How does Haskell know which version of `read` to use?]



Try `tokeniser "do_{abc_{comment}_{345})"`, as well as versions with illegal variable names and illegal characters.

□

12. Extension: Compiling for a graphical language

12.1. Introduction

This series of extension exercises is about taking a simple graphical description language and compiling it to turtle graphics.

We will start from a description of the abstract syntax, rather than the concrete syntax. These exercises assume a good knowledge of Haskell.

The exercises are closely based on exercises designed by Matt Naylor.

12.2. A special purpose language

In this series of sections you will write a compiler for a special purpose language to special purpose 'hardware': a black/white pixel language compiled into Turtle graphics.¹ We will use an Ascii character '#' to represent a black pixel, and a space, ' ', to represent a white pixel.

For these exercises you will work only with the abstract syntax; however, you are free to design a concrete syntax for each notation and implement a lexer/parser and pretty-printer using BNFC.

12.2.1. The high-level language

Abstract syntax

The code should be placed in a file called `Picture.hs` that starts with the line:

```
module Picture where
```

The abstract syntax used to describe the high-level language is:

```
data Picture = Horizontal Int Int Int  
             | Vertical  Int Int Int
```

¹I am grateful to Matt Naylor for inventing this series of exercises.

12. Extension: Compiling for a graphical language

| Overlay Picture Picture

- **Horizontal** $x\ y\ n$ is the picture consisting of a *horizontal* line of n pixels starting at coordinate (x, y) .
- **Vertical** $x\ y\ n$ is the picture consisting of a *vertical* line of n pixels starting at coordinate (x, y) .
- **Overlay** $p\ q$ (which may also be written $p\ \text{'Overlay'}\ q$) is the picture consisting of the black pixels of p and q .

The coordinate system runs West to East for x values and North to South for y values. Positive lengths run with the coordinate system, and negative lengths against it: for example “**Horizontal** 1 (−1) 5” is a line from $(1, -1)$ to $(6, -1)$; “**Vertical** 2 2 (−5)” is a line from $(2, 2)$ to $(2, -3)$. A 3-by-3 box may be drawn by:

| | |
|--|------------------|
| <code>box = Horizontal 0 0 3</code> | Output: |
| <code>'Overlay' Horizontal 2 2 (−3)</code> | <code>###</code> |
| <code>'Overlay' Vertical 0 0 3</code> | <code># #</code> |
| <code>'Overlay' Vertical 2 0 3</code> | <code>###</code> |

Ex. 12.1* Define a function `count :: Picture → Int` that counts the number of components in a picture. The expression `count box` should return 4. □

Semantics

The meaning of a picture is a “canvas”: a two-dimensional grid in which each cell is either black or white. We will model a canvas by a list of black cells.

```

type Coord = Int  -- to document how we are using Int
type CoordPair = (Coord, Coord)
type Canvas = [CoordPair]

```

A canvas representing a “+” symbol centred on $(0, 0)$ is:

| | |
|---|------------------|
| <code>plus :: Canvas</code> | <code>#</code> |
| <code>plus = [(0,0), (−1,0), (1,0), (0,−1), (0,1)]</code> | <code>###</code> |
| | <code>#</code> |

We will need the utility:

```

range :: Int → [Int]
range n | n < 0 = [n+1..0]
        | otherwise = [0..n−1]

```

12. Extension: Compiling for a graphical language

Ex. 12.2* What does `range` calculate?

□

Ex. 12.3*

1. Define `horizontal :: CoordPair → Int → Canvas` so that `horizontal (x, y) n` is the list of cells painted by the horizontal line of length `n` starting at coordinate `(x,y)`.
Try to do it using the pattern: `horizontal (x, y) n = [? | i <- range n]`.
2. Define a similar function `vertical :: CoordPair → Int → Canvas`.
3. Define a function `overlay :: Canvas → Canvas → Canvas` that combines two canvases by uniting them as sets.²
4. Define a function `draw :: Picture → Canvas` that describes the semantics of the picture.
Test it on `box`.

□

Rendering

It is easy to draw canvases on the screen, allowing them to be visualised. This is not necessary for compilation, but does make life easier!

Ex. 12.4*

1. Define a function `xbounds :: Canvas → (Coord, Coord)` that returns the x-coordinates of the left (most negative x-value) and right edge (most positive x-value) of the canvas.
2. Define a similar function `ybounds :: Canvas → (Coord, Coord)`.
3. Define a function `cell :: Canvas → CoordPair → Char`. `cell c (x,y)` returns `'#'` if canvas `c` is painted at coordinate `(x,y)`, and `' '` (space) otherwise.
4. Define a function `render :: Canvas → String` which renders each cell in the bounding box of the canvas, in raster order. Lines should be terminated by `'\n'`.

For example, `render plus` should produce:

```
"_#_\n###\n_#\n"
```

and `putStr (render plus)` should produce:

²Repeated values are not a problem, but you can eliminate them using functions in the standard prelude, or in `Data.List`. They can be found using `Hoogλe`.

12. Extension: Compiling for a graphical language

```
#  
###  
#
```

5. Define

```
mystery :: Picture  
mystery =      Vertical 0 0 5  
              'Overlay' Vertical 3 0 5  
              'Overlay' Horizontal 1 2 2  
              'Overlay' Vertical 6 0 5  
              'Overlay' Horizontal 5 0 3  
              'Overlay' Horizontal 5 4 3  
              'Overlay' Vertical 9 0 3  
              'Overlay' Vertical 9 4 1
```

What is the result of `putStr(render(draw pic))`?

□

12.2.2. A low-level language

In this section we will develop the semantics of the low-level language, a turtle-graphics machine.

The code for this section should go in the same directory as `Picture.hs`, in a file called `Turtle.hs` and begin:

```
module Turtle where
```

```
import Picture
```

Syntax and semantics

The turtle (as we shall call the machine) is a pen that faces a particular direction (North, South, East or West) and can be 'up' (do not make a mark) or 'down' (make a mark). It executes a list of commands:

```
data Command = Rotate  -- Rotate 90 degrees clockwise  
              | Forward -- Move one unit forward  
              | Lower   -- Lower the pen  
              | Raise   -- Raise the pen  
deriving (Eq, Show)
```

12. Extension: Compiling for a graphical language

A 3-by-3 box can be drawn by:

```
boxCommands :: [Command]
boxCommands = [Lower, Forward, Forward, Rotate, Forward, Forward,
               Rotate, Forward, Forward, Rotate, Forward, Raise]
```

Ex. 12.5 *

1. Define a type, **data** `Direction` = ... , to represent the direction a turtle may be facing.
2. Define a function `rotate :: Direction -> Direction` that calculates the result of a 90° clockwise rotation.
3. Coordinates in the turtle's world increase from West to East and from North to South. Define a function, `forward :: Direction -> CoordPair -> CoordPair`, that calculates the coordinates after moving one unit in the given direction.
4. Define a type, **data** `Pen` = ... , to represent if the pen is up or down.
5. Define a function, `paint :: Pen -> CoordPair -> Canvas -> Canvas`, which "paints" the cell at the given coordinates if the pen is lowered; otherwise it returns an unchanged canvas.
6. The state of the turtle and canvas is given by five components:
 - a) A list of commands to be executed;
 - b) the (x, y) coordinates of the turtle;
 - c) the direction the turtle is facing;
 - d) the pen's state (up or down); and
 - e) the canvas.

These can be represented by the type:

```
type State = ([Command], CoordPair, Direction, Pen, Canvas)
```

Define a function, `step :: State -> State`, that describes a structural operational (small step) semantics for the turtle. The function `step` takes a state in which the list of commands is non-empty, and executes the first command and returns an updated state.

7. Define a function, `run :: State -> Canvas` returns the canvas obtained by executing all the instructions in the state.
8. Define a function, `follow :: [Command] -> Canvas`, which is like run, but with an initial state that has:

12. Extension: Compiling for a graphical language

- initial coordinates (0, 0);
- facing east;
- pen raised; and
- a blank canvas.

Test your function with `putStr(render(follow boxCommands))`.

9. What is the result of

```
example :: [Command]
example =
  [Rotate, Lower, Forward, Forward, Forward, Forward, Raise,
   Rotate, Rotate, Forward, Forward, Rotate, Lower, Forward,
   Forward, Raise, Rotate, Forward, Forward, Rotate, Rotate,
   Lower, Forward, Forward, Forward, Forward, Raise, Rotate,
   Forward, Forward, Rotate, Lower, Forward, Forward, Forward,
   Forward, Raise, Rotate, Rotate, Rotate, Forward, Forward,
   Rotate, Rotate, Rotate, Lower, Raise, Forward, Forward,
   Lower, Forward, Forward ]
```

□

12.3. A compiler for a special purpose language

12.3.1. Preliminaries

The code for this section should go in a file called `Compile.hs` that begins:

```
module Compile where
```

```
import Picture
```

```
import Turtle
```

All files should be in the same directory.

Sequences of commands produced by the functions below may be required to satisfy various preconditions (assumption on initial state) and invariants (an aspect of the state that must be the same at the start and the end, although may change in the middle as long as it is restored):

Facing precondition The turtle is facing East at the start.

Direction invariant The turtle's direction at the end of the sequence is the same as that at the start.

12. Extension: Compiling for a graphical language

Pen invariant The turtle's pen is raised before and after the commands. (Note this includes the precondition of raised pen.)

Position invariant The turtle's position at the end of the sequence is the same as that at the start.

12.3.2. Compilation

This exercise is to produce a working compiler, not one that generates efficient code; that comes later.

Be warned: the functions for moving the turtle around ([advance](#), [move](#), [horizontal](#) and [vertical](#)) are simple in structure *but* it is very easy to get off-by-one errors in numerical arguments, and the case for zero wrong.

Ex. 12.6 *

1. Define a function `advance :: Int -> [Command]` such that `advance n` is a sequence of commands that causes the turtle to move forwards `n` units (if `n` is negative, then the movement is backwards) and also satisfies the direction invariant.
2. Define a function `move :: Int -> Int -> [Command]`, assuming the facing precondition, such that `move i j` is a sequence of commands that causes a turtle at (x, y) to move to $(x + i, y + j)$ and also satisfies the direction invariant.
3. Define a function `hor :: Int -> Int -> Int -> [Command]`, assuming the facing precondition and the pen starts raised, such that `hor i j n` is a sequence of commands that cause a turtle at (x, y) to draw a horizontal line of length `n` starting at $(x + i, y + j)$. The commands should satisfy the direction, pen and position invariants, and return the turtle to its starting position.
4. Define a function `ver :: Int -> Int -> Int -> [Command]`, analogous to `hor`, but which draws a vertical line.
5. Define a function `compile :: Picture -> [Command]` that compiles the given picture into a suitable list of turtle commands.
6. Test your compiler with `putStr (render (follow (compile mystery)))`.
7. Try your own pictures. Does your compiler do the correct thing with the smallest possible pictures?

□

12.4. Optimising a compiler for a special purpose language

12.4.1. Set up

The code for these exercises should live in a file called `Optimise.hs`. The first thing the code should do is to import `Picture`, `Turtle` and `Compile`.

Ex. 12.7* The following list of commands (among others) should be used to test your code:

```
testOpt :: [Command]
testOpt =
  [Rotate, Forward, Forward, Forward, Forward, Rotate, Rotate, Rotate, Lower, Forward,
    Forward, Forward, Forward, Rotate, Rotate, Rotate, Raise, Rotate, Rotate, Rotate,
    Forward, Forward, Forward, Forward, Rotate, Rotate, Rotate, Rotate, Rotate, Forward,
    Forward, Forward, Forward, Rotate, Rotate, Rotate, Rotate, Rotate, Forward, Rotate,
    Forward, Forward, Forward, Rotate, Rotate, Rotate, Raise, Lower, Forward, Forward,
    Raise, Rotate, Rotate, Forward, Forward, Forward, Rotate, Rotate, Rotate, Rotate,
    Rotate, Forward, Forward, Forward, Rotate, Rotate, Rotate, Rotate, Rotate, Forward,
    Forward, Rotate, Forward, Forward, Lower, Rotate, Rotate, Rotate, Lower]
```


Define the object in your file. What does it draw? ☐

A *peephole optimisation* is a program transformation that looks, through a “peephole”, at short sequences of commands and replaces them with more efficient sequences of commands.

Ex. 12.8*

1. Define a function `spinless :: [Command] -> [Command]` that deletes sequences of four consecutive rotate commands.

Try `length testOpt` and `length (spinless testOpt)`. What is the difference?

2. Define `equiv :: [Command] -> [Command] -> Bool` such that `cs 'equiv' ds == True` exactly when the commands `cs` and `ds` draw identical canvases. [Hint: You may find `nub` and `sort`, from the library `Data.List`, to be useful.] 

This function defines *semantic equivalence* of turtle programs.

What is the value of `testOpt 'equiv' (spinless testOpt)`?

☐

12.4.2. String rewriting

Consider:

```
cmdsOne :: [Command]
cmdsOne = [Rotate, Rotate, Raise, Rotate, Rotate, Forward]
```

Function `spinless` has no effect on `cmdsOne` as `cmdsOne` does not have four consecutive rotate commands. However, the rotate commands can be deleted. We can manipulate the sequence of commands into a form where `spinless` can be applied by a *rewrite rule* that moves pen raises earlier in the sequence:

`[..., Rotate, Raise, ...]` \rightsquigarrow `[..., Raise, Rotate, ...]`. The effect is to bring rotations together.

Ex. 12.9*

1. Define a function, `rewriteOnce :: [Command] -> [Command]`, that applies the rule exactly once if an opportunity to apply the rule exists, otherwise it just returns the original list.

The result of `rewriteOnce cmdsOne` should be

```
[Rotate, Raise, Rotate, Rotate, Rotate, Forward]
```

2. Define a function, `rewriteAll :: [Command] -> [Command]`, that repeatedly applies the rule until it cannot be applied any more. (This is another example of the calculation of a fixed point.)
3. Try

```
map (length . spinless) [cmdsOne, rewriteAll cmdsOne, testOpt, rewriteAll testOpt]
```

Does moving pen raises before rotates *enable* `spinless` in either case?

4. Swapping `Rotate` and `Raise` is not the only useful rewrite. Update `rewriteOnce` to include:

```
Lower, Rotate ~> Rotate, Lower
```

```
Lower, Lower ~> Lower
```

```
Raise, Raise ~> Raise
```

```
Lower, Raise ~> Raise
```

```
Raise, Lower ~> Lower
```

One of these new rewrites is **wrong** (that is, it does not preserve the semantics)!

Which? [Hint: try each rewritable sequence of length two.]

□

12.4.3. Conditional rewriting

Some rewrite rules can be applied only in restricted situations. For example, *only if the pen is raised*,

[..., Forward, Rotate, Forward, Rotate, Forward, ...]

is the same as

[..., Rotate, Forward, Rotate, ...]

This gives a *conditional rewrite rule*, CR1:

If

1. the pen is raised;
2. `cs :: [Command]` consists only of `Forward` and `Rotate` commands;
and
3. `cs` preserves the direction invariant (the direction at the end is the same as at the start)

then `[Forward, Rotate]++cs++[Rotate, Forward] ~→ [Rotate]++cs++[Rotate]`

Ex. 12.10 *

1. Execute CR1 by hand, as many times as possible on:

`cmdsTwo = [Forward, Forward, Rotate, Forward, Forward,
Rotate, Forward, Forward, Rotate]`

2. List `p` is a prefix of list `s` if there is some list `t` such that `s==p++t`.

Define a function `match :: [Command] → Bool` that returns `True` exactly when some prefix of its argument matches the precondition for CR1.

3. Define a function, `replace :: [Command] → [Command]`, which when applied to `cs :: [Command]` satisfying `match cs` applies CR1.
4. Define a function, `condrewriteOnce :: Pen → [Command] → [Command]`, that takes a (current) pen position and applies CR1 once to a list of commands, if possible. Note that your function will need to keep track of pen position.
5. Define a function, `condrewriteAll :: [Command] → Command`, that applies CR1 as many times as possible.

12. Extension: Compiling for a graphical language

6. Evaluate in GHCi:

- a) `length cmdsTwo` – `length (condrewriteAll cmdsTwo)`
- b) `length testOpt` – `length (condrewriteAll testOpt)`
- c) `length testOpt` – `length (condrewriteAll (spinless (rewriteAll testOpt)))`

□

12.4.4. An optimising compiler

Ex. 12.11 *

1. Define

```
ocompile :: Picture -> [Command]
ocompile = condrewriteAll . spinless . rewriteAll . compile
```

2. Evaluate `compile mystery 'equiv' ocompile mystery` and `length (compile mystery) – length (ocompile mystery)`.

□

Bibliography

- [1] A. Ranta, *Implementing Programming Languages*, ser. Texts in Computing 16. King's College, London, UK: College Publications, 2012.
- [2] E. W. Dijkstra, *A Discipline of Programming*. Englewood Cliffs, NJ: Prentice Hall, 1976.
- [3] G. Hutton, *Programming in Haskell*. Cambridge, UK: Cambridge University Press, 2007. [Online]. Available: <http://www.cs.nott.ac.uk/~gmh/book.html>.
- [4] — —, *Programming in Haskell*, second edition. Cambridge, UK: Cambridge University Press, 2016. [Online]. Available: <http://www.cs.nott.ac.uk/~gmh/book.html>.

A. The language M

A.1. Introduction

M is a small machine-level assembly language. It is named for **Matt Naylor**, who used to teach compilation in York.

A.2. Syntax

An M program is a non-empty sequence of commands. Each command is an optionally labelled instruction.

The machine is furnished with a collection of registers (lexical class “Reg”), a program counter (“ κ ”) and a memory (“ M ”).

For a label ℓ we write ‘ ℓ ’ to mean the line-number of ℓ ; similarly we use r' to mean the value stored in register r .

In the description of the branching (conditional jump) instructions, the notation $\kappa := \ell' \triangleleft g \triangleright \kappa' + 1$ means “assign ℓ' to κ if g , else assign $\kappa' + 1$ to κ ”.

Many instructions come in pairs: the ‘ordinary’ version that refers to a value in a register and the ‘immediate’ version that directly refers to an integer constant.

A.2.1. Lexical categories

Both registers and labels are represented by the lexical category “String”¹ The categories “String” and “Integer” are standard, and not given.

A.2.2. Syntactic categories other than instructions

$$\begin{array}{ll} M ::= [C]_{\text{;}}^+ & \text{Programs} \\ C ::= I \mid [\text{String} \text{ ”. ”}]_{\text{;}}^+ I & \text{Commands} \end{array}$$

¹We use “String”, rather than “Identifier” as we will require labels and registers names that are *not* identifiers. Such names will be generated by our compiler before being replaced by strings that *do* correspond to identifiers.

A. The language M

I represents the syntactic category of instructions, defined in subsection A.2.3.

[Note: it is conventional when writing machine code to place a newline after each instruction. This is hard to enforce in BNFC, but BNFC does not prevent it. The BNFC generated pretty-printer puts a new line after a semi-colon, so the above definition does achieve an approximation to the standard conventions.]

A.2.3. The syntactic category of instructions

[Presented accompanied by an informal semantics. In the context of an expression, v' means the value stored in register v , n (unprimed) means the numeric value itself.]

| | |
|------------------------------|---|
| $I ::=$ | |
| "nop" | null instruction |
| "hlt" | halt instruction |
| "prn" String | print instruction |
| "mov" String String | $\text{mov } r \ s \equiv r := s'$ |
| "movi" String Integer | $\text{movi } r \ n \equiv r := n$ |
| "add" String String String | $\text{add } r \ s \ t \equiv r := s' + t'$ |
| "addi" String String Integer | $\text{addi } r \ s \ n \equiv r := s' + n$ |
| "mul" String String String | $\text{mul } r \ s \ t \equiv r := s' * t'$ |
| "muli" String String Integer | $\text{muli } r \ s \ n \equiv r := s' * n$ |
| "neg" String String | $\text{neg } r \ s \equiv r := -s'$ |
| "negi" String Integer | $\text{negi } r \ n \equiv r := -n$ |

[Continued overleaf]

A. The language M

| | |
|---|---|
| "jmp" String | $\text{jmp } r \equiv \kappa := r'$ |
| "jmp _i " String | $\text{jmp}_i \ell \equiv \kappa := \ell'$ |
| "beq" String String String | $\text{beq } \ell \ r \ s \equiv \kappa := \ell' \triangleleft r' = s' \triangleright \kappa' + 1$ |
| "beq _i " String String Integer | $\text{beq}_i \ell \ r \ n \equiv \kappa := \ell' \triangleleft r' = n \triangleright \kappa' + 1$ |
| "bne" String String String | $\text{bne } \ell \ r \ s \equiv \kappa := \ell' \triangleleft r' \neq s' \triangleright \kappa' + 1$ |
| "bne _i " String String Integer | $\text{bne}_i \ell \ r \ n \equiv \kappa := \ell' \triangleleft r' \neq n \triangleright \kappa' + 1$ |
| "blt" String String String | $\text{blt } \ell \ r \ s \equiv \kappa := \ell' \triangleleft r' < s' \triangleright \kappa' + 1$ |
| "blt _i " String String Integer | $\text{blt}_i \ell \ r \ n \equiv \kappa := \ell' \triangleleft r' < n \triangleright \kappa' + 1$ |
| "bge" String String String | $\text{bge } \ell \ r \ s \equiv \kappa := \ell' \triangleleft r' \geq s' \triangleright \kappa' + 1$ |
| "bge _i " String String Integer | $\text{bge}_i \ell \ r \ n \equiv \kappa := \ell' \triangleleft r' \geq n \triangleright \kappa' + 1$ |
| "laddr" String String | $\text{laddr } r \ \ell \equiv r := \ell'$ |
| "load" String String | $\text{load } r \ a \equiv r := M(a')$ |
| "load _i " String Integer | $\text{load}_i r \ a \equiv r := M(n)$ |
| "store" String String | $\text{store } r \ n \equiv M(a') := r'$ |
| "store _i " String Integer | $\text{store}_i r \ n \equiv M(n) := r'$ |

A.3. Static semantics

Instructions that refer to a label (jmp, jmp_i, b..., laddr) always have the label as the last String parameter. There must be exactly one line marked with the label. All other string parameters refer to registers.

A.4. Dynamic semantics

The semantics is given as a small-step operational semantics, \longrightarrow , from which we derive a semantics \Rightarrow :

$$\text{MprogS} \frac{p \vdash (0, \emptyset, \emptyset, \epsilon) \longrightarrow^* (\infty, m, M, o)}{\vdash p \Rightarrow o}$$

Here \longrightarrow^* is the transitive closure of \longrightarrow .

The context is the program to be executed; the state consists of:

1. the program counter, ι , for which the special value ∞ represents no location,
2. the register map, m , a function from registers to Integers,

A. The language M

3. the memory map, M , a function from addresses to Integers,
4. the output buffer, o .

The definitions below use *functional override*. Given a function f , a domain element d and a range element r , $f[d \leftarrow r]$ is a function like f , except that it maps d to r . The evaluation rule is:

$$(f[d \leftarrow r])e = \begin{cases} r & \text{if } e = d \\ f e & \text{otherwise} \end{cases}$$

$$\text{MhltS} \frac{}{p \vdash (\iota, m, M, o) \longrightarrow (\infty, m, M, o)} [p\iota = \text{hlt}]$$

$$\text{MnopS} \frac{}{p \vdash (\iota, m, M, o) \longrightarrow (\iota + 1, m, M, o)} [p\iota = \text{nop}]$$

$$\text{MprnS} \frac{}{p \vdash (\iota, m, M, o) \longrightarrow (\iota + 1, m, M, o.(mr))} [p\iota = \text{prn } r]$$

$$\text{MmovS} \frac{}{p \vdash (\iota, m, M, o) \longrightarrow (\iota + 1, m[r \leftarrow ms], M, o)} [p\iota = \text{mov } r s]$$

$$\text{MmoviS} \frac{}{p \vdash (\iota, m, M, o) \longrightarrow (\iota + 1, m[r \leftarrow n], M, o)} [p\iota = \text{movi } r n]$$

$$\text{MaddS} \frac{}{p \vdash (\iota, m, M, o) \longrightarrow (\iota + 1, m[r \leftarrow ms + mt], M, o)} [p\iota = \text{add } r s t]$$

$$\text{MaddiS} \frac{}{p \vdash (\iota, m, M, o) \longrightarrow (\iota + 1, m[r \leftarrow ms + n], M, o)} [p\iota = \text{addi } r s n]$$

$$\text{MmulS} \frac{}{p \vdash (\iota, m, M, o) \longrightarrow (\iota + 1, m[r \leftarrow ms * mt], M, o)} [p\iota = \text{mul } r s t]$$

$$\text{Mmulis} \frac{}{p \vdash (\iota, m, M, o) \longrightarrow (\iota + 1, m[r \leftarrow ms * n], M, o)} [p\iota = \text{muli } r s n]$$

$$\text{MnegS} \frac{}{p \vdash (\iota, m, M, o) \longrightarrow (\iota + 1, m[r \leftarrow -1 * ms], M, o)} [p\iota = \text{neg } r s]$$

$$\text{MnegiS} \frac{}{p \vdash (\iota, m, M, o) \longrightarrow (\iota + 1, m[r \leftarrow -n], M, o)} [p\iota = \text{negi } r n]$$

$$\text{MjmpS} \frac{}{p \vdash (\iota, m, M, o) \longrightarrow (mr, m, M, o)} [p\iota = \text{jmp } r]$$

$$\text{Mjmpis} \frac{}{p \vdash (\iota, m, M, o) \longrightarrow (\ell, m, M, o)} [p\iota = \text{jmpis } \ell]$$

A. The language M

$$\begin{array}{l}
\text{MbeqST} \frac{}{p \vdash (\iota, m, M, o) \longrightarrow (\ell, m, M, o)} [p \iota = \text{beq } \ell \ s \ t, m \ s = m \ t] \\
\text{MbeqSF} \frac{}{p \vdash (\iota, m, M, o) \longrightarrow (\iota + 1, m, M, o)} [p \iota = \text{beq } \ell \ s \ t, m \ s \neq m \ t] \\
\text{MbeqiST} \frac{}{p \vdash (\iota, m, M, o) \longrightarrow (\ell, m, M, o)} [p \iota = \text{beqi } \ell \ s \ n, m \ s = n] \\
\text{MbeqiSF} \frac{}{p \vdash (\iota, m, M, o) \longrightarrow (\iota + 1, m, M, o)} [p \iota = \text{beqi } \ell \ s \ t, m \ s \neq n] \\
\text{MbneST} \frac{}{p \vdash (\iota, m, M, o) \longrightarrow (\ell, m, M, o)} [p \iota = \text{bne } \ell \ s \ t, m \ s \neq m \ t] \\
\text{MbneSF} \frac{}{p \vdash (\iota, m, M, o) \longrightarrow (\iota + 1, m, M, o)} [p \iota = \text{bne } \ell \ s \ t, m \ s = m \ t] \\
\text{MbneiST} \frac{}{p \vdash (\iota, m, M, o) \longrightarrow (\ell, m, M, o)} [p \iota = \text{bnei } \ell \ s \ n, m \ s \neq n] \\
\text{MbneiSF} \frac{}{p \vdash (\iota, m, M, o) \longrightarrow (\iota + 1, m, M, o)} [p \iota = \text{bnei } \ell \ s \ t, m \ s = n] \\
\text{MbltST} \frac{}{p \vdash (\iota, m, M, o) \longrightarrow (\ell, m, M, o)} [p \iota = \text{blt } \ell \ s \ t, m \ s < m \ t] \\
\text{MbltSF} \frac{}{p \vdash (\iota, m, M, o) \longrightarrow (\iota + 1, m, M, o)} [p \iota = \text{blt } \ell \ s \ t, m \ s \geq m \ t] \\
\text{MbltiST} \frac{}{p \vdash (\iota, m, M, o) \longrightarrow (\ell, m, M, o)} [p \iota = \text{blti } \ell \ s \ n, m \ s < n] \\
\text{MbltiSF} \frac{}{p \vdash (\iota, m, M, o) \longrightarrow (\iota + 1, m, M, o)} [p \iota = \text{blti } \ell \ s \ t, m \ s \geq n] \\
\text{MbgeST} \frac{}{p \vdash (\iota, m, M, o) \longrightarrow (\ell, m, M, o)} [p \iota = \text{bge } \ell \ s \ t, m \ s \geq m \ t] \\
\text{MbgeSF} \frac{}{p \vdash (\iota, m, M, o) \longrightarrow (\iota + 1, m, M, o)} [p \iota = \text{bge } \ell \ s \ t, m \ s < m \ t] \\
\text{MbgeiST} \frac{}{p \vdash (\iota, m, M, o) \longrightarrow (\ell, m, M, o)} [p \iota = \text{bgei } \ell \ s \ n, m \ s \geq n] \\
\text{MbgeiSF} \frac{}{p \vdash (\iota, m, M, o) \longrightarrow (\iota + 1, m, M, o)} [p \iota = \text{bgei } \ell \ s \ t, m \ s < n] \\
\text{MladdrS} \frac{}{p \vdash (\iota, m, M, o) \longrightarrow (\iota + 1, m[r \leftarrow \ell], M, o)} [p \iota = \text{laddr } r \ \ell] \\
\text{MloadS} \frac{}{p \vdash (\iota, m, M, o) \longrightarrow (\iota + 1, m[r \leftarrow M(m \ a)], M, o)} [p \iota = \text{load } r \ a] \\
\text{MloadiS} \frac{}{p \vdash (\iota, m, M, o) \longrightarrow (\iota + 1, m[r \leftarrow M \ a], M, o)} [p \iota = \text{loadi } r \ a] \\
\text{MstoreS} \frac{}{p \vdash (\iota, m, M, o) \longrightarrow (\iota + 1, m, M[m \ a \leftarrow m \ r], o)} [p \iota = \text{store } r \ a] \\
\text{MstoreiS} \frac{}{p \vdash (\iota, m, M, o) \longrightarrow (\iota + 1, m, M[a \leftarrow m \ r], o)} [p \iota = \text{storei } r \ a]
\end{array}$$

B. The language N

B.1. Introduction

N is a toy imperative language. It is named for Matt Naylor, who used to teach compilation at the University of York.

B.2. Syntax

An N program is a single statement, described by syntactic class S .
The lexical classes “Identifier” and “Integer” are standard, and omitted.

B. The language N

| | |
|---|-----------------------|
| $V ::= \text{Identifier} ":" T$ | Variable declaration |
| $T ::= \text{"Int"} \mid \text{"Bool"}$ | Type names |
| $P ::= \text{Identifier} ":" \{ "[V]^*," \mid "[V]^*," S \}$ | Procedure declaration |
| $D ::= V \mid P$ | Declaration |
| $S ::=$ | Statements |
| "skip" | Null statement |
| $\mid \text{"print"} E$ | Print statement |
| $\mid \text{Identifier} ":=" E$ | Assignment |
| $\mid \text{"if"} E \text{"then"} S \text{"else"} S \text{"end"}$ | Choice |
| $\mid \text{"while"} E \text{"do"} S \text{"end"}$ | Iteration |
| $\mid \{ " D ";" S " \}$ | Declaration block |
| $\mid \text{Identifier} \{ "[E]^*," \mid "[\text{Identifier}]^*," \}$ | Procedure call |
| $\mid [S]^+_{","}$ | Sequencing |
| $E ::=$ | Expressions |
| $\text{Integer} \mid \text{"neg"} E \mid E "*" E \mid E "+" E$ | Integer expressions |
| $\mid \text{"true"} \mid \text{"false"} \mid \text{"not"} E \mid E \text{"or"} E$ | Boolean expressions |
| $\mid E \text{"and"} E \mid E = E \mid E < E$ | |
| $\mid \text{Identifier}$ | Variables |

All lists (for example, sequencing of statements) are right-associative. A declaration block introduces a single new variable or procedure.

The binding power of the expression operators decreases from left to right (for example, "*" binds more tightly than "+" and "not" binds more tightly than "and"). The relational operators ("=" and "<") bind less tightly than any arithmetic operator. All operators are right-associative.

B.3. Static semantics

B.3.1. Miscellaneous

At any scope level, names may only be declared once. Names may be re-declared in a sub-scope (that is, in a sub-block).

The number of parameters in a procedure call, in each category, must agree with the definition.

B. The language N

B.3.2. Typing

N programs must be well-typed. A statement is well-typed or not; if statement s is well-typed we write “ $\lfloor s \rfloor$ *valid*”. An expression is well typed or not, if it is well-typed then it has a type; we write “ $e : t$ ” if expression e has type t .

If ℓ is a list of expressions, and d is a list of variable declarations, then $\ell : d$ mean that ℓ and d have the same length, and the type of the k th element of ℓ has the same type as the k th declaration in d .

A context for a type judgement is a set of variable and procedure types.

Programs

A program is a statement evaluated in an empty context.

$$\text{NprogT} \frac{\emptyset \vdash \lfloor s \rfloor \text{ valid}}{\vdash \lfloor s \rfloor \text{ valid}}$$

Procedure definitions

Note how the rule ‘NproclistT’ enforces definition before use.

In the rule ‘NprocT’ the local variables replace any occurrences in the context.

$$\text{NproclistT} \frac{\Gamma \vdash \lfloor p \rfloor \text{ valid} \quad \Gamma, p \vdash \lfloor q \rfloor \text{ valid}}{\Gamma \vdash \lfloor p; q \rfloor \text{ valid}}$$

$$\text{NprocT} \frac{(\Gamma \setminus i, o), i, o \vdash \lfloor s \rfloor \text{ valid}}{\Gamma \vdash \lfloor n : \{i; o; s\} \rfloor \text{ valid}}$$

Statements

Note how rule ‘NassT’ fixes the type of the expression to be the type of the variable to which its value is assigned.

$$\text{NskipT} \frac{}{\Gamma \vdash \lfloor \text{skip} \rfloor \text{ valid}}$$

$$\text{NprintT} \frac{\Gamma \vdash e : \text{Int}}{\Gamma \vdash \lfloor \text{print } e \rfloor \text{ valid}}$$

$$\text{NassT} \frac{\Gamma \vdash e : T}{\Gamma \vdash \lfloor v := e \rfloor \text{ valid}} [v : T \text{ in } \Gamma]$$

B. The language N

$$\text{NifT} \frac{\Gamma \vdash g : \text{Bool} \quad \Gamma \vdash [s] \text{ valid} \quad \Gamma \vdash [t] \text{ valid}}{\Gamma \vdash [\text{if } g \text{ then } s \text{ else } t \text{ end}] \text{ valid}}$$

$$\text{NdoT} \frac{\Gamma \vdash g : \text{Bool} \quad \Gamma \vdash [s] \text{ valid}}{\Gamma \vdash [\text{while } g \text{ do } s \text{ end}] \text{ valid}}$$

$$\text{NseqT} \frac{\Gamma \vdash [s] \text{ valid} \quad \Gamma \vdash [t] \text{ valid}}{\Gamma \vdash [s; t] \text{ valid}}$$

$$\text{NvarblockT} \frac{(\Gamma \setminus v), v : T \vdash [s] \text{ valid}}{\Gamma \vdash [\{v : T; s\}] \text{ valid}}$$

$$\text{NprocblockT} \frac{(\Gamma \setminus p, i, o), p : \{i \mid o\}, i, o \vdash [t] \text{ valid} \quad (\Gamma \setminus p), p : \{i \mid o\} \vdash [s] \text{ valid}}{\Gamma \vdash [\{p : \{i \mid o \mid t\}; s\}] \text{ valid}}$$

Note how this rule demands that the body of p is type-checked in the context of its parameters and the procedure signature (to allow for recursive procedures). The main statement is only type-checked in the context of the procedure signature, as the local variables of the procedure are hidden from it.

$$\text{NcallT} \frac{\Gamma \vdash i : f \quad \Gamma \vdash o : g}{\Gamma \vdash [n\{i \mid o\}] \text{ valid}} [n : \{f \mid g\} \text{ in } \Gamma]$$

Expressions

$$\text{NintT} \frac{}{\Gamma \vdash n : \text{Int}} [n \text{ is a lexical Integer}]$$

$$\text{NnegT} \frac{\Gamma \vdash e : \text{Int}}{\Gamma \vdash \text{neg } e : \text{Int}}$$

$$\text{N*T} \frac{\Gamma \vdash e : \text{Int} \quad \Gamma \vdash f : \text{Int}}{\Gamma \vdash e * f : \text{Int}}$$

$$\text{N+T} \frac{\Gamma \vdash e : \text{Int} \quad \Gamma \vdash f : \text{Int}}{\Gamma \vdash e + f : \text{Int}}$$

$$\text{NtrueT} \frac{}{\Gamma \vdash \text{true} : \text{Bool}}$$

$$\text{NfalseT} \frac{}{\Gamma \vdash \text{false} : \text{Bool}}$$

$$\text{NnotT} \frac{\Gamma \vdash e : \text{Bool}}{\Gamma \vdash \text{not } e : \text{Bool}}$$

B. The language N

$$\text{NandT} \frac{\Gamma \vdash e : \text{Bool} \quad \Gamma \vdash f : \text{Bool}}{\Gamma \vdash e \text{ and } f : \text{Bool}}$$

$$\text{NorT} \frac{\Gamma \vdash e : \text{Bool} \quad \Gamma \vdash f : \text{Bool}}{\Gamma \vdash e \text{ or } f : \text{Bool}}$$

$$\text{N=T} \frac{\Gamma \vdash e : \text{Int} \quad \Gamma \vdash f : \text{Int}}{\Gamma \vdash e = f : \text{Bool}}$$

$$\text{N<T} \frac{\Gamma \vdash e : \text{Int} \quad \Gamma \vdash f : \text{Int}}{\Gamma \vdash e < f : \text{Bool}}$$

The rule for program variables has a type meta-variable, T .

$$\text{NvarT} \frac{}{\Gamma \vdash v : T} [v : T \text{ in } \Gamma]$$

B.3.3. Expression simplification

Expressions

Expressions in N obey the usual rules.

Conjunction ('and') is associative and commutative with zero 'false' and unit 'true'.

Disjunction ('or') is associative and commutative with zero 'true' and unit 'false'.

Logical negation ('not') is an involution with 'not true = false' and

'not false = true'.

Addition ('+') is associative and commutative with unit 0. It has no zero. It obeys the usual laws for simplifying constants (for example, ' $3 + 4 = 7$ ',

' $3 + \text{neg } 4 = \text{neg } 1$ ').

Multiplication ('*') is associative and commutative with unit 1 and zero 0. It obeys the usual laws for simplifying constants (for example, ' $3 * 4 = 12$ ',

' $3 * \text{neg } 4 = \text{neg } 12$ ').

Arithmetic negation ('neg') is an involution with fixed point 0.

Equality ('=') is commutative and reflexive.

B. The language N

Statements

Constant guards in choices can be simplified:

$$\begin{aligned}\text{if true then } s \text{ else } t \text{ end} &= s \\ \text{if false then } s \text{ else } t \text{ end} &= t\end{aligned}$$

False guards in iterations can be simplified:

$$\text{while false do } s \text{ end} = \text{skip}$$

Reflexive assignments do no computation:

$$x := x = \text{skip}$$

Any statement with components can be simplified by simplifying the components, where appropriate.

[There are many other laws for N which are not worth worrying about at this stage; for example,

$$\begin{aligned}(x := e; x := f) &= x := f && \text{if } x \text{ does not appear in } f \\ (x := e; y := f) &= (x := e; y := f') && \text{where } f' = f \text{ with every occurrence of } x \text{ replaced by } e\end{aligned}$$

]

B.4. Dynamic semantics

B.4.1. Notation

The semantics is given as a big-step operational semantics. The transition relation for statements is written \Downarrow , and relates program/state pairs to state/output pairs. States are modelled as functions from variable names to values.

We need the concepts of:

The empty string of outputs ϵ .

Concatenation of output strings Given output strings a and b , we represent their concatenation by $a.b$.

The empty state modelled by the empty function, \emptyset .

B. The language N

An evaluation function The meaning of expression e in state t is written $t[e]$, which is defined:

$$\begin{aligned}
 t[n] &= n & n \text{ an Int constant} \\
 t[\text{neg } a] &= -1 * t[a] \\
 t[a * b] &= t[a] * t[b] \\
 t[a + b] &= t[a] + t[b] \\
 t[\text{true}] &= \top \\
 t[\text{false}] &= \perp \\
 t[\text{not } a] &= \neg t[a] \\
 t[a \text{ and } b] &= t[a] \wedge t[b] \\
 t[a \text{ or } b] &= t[a] \vee t[b] \\
 t[a = b] &= t[a] = t[b] \\
 t[a < b] &= t[a] < t[b] \\
 t[v] &= t v & v \text{ an identifier} \\
 t[v, w] &= t[v], t[w]
 \end{aligned}$$

The last equation is for a syntax not available in the programming language, but necessary to explain the semantics: it describes evaluating a list of expressions.

Functional override and restriction Given a function f , a domain element d and a range element r , $f[d \leftarrow r]$ is a function like f , except that it maps d to r . The evaluation rule is:

$$(f[d \leftarrow r]) e = \begin{cases} r & \text{if } e = d \\ f e & \text{otherwise} \end{cases}$$

Given two functions, f and g , we write $f \downarrow g$ to mean a function like f restricted to g 's domain of definition.

$$(f \downarrow g) x = \begin{cases} f x & \text{if } g x \text{ is defined} \\ \text{undefined} & \text{otherwise} \end{cases}$$

Context extension We use $\Gamma \oplus d$ to mean a context like Γ but with any references to names declared in d over-ridden by the declarations in d .

B. The language N

B.4.2. Programs

The proof rule for entire programs is captured by the relation $\Downarrow\Downarrow$ which relates a complete program to its output:

$$\text{NprogD} \frac{\emptyset \vdash (s, \emptyset) \Downarrow (t, o)}{\vdash s \Downarrow\Downarrow o}$$

B.4.3. Statements

$$\text{NskipD} \frac{}{\Gamma \vdash (\mathbf{skip}, t) \Downarrow (t, \epsilon)}$$

$$\text{NprintD} \frac{}{\Gamma \vdash (\mathbf{print} \ e, t) \Downarrow (t, t[e])}$$

$$\text{NassD} \frac{}{\Gamma \vdash (v := e, t) \Downarrow (t[v \leftarrow t[e]], \epsilon)}$$

$$\text{NifTD} \frac{\Gamma \vdash (s, t) \Downarrow (u, o)}{\Gamma \vdash (\mathbf{if} \ g \ \mathbf{then} \ s \ \mathbf{else} \ r \ \mathbf{end}, t) \Downarrow (u, o)} [t[g]]$$

$$\text{NifFD} \frac{\Gamma \vdash (r, t) \Downarrow (u, o)}{\Gamma \vdash (\mathbf{if} \ g \ \mathbf{then} \ s \ \mathbf{else} \ r \ \mathbf{end}, t) \Downarrow (u, o)} [\neg t[g]]$$

$$\text{NdoTD} \frac{\Gamma \vdash (s, t) \Downarrow (m, a) \quad \Gamma \vdash (\mathbf{while} \ g \ \mathbf{do} \ s \ \mathbf{end}, m) \Downarrow (u, b)}{\Gamma \vdash (\mathbf{while} \ g \ \mathbf{do} \ s \ \mathbf{end}, t) \Downarrow (u, a.b)} [t[g]]$$

$$\text{NdoFD} \frac{}{\Gamma \vdash (\mathbf{while} \ g \ \mathbf{do} \ s \ \mathbf{end}, t) \Downarrow (t, \epsilon)} [\neg t[g]]$$

$$\text{Nvarblock0D} \frac{\Gamma, v \vdash (s, t[v \leftarrow \epsilon]) \Downarrow (u, o)}{\Gamma \vdash (\{v : T; s\}, t) \Downarrow (u \setminus v, o)} [v \text{ in } \Gamma]$$

$$\text{Nvarblock1D} \frac{\Gamma \vdash (s, t[v \leftarrow \epsilon]) \Downarrow (u, o)}{\Gamma \vdash (\{v : T; s\}, t) \Downarrow (u[v \leftarrow s \ v], o)} [v \text{ in } \Gamma]$$

$$\text{Nvarblock2D} \frac{\Gamma \setminus v, v \vdash (s, t[v \leftarrow \epsilon]) \Downarrow (u, o)}{\Gamma \vdash (\{v : T; s\}, t) \Downarrow (u \setminus v, o)} [v : \{i \mid o \mid x\} \text{ in } \Gamma]$$

B. The language N

A variable declaration block is treated differently depending on if the variable does not already exist ('Nvarblock0D'); if it exists as a variable ('Nvarblock1D'); and if it exists as a procedure. If it does not exist, it must be added to the context and to the state with a null value (' ϵ '); when the block finishes the variable must be removed from the state (' $u \setminus v$ '). If the name exists as a variable, the context does not change but the variable still needs to be set to null; when the block finishes the previous value of the variable (' $s \ v$ ') must be restored. If the name existed as a procedure it is replaced in the context by the name as a variable, and its value set to null; when the block finishes the variable must be removed from the state.

$$\text{Nprocblock0D} \frac{\Gamma \setminus p, p : \{i \mid o \mid x\} \vdash (s, t) \Downarrow (u, o)}{\Gamma \vdash (\{p : \{i \mid o \mid x\}; s\}, t) \Downarrow (u, o)} [p \notin \Gamma]$$

$$\text{Nprocblock1D} \frac{\Gamma \setminus p, p : \{i \mid o \mid x\} \vdash (s, t) \Downarrow (u, o)}{\Gamma \vdash (\{p : \{i \mid o \mid x\}; s\}, t) \Downarrow (u[p \leftarrow s], o)} [p \in \Gamma]$$

There are two cases for a procedure declaration: there is no variable of the same name, but perhaps a procedure of the same name ('Nprocblock0D') and there is a variable of the same name, and so not a procedure of the same name ('Nprocblock1D'). If the variable does not exist, the context has any old version of the procedure removed and the new one added. If the variable does exist, it is replaced in the context by the procedure, and the variable's value restored at the end of the block.

$$\text{NcallD} \frac{\Gamma \oplus (f, w) \vdash (s, t[f \leftarrow e]) \Downarrow (u, o)}{\Gamma \vdash (p\{e \mid v\}, t) \Downarrow (u[v \leftarrow u\llbracket w \rrbracket] \Downarrow t, o)} [p : \{f \mid w \mid s\} \text{ in } \Gamma]$$

$$\text{NseqD} \frac{\Gamma \vdash (r, t) \Downarrow (m, a) \quad \Gamma \vdash (s, m) \Downarrow (u, b)}{\Gamma \vdash (r; s, t) \Downarrow (u, a.b)}$$

C. Rules for compiling N to M

C.1. Rules for an unbounded register machine

C.1.1. Assumptions

We assume:

1. A source code value fits in a register.
2. The M machine has as many registers as needed to compile the N program of interest, without needing external memory.
3. Registers names are the same as used in the source code.
4. An unbounded number of names (of registers and labels) are available.

Assumptions 2 and 3 are dealt with in a final optimisation phase.

Some rules need to allocate registers and labels that are unused. We do this by declaring the names of the registers and labels to be *fresh*. That is, the names are unused anywhere in the compilation process so far.

M code fragments will often be several lines of instructions. We group these together by drawing a box around them.

C.1.2. Rules

We define the following relations by Natural Deduction rules:

$s \mapsto \boxed{c}$ N *program* s compiles to M program c .

$s \rightsquigarrow \boxed{c}$ N *statement* s compiles to M code c .

$e \hookrightarrow \left(\boxed{c}, r \right)$ N expression e compiles to M code c , leaving the value of e in register r .

$(g, \ell) \rightarrow c$ If g evaluates to true, code c jumps to label ℓ and otherwise falls through to the next instruction.

C. Rules for compiling N to M

$(g, \ell) \not\vdash c$ If g evaluates to false, code c jumps to label ℓ and otherwise falls through to the next instruction.

While most of the relations we deal with are actually functions, those above are not functional: in some cases we have alternative right-hand-sides for the same left-hand-sides. An implementation will deal with this by choosing the rule that leads to the most efficient code.

The context of a judgement is the collection of variables and procedures in scope. Most rules are independent of the context, and we use Γ to represent it; where rules are dependent on the context we will write Γ as the pair (B, Π) of variables (B^1) and procedures (Π) in scope.

Programs

Programs always end with a halt instruction.

$$\text{NMprog} \frac{\Gamma \vdash s \rightsquigarrow \boxed{c}}{\Gamma \vdash s \mapsto \boxed{\begin{smallmatrix} c \\ \text{hlt} \end{smallmatrix}}}$$

Statements

Simple statements The N skip statement compiles to an empty list of M instructions.

$$\text{NMskip} \frac{}{\Gamma \vdash \mathbf{skip} \mapsto \boxed{\quad}}$$

The N print statement compiles to an expression evaluation and assignment, followed by printing the register.

$$\text{NMprint} \frac{\Gamma \vdash t := e \rightsquigarrow \boxed{c}}{\Gamma \vdash \mathbf{print} \ e \rightsquigarrow \boxed{\begin{smallmatrix} c \\ \text{prn } t \end{smallmatrix}}} [t \text{ fresh}]$$

The N sequential composition statement compiles to concatenation of M lists of instructions.

$$\text{NMseq} \frac{\Gamma \vdash s \rightsquigarrow \boxed{c} \quad \Gamma \vdash t \rightsquigarrow \boxed{d}}{\Gamma \vdash s; t \rightsquigarrow \boxed{\begin{smallmatrix} c \\ d \end{smallmatrix}}}$$

¹The Greek letter 'B' is pronounced like a 'V' in standard English usage.

C. Rules for compiling N to M

Note that the front end generates a representation as a list of sequential statements. An implementation based on this rule must use it in that context; a fold would be a suitable implementation structure.

Assignment of Int expressions

Constant assignment

$$\text{NMassConst} \frac{}{\Gamma \vdash x := n \rightsquigarrow \boxed{\text{movi } x \ n}} [n \text{ an integer constant}]$$

$$\text{NMassVar} \frac{}{\Gamma \vdash x := v \rightsquigarrow \boxed{\text{mov } x \ v}} [v \text{ a variable name}]$$

Simplistic rules

$$\text{NMassPlus} \frac{\Gamma \vdash t := e \rightsquigarrow \boxed{c} \quad \Gamma \vdash u := f \rightsquigarrow \boxed{d}}{\Gamma \vdash x := e + f \rightsquigarrow \boxed{\begin{array}{c} c \\ d \\ \text{add } x \ t \ u \end{array}}} [t \text{ fresh}, u \text{ fresh}]$$

$$\text{NMassMul} \frac{\Gamma \vdash t := e \rightsquigarrow \boxed{c} \quad \Gamma \vdash u := f \rightsquigarrow \boxed{d}}{\Gamma \vdash x := e * f \rightsquigarrow \boxed{\begin{array}{c} c \\ d \\ \text{mul } x \ t \ u \end{array}}} [t \text{ fresh}, u \text{ fresh}]$$

$$\text{NMassNeg} \frac{\Gamma \vdash t := e \rightsquigarrow \boxed{c}}{\Gamma \vdash x := \text{neg } e \rightsquigarrow \boxed{\begin{array}{c} c \\ \text{neg } x \ t \end{array}}} [t \text{ fresh}]$$

Sophisticated rules First we give rules for \hookrightarrow :

$$\text{ExpVar} \frac{}{\Gamma \vdash v \hookrightarrow (\boxed{}, v)} [v \text{ a variable name}]$$

$$\text{ExpNonVar} \frac{\Gamma \vdash t := e \rightsquigarrow \boxed{c}}{\Gamma \vdash e \hookrightarrow (\boxed{c}, t)} [e \text{ not a variable name}, t \text{ fresh}]$$

Next we give the rules for compilation:

C. Rules for compiling N to M

$$\text{NMass}+ \frac{\Gamma \vdash e \hookrightarrow (\boxed{c}, t) \quad \Gamma \vdash f \hookrightarrow (\boxed{d}, u)}{\Gamma \vdash x := e + f \rightsquigarrow \boxed{\begin{array}{l} c \\ d \\ \text{add } x \ t \ u \end{array}}}$$

$$\text{NMass}* \frac{\Gamma \vdash e \hookrightarrow (\boxed{c}, t) \quad \Gamma \vdash f \hookrightarrow (\boxed{d}, u)}{x := e * f \rightsquigarrow \boxed{\begin{array}{l} c \\ d \\ \text{mul } x \ t \ u \end{array}}}$$

$$\text{NMass}- \frac{\Gamma \vdash e \hookrightarrow (\boxed{c}, t)}{\Gamma \vdash x := \text{neg } e \rightsquigarrow \boxed{\begin{array}{l} c \\ \text{neg } x \ t \end{array}}}$$

Improved rules for expressions with constants

$$\text{NMAss}+n \frac{\Gamma \vdash e \hookrightarrow (\boxed{c}, v)}{\Gamma \vdash x := e + n \rightsquigarrow \boxed{\begin{array}{l} c \\ \text{addi } x \ v \ n \end{array}}} \quad [n \text{ an Int constant}]$$

$$\text{NMAss}*n \frac{\Gamma \vdash e \hookrightarrow (\boxed{c}, v)}{\Gamma \vdash x := e * n \rightsquigarrow \boxed{\begin{array}{l} c \\ \text{muli } x \ v \ n \end{array}}} \quad [n \text{ an Int constant}]$$

Assignment of Bool expressions See exercises.

Guarded statements

Guards There are two strategies to compile guards:

1. compile the guard expression to a value, then jump based on that value;
2. compile the guard to a jump.

The rules below use Strategy 2.

$$\text{NMbrF} \frac{}{\Gamma \vdash (\text{false}, \ell) \rightarrow \boxed{\quad}}$$

C. Rules for compiling N to M

$$\text{NMbrT} \frac{}{\Gamma \vdash (\mathbf{true}, \ell) \rightarrow \boxed{\text{jmp } \ell}}$$

$$\text{MNbrNot} \frac{\Gamma \vdash (g, \ell) \not\rightarrow \boxed{c}}{\Gamma \vdash (\mathbf{not } g, \ell) \rightarrow \boxed{c}}$$

The rule ‘NMbrVar’ assumes an encoding for Boolean values where 0 represents \perp (false) and anything else represents \top (true).

$$\text{NMbrVar} \frac{}{\Gamma \vdash (v, l) \rightarrow \boxed{\text{bnei } v \ 0 \ l}}$$

The rules ‘NMbrOr’ and ‘NMbrAnd’ produce short-circuiting code.

$$\text{NMbrOr} \frac{\Gamma \vdash (g, \ell) \rightarrow \boxed{c} \quad \Gamma \vdash (h, \ell) \rightarrow \boxed{d}}{\Gamma \vdash (g \mathbf{or } h, \ell) \rightarrow \boxed{\begin{smallmatrix} c \\ d \end{smallmatrix}}}$$

$$\text{NMbrAnd} \frac{\Gamma \vdash (g, \text{end}) \not\rightarrow \boxed{c} \quad \Gamma \vdash (h, \ell) \rightarrow \boxed{d}}{\Gamma \vdash (g \mathbf{and } h, \ell) \rightarrow \boxed{\begin{smallmatrix} c \\ d \\ \text{end : nop} \end{smallmatrix}}} \text{[end fresh]}$$

$$\text{NMbr<} \frac{\Gamma \vdash a \hookrightarrow (\boxed{c}, v) \quad \Gamma \vdash b \hookrightarrow (\boxed{d}, w)}{\Gamma \vdash (a < b, \ell) \rightarrow \boxed{\begin{smallmatrix} c \\ d \\ \text{blt } \ell \ v \ w \end{smallmatrix}}}$$

The rules for $\not\rightarrow$ are dual to the rules for \rightarrow . See the exercises.

Choice statement

$$\text{NMif} \frac{\Gamma \vdash (g, \text{els}) \not\rightarrow \boxed{c} \quad \Gamma \vdash p \rightsquigarrow \boxed{d} \quad \Gamma \vdash q \rightsquigarrow \boxed{e}}{\Gamma \vdash \mathbf{if } g \mathbf{ then } p \mathbf{ else } q \mathbf{ end } \rightsquigarrow \boxed{\begin{smallmatrix} c \\ d \\ \text{jmp } \text{end} \\ \text{els : nop} \\ e \\ \text{end : nop} \end{smallmatrix}}} \text{[els fresh, end fresh]}$$

C. Rules for compiling N to M

Loop statement There are two strategies: top-test ('NMLoopT') and bottom-test ('NMLoopB').

$$\begin{array}{c}
 \text{NMLoopT} \frac{\Gamma \vdash (g, \text{end}) \not\sim \boxed{c} \quad \Gamma \vdash p \rightsquigarrow \boxed{d}}{\Gamma \vdash \text{while } g \text{ do } p \text{ end} \rightsquigarrow \boxed{\begin{array}{l} \text{top : nop} \\ c \\ d \\ \text{jmp } \text{top} \\ \text{end : nop} \end{array}}} [top \text{ fresh}, end \text{ fresh}] \\
 \\
 \text{NMLoopB} \frac{\Gamma \vdash p \rightsquigarrow \boxed{c} \quad \Gamma \vdash (g, \text{top}) \rightarrow \boxed{d}}{\Gamma \vdash \text{while } g \text{ do } p \text{ end} \rightsquigarrow \boxed{\begin{array}{l} \text{jmp } \text{tst} \\ \text{top : nop} \\ c \\ \text{tst : nop} \\ d \end{array}}} [top \text{ fresh}, tst \text{ fresh}]
 \end{array}$$

Blocks and procedures First we define two pseudo instructions to push and pop values on a stack that lives at the top of memory, and grows downwards. The address of the top of the stack is in register s . We assume a register is w words of memory.

$$\begin{array}{l}
 \text{push } r = \boxed{\begin{array}{l} \text{sub } s \ s \ w \\ \text{store } r \ s \end{array}} \\
 \text{pop } r = \boxed{\begin{array}{l} \text{load } r \ s \\ \text{add } s \ s \ w \end{array}}
 \end{array}$$

N declaration blocks declare a single variable or procedure. This makes the rules simpler.

$$\begin{array}{c}
 \text{NMBlockN} \frac{(B, \Pi) \vdash p \rightsquigarrow \boxed{c}}{(B, \Pi) \vdash \{v : T; p\} \rightsquigarrow \boxed{\begin{array}{l} \text{push } v \\ c \\ \text{pop } v \end{array}}} [v \text{ in } B] \\
 \\
 \text{NMBlockO} \frac{(B \cup \{v : T\}, \Pi) \vdash p \rightsquigarrow \boxed{c}}{(B, \Pi) \vdash \{v : T; p\} \rightsquigarrow \boxed{c}} [v \text{ in } B]
 \end{array}$$

C. Rules for compiling N to M

As a simplification the rule 'NMproc' assumes that all procedure names are unique (easily achieved by systematically changing all procedure names to fresh ones).

$$\text{NMProc} \frac{(B \cup i \cup o, \Pi) \vdash s \rightsquigarrow \boxed{c}}{(B, \Pi) \vdash \{p\{i \mid o \mid s\}; t\} \rightsquigarrow \boxed{\begin{array}{l} p : \text{nop} \\ c \\ \text{pop } r \\ \text{jmp } r \end{array}}} [r \text{ fresh}]$$

Note the code generated for procedures is placed *after* the 'hlt' instruction generated to go after the main program.

A possibly-recursive procedure call is compiled by the rule:

$$\text{NMcall} \frac{\mathbf{v} \equiv \mathbf{e} \rightsquigarrow \boxed{c} \quad \mathbf{r} \equiv \mathbf{w} \rightsquigarrow \boxed{d}}{(B, \Pi) \vdash p(\mathbf{e})(\mathbf{r}) \rightsquigarrow \boxed{\begin{array}{l} \text{PUSH } (\mathbf{v} \cup \mathbf{w}) \setminus \mathbf{r} \\ c \\ \text{ladr } t \ a \\ \text{push } t \\ \text{jmp } p \\ a : \text{nop} \\ d \\ \text{POP } (\mathbf{v} \cup \mathbf{w}) \setminus \mathbf{r} \end{array}}} [p(\mathbf{v})(\mathbf{w}) \text{ in } \Pi, t \text{ fresh}, a \text{ fresh}]$$

with pseudo-instructions:

$$\begin{array}{ll} \text{PUSH } w = \boxed{\text{push } w} & \text{PUSH } (w, \mathbf{v}) = \boxed{\begin{array}{l} \text{push } w \\ \text{PUSH } \mathbf{v} \end{array}} \\ \text{POP } w = \boxed{\text{pop } w} & \text{POP } (w, \mathbf{v}) = \boxed{\begin{array}{l} \text{POP } \mathbf{v} \\ \text{pop } w \end{array}} \end{array}$$

D. Optimisation

D.1. Constant propagation

D.1.1. Purpose

An arithmetic M instruction, such as “add $x\ y\ z$ ”, can be replaced by the more efficient immediate version “addi $x\ y\ n$ ” if it is known that register z holds the value n . In practice, we know this if an assignment “movi $z\ n$ ” is the only assignment to z *reaching in* to the instruction.

Following constant propagation it is sometimes possible to eliminate instructions (section D.2).

D.1.2. Algorithm

The *reach-in set* of an instruction is the set of line numbers of instructions whose assignments affect the instruction.

1. Compute the *reach-in sets* for each instruction.
2. If exactly one instruction assigning to z is in the reach-in set, and has the form “movi $z\ n$ ”, then instruction “use $x\ y\ z$ ” may be replaced by its immediate version.

Reach-in sets are computed using the (reverse) *control flow graph*, the *generation (gen) sets* and the *kill sets*.

Control flow graph a directed graph whose nodes are the lines of the M program.

An arc goes from node x to node y exactly when control can pass from instruction x to instruction y . Given a node there is a set of (immediate) successor and predecessor nodes.

Gen set For line number, ℓ :

$$\text{gen } \ell = \begin{cases} \{\ell\} & \text{if } \ell \text{ assigns} \\ \{\} & \text{otherwise} \end{cases}$$

D. Optimisation

Kill set For line number, ℓ :

$$\text{kill } \ell = \begin{cases} \{ m \mid m \neq \ell, m \text{ assigns to the same register as } \ell \} & \text{if } \ell \in \text{gen } \ell \\ \{\} & \text{otherwise} \end{cases}$$

Reach-in sets The reach-in sets are the smallest sets that satisfy:

$$\text{reachIn } \ell = \bigcup_{m \in \text{predecessor } \ell} (\text{gen } m \cup (\text{reachIn } m \setminus \text{kill } m))$$

They may be calculated by repeatedly applying the function \mathcal{R} to the initial approximation, 'null' ($\text{null } \ell = \{\}$) until a fixed-point is found. The function \mathcal{R} is defined:

$$\mathcal{R} f \ell = \bigcup_{m \in \text{predecessor } \ell} (\text{gen } m \cup (g m \setminus \text{kill } m))$$

where

$$g = \begin{cases} \mathcal{R} f & \text{if } m < \ell \\ f & \text{otherwise} \end{cases}$$

[Note: a correct, but less efficient, version may be obtained by using $g = f$.]

D.2. Dead-code elimination

D.2.1. Purpose

An M immediate load to register instruction, such as “movi x 0” can be replaced by ‘nop’ if its value of x is never used, or, in the jargon, not *live-out*. This increases the speed of execution. Later on peephole optimisation can be used to eliminate the ‘nop’ instructions, reducing the size of the code and speeding up its execution still further.

Assignments whose values are live-out can become dead following constant propagation (section D.1).

D.2.2. Algorithm

The *live-out* set of an instruction is the set of registers assigned to by the instruction that might be used later in the computation.

D. Optimisation

1. Compute the live-out sets for each instruction.
2. Any immediate assignment whose assigned variable is not in its live-out set may be replaced by a no-op instruction.

Live out sets are computed using the *control flow graph*, the *use sets* and the *definition (def) sets*.

Control flow graph See section D.1.

Use sets For line number, ℓ : $\text{use } \ell = \{ r \mid r \text{ is read at } \ell \}$

def sets For line number, ℓ : $\text{def } \ell = \{ r \mid r \text{ is assigned to at } \ell \}$

Live-out sets The live out sets are the smallest sets that satisfy:

$$\text{liveOut } \ell = \bigcup_{m \in \text{sucessor } \ell} (\text{use } m \cup (\text{liveOut } m \setminus \text{def } m))$$

They may be calculated by repeatedly applying the function L to the initial approximation, 'null' ($\text{null } \ell = \{\}$) until a fixed-point is found. The function \mathcal{L} is defined:

$$\mathcal{L} f \ell = \bigcup_{m \in \text{sucessor } \ell} (\text{use } m \cup (g m \setminus \text{def } m))$$

where

$$g = \begin{cases} \mathcal{L} f & \text{if } \ell < m \\ f & \text{otherwise} \end{cases}$$