Go! reference manual

Version 1.1f

Francis G. M^{c}Cabe

Network Agent Press
March 29, 2008

Go! Reference Manual

ISBN 0-9754449-3-X

Network Agent Press
Contact frankmccabe@mac.com

For Miko and Stephen, without whom, none of this makes sense

Preface

Go! is a logic programming language that is oriented to the needs of secure production quality Internet based distributed applications. It is object-oriented, multi-threaded and strongly typed. It supports multi-paradigm programming styles – with different notations for functions, predicates and action procedures.

This manual offers an in-depth and complete description of the Go! language. However, it is not a tutorial on the language, and the order of presentation within the manual reflects that.

Contents

List of Programs

List of Figures

List of Examples

List of Tables

Part I

Syntax and semantics of Go! programs

Language Syntax 1

We take a layered approach to understanding Go!'s syntax. The lowest level is the character level; then there is the lexical or token level, the parse level and finally the well typed formulae level. Each of these levels identifies a different 'level of abstraction'; each focusses on a different aspect of a well formed Go! program.

This chapter is concerned with the lower three of these abstractions: characters, tokens and parse trees. This progressively moves up from the physical character of files containing Go! programs to a tree-like representation of the contents of a Go! source file. In later chapters, we focus on the subset of parse trees that can be given a well defined meaning: i.e., are well typed and form coherent units of execution.

1.1 Characters and lexical syntax

1.1.1 Unicode characters

Go! uses the Unicode character encoding system [**?**]. More specifically, the Go! run-time uses Unicode characters internally to represent symbols and the Go! compiler and run-time engine interpret streams of UTF-8 bytes and UTF-16 words as Unicode. Identifiers in Go! programs as well as data processed by Go! programs are assumed to be Unicode based.

A Go! language processor is free to determine the encoding used in a given source text file. However, if the first two bytes in the file are `\+feff;` or `\+fffe;` then the processor must interpret the source text as being encoded using UTF-16, or byte-swapped UTF-16 respectively. In addition, if no external indication of the

character encoding is supplied, then the language processor **must** default to the UTF-8 encoding scheme.

Although Go! is a Unicode-based language, all of the the characters used within the language definition of Go! – such as the built-in operators and syntactic features – are contained within the ASCII subset of the Unicode character set. Thus, Go! can be used in an ASCII-based environment.

Character Categories

The Unicode consortium has identified a number of *character catagories*. A character category distinguishes the typical role of a character – numeric digit, punctuation mark, regular text – in a language independent way.

Go! uses these character categories to classify characters for the the purposes of discriminating identifiers and number values occurring in the program source text. The purpose of this is to permit non-English programmers to use non-English identifiers in the text of the program.

The character categories used by Go! are:

Table 1.1: Unicode character categories

Cat	Description	Go! predicate	Page
Cc	Other, Control	`__isCcChar`	181
Cf	Other, Format	`__isCfChar`	181
Cn	Other, Not assigned	`__isCnChar`	182
Co	Other, Private	`__isCoChar`	182
Cs	Other, Surrogate	`__isCsChar`	182
Ll	Letter, Lowercase	`__isLlChar`	183
Lm	Letter, Modifier	`__isLmChar`	184
Lo	Letter, Other	`__isLoChar`	184
Lt	Letter, Titlecase	`__isLtChar`	183
Lu	Letter, Uppercase	`__isLuChar`	183
Mc	Mark, Spacing Combining	`__isMcChar`	185
Me	Mark, Enclosing	`__isMeChar`	185
Mn	Mark, Non spacing	`__isMnChar`	184

continued...

Table 1.1 Unicode character categories (cont.)

Cat	Description	Go! predicate	Page
Nd	Number, Numerical digit	`__isNdChar`	185
Nl	Number, Letter	`__isNlChar`	186
No	Number, Other	`__isNoChar`	186
Pc	Punctuation, Connector	`__isPcChar`	188
Pd	Punctuation, Dash	`__isPdChar`	188
Pe	Punctuation, Close	`__isPeChar`	189
Pf	Punctuation, Final Quote	`__isPfChar`	189
Pi	Punctuation, Initial Quote	`__isPiChar`	189
Po	Punctuation, Other	`__isPoChar`	190
Ps	Punctuation, Open	`__isPsChar`	190
Sc	Symbol, Currency	`__isScChar`	187
Sk	Symbol, Modifier	`__isSkChar`	187
Sm	Symbol, Math	`__isSmChar`	187
So	Symbol, Other	`__isSoChar`	188
Zl	Separator, Line	`__isZlChar`	190
Zp	Separator, Paragraph	`__isZpChar`	191
Zs	Separator, Space	`__isZsChar`	191

1.2 Tokens

It is required that the input of a Go! language processor is first partitioned into contiguous sequences of characters called *tokens*. There are several different kinds of tokens; corresponding to the identifiers, symbols, character literals, string literals and punctuation marks necessary to correctly parse a Go! program.

We use Go!'s own grammar notation to describe the tokenization and parsing rules for Go!. Apart from demonstrating the power of Go!'s grammar notation – which is modelled on `Prolog`'s DCG grammars – it also demonstrates that we can eat our own dog food!

In the productions for the tokenizer listed below we assume that, in an actual tokenizer, all the productions for a given non-terminal are collected together – as is required by Go!'s grammar.

However, for explanatory purposes, the grammar rules for some of the non-terminals are distributed thoughout the text below. The tokenizer rules depend on the `token` type defined below:

```
tokType ::= ID(symbol)                -- Identifier
  | IN(integer)                       -- Integer literal
  | FT(float)                         -- Floating point literal
  | ST(string)                        -- String literal
  | SY(string)                        -- Symbol
  | CH(char)                          -- Character literal
  | LPAR | RPAR | LBRA | RBRA
  | LBRCE | RBRCE                     -- Punctuation
  | COMMA | CONS | TERM
  | EOF.                              -- End of file
```

This type definition defines the legal kinds of tokens that can be expected; and also forms the basis of the stream processed at the next higher level in the Go! language specification: namely the syntactic grammar.

1.2.1 Comments and whitespace

Tokens in a Go! source text may separated by zero or more *comments* and/or white space text – some pairs of tokens *require* some intervening space or comments for proper recognition. For example, a number following an identifier requires at least one white space character; otherwise the rules for identifier would 'swallow' the number token. White space characters and comments may be used for the purposes of recognizing tokens; but must otherwise be discarded by a language processor.

There are two styles of comment in Go! source texts – *line* comments and *block* comments.

Line comment

A line comment consists of the characters `--␣` i.e., two hyphen characters and a whitespace character, followed by all the characters up to the next new-line character or end-of-file which ever is first.

Block comment

A block comment consists of the characters `/*` followed by any characters and is terminated by the characters `*/`

The Go! tokeniser rules for white space and comments are:

```
whiteSpace:[char]{}.
whiteSpace(X):-__isZsChar(X).
whiteSpace(X):-__isZlChar(X).
whiteSpace(X):-__isZpChar(X).

comments:[integer+,integer-]-->string.
comments(Lno,Lx) --> comment(Lno,Ly)!, comments(Ly,Lx).
comments(Lno,Lno) --> "".

comment:[integer+,integer-]-->string.
comment(Lno,Lx) --> "-- ",lineComment(Lno,Lx).
comment(Lno,Lx) --> "--\t",lineComment(Lno,Lx).
comment(Lno,Lx) --> "/*",bodyComment(Lno,Lx).
comment(Lno,Lno+1) --> "\n".
comment(Lno,Lno) --> [X],{__isZsChar(X)}. -- ignore space

lineComment:[integer+,integer-]-->string.
lineComment(Lno,Lno+1) --> [X],{__isZlChar(X)}. -- new line
lineComment(Lno,Lx) --> [X],{\+__isZlChar(X)},
        lineComment(Lno,Lx).

bodyComment:[integer+,integer-]-->string.
bodyComment(Lno,Lno) --> "*/".
bodyComment(Lno,Lx) --> [X],{__isZlChar(X)},
        bodyComment(Lno+1,Lx).
bodyComment(Lno,Lx) --> [X],bodyComment(Lno,Lx).
```

1.2.2 Identifiers

Identifiers serve many purposes within a Go! program: to identify variables and parameters, to identify types, even to identify syntactic operators.

Go! identifiers are modelled on the standard Unicode identifier syntax; which in turn is a generalization of the common notation for identifier syntax found in many programming languages.

The basic rule for identifiers is that they have an initial start letter – which is either a letter character, or a character which could be used in a word or an underscore character – followed by a sequence of zero or more characters from a somewhat extended set – including the digit characters.

What makes this style of identifier 'different' is that the Unicode standard defines many thousands of 'letter' character; this allows identifiers to be written in non roman scripts – such as Kanji for example.

The following Go! grammar rules capture the definition of an identifier:

```
tok:[tokType-]-->string. -- a grammar that returns a token

/* An identifier is an idStart character,
   followed by a sequence of idChar */
tok(ID(I)) --> idStart(X), idChar(C)*C^N, I=implode([X,..N]).
...
idStart:[char-]-->string.
idStart(X) --> [X],{__isLetterChar(X)}.
idStart('_) --> "_".
idStart('\+ff3f;) --> "\+ff3f;". -- full width low line

idChar:[char-]-->string.
idChar(X) --> idStart(X).
idChar(X) --> [X],{__isNdChar(X)}.
idChar(X) --> [X],{__isMnChar(X)}.
idChar(X) --> [X],{__isMcChar(X)}.
idChar(X) --> [X],{__isPcChar(X)}.
idChar(X) --> [X],{__isCfChar(X)}.
...
```

The `idStart` rule defines those characters that may start an identifier, and the `idChar` rule identifies those characters that may

continue an identifier. Together, these rules state that any 'letter' character may start an identifier, and letters and numbers may follow this initial letter.

The grammar condition

```
idChar(C)*C^N
```

matches a sequence of zero or more `idChar`s, and puts the resulting list in the variable `N`. See Section 6.2.4 on page 114 for a more complete explanation.

1.2.3 Characters

An individual character literal value is written as a back-tick character ‘ followed by a *string character*. A string character consists of any character except new-line, paragraph separator, ’, or the double quote character "; or the \ character followed by a *character* reference.

For example, the new-line character is written:

```
‘\n
```

The grammar rule for character literals is:

```
tok(CH(ch)) --> "\‘", strChar(ch).
```

where `strChar` is the production to defines a legal character that may occur in a `char` value, or a quoted `symbol` or a string.

Character reference

Character references are used in several places in Go!'s syntax: within string literals, symbols and character literals. There are also several special forms of character reference. The common Unix names for characters such as `\n` for new-line are recognized, as is a special notation for entering arbitrary Unicode characters.

There are roughly three categories of character references: characters which do not need escaping, characters that are represented

using a backslash escape, and characters denoted by their hexadecimal character code.

For example the string `"string"` is equivalent to the list:

```
[‘s,‘t,‘r,‘i,‘n,‘g]
```

A string containing just a new-line character is:

```
"\n"
```

and a string containing the Unicode sentinel character would be denoted:

```
"\+fffe;"
```

The rules for character references are:

```
/* Special character sequence following a back-slash
*  in a symbol or literal string
*/
strChar:[char-]-->string.
strChar(‘\a) --> "\\a".
strChar(‘\b) --> "\\b".
strChar(‘\d) --> "\\d".
strChar(‘\e) --> "\\e".
strChar(‘\f) --> "\\t".
strChar(‘\n) --> "\\n".
strChar(‘\r) --> "\\r".
strChar(‘\t) --> "\\t".
strChar(‘\v) --> "\\v".
strChar(‘\’) --> "\\’".
strChar(‘\") --> "\\"".
strChar(‘\‘) --> "\\‘".
strChar(‘\\) --> "\\\\".
strChar(Cr) --> "\\+",hexSeq(0,C),";",Cr=__charOf(C).
strChar(X) --> "\\", [X].
strChar(X) --> [X],{\+__isCcChar(X)}.
...
hexDig:[integer-]-->string.
hexDig(__digitCode(X)) --> [X],{__isNdChar(X)}.
```

```
hexDig(10) --> "a".  hexDig(10) --> "A".
hexDig(11) --> "b".  hexDig(11) --> "B".
hexDig(12) --> "c".  hexDig(12) --> "C".
hexDig(13) --> "d".  hexDig(13) --> "D".
hexDig(14) --> "e".  hexDig(14) --> "E".
hexDig(15) --> "f".  hexDig(15) --> "F".

hexSeq:[integer+,integer-]-->string.
hexSeq(I,N) --> hexDig(X)!,hexSeq(N*16+X,N).
hexSeq(N,N) --> "".
```

The rules for `strChar` above may be a little confusing, because of the rules for quoting characters in strings. The required sequence of characters to represent a new-line character is a back-slash followed by an `n`: `"\n"`. The string that denotes this in the grammar rule requires two backslashes – since a backslash in a string must itself be represented by two backslash characters. This is especially spectacular in the case of the `strChar` rule for the backslash character: this requires two backslashes in sequence and the string that denotes this sequence in the grammar rule consists of four backslash characters!

1.2.4 Symbols

Symbols are written as a sequence of characters – not including new-line or other control characters – surrounded by ' marks. More specifically, symbols are written as a sequence of string characters (see Section 1.2.3 above) surrounded by ' characters.

For example,

```
'a symbol'
```

is a `symbol`, as is

```
'#$'
```

The grammar rule for `symbol` tokens is

```
...
tok(SY(L)) --> "\'", strChar(C)*C^L,"\'".
...
```

1.2.5 String literals

String literals are written as a sequence of string characters (see 1.2.3) – not including new-line or paragraph separator characters – surrounded by " marks.

The reason for not permitting new-lines to occur in string literals is that that enables a particularly silly kind of syntax error to be picked up easily: a missing string quote will always generate a syntax error at the end of the line. The restriction does not affect the possible string literals, as it is always possible to use `\n` to indicate a new-line character, and the Go! compiler concatenates sequences of string literals into a single string literal.

One benefit of the automatic string merge feature is that when a program has a lengthy string embedded in it it can be formatted both for display and for program tidiness.

The grammar rule for string literals is very similar to the production for symbols. Note however, that string literals are interpreted as synonyms for lists of `char`s.

```
tok(ST(L)) --> "\"", strChar(C)*C^L,"\"".
```

Numbers

Go! numbers are built from the `__isNdChar` character class. This class of characters includes many digit characters; all of which share the semantic property that they can be interpreted as decimal digits.

Go! distinguishes integer literals (and values) from floating point literals and values. Due to the sometimes complex rules for sub-typing, these are not generally substitutable for each other.

```
tok(Nm) --> [X],{__isNdChar(X)},
    numberSeq(__digitCode(X),I),
    ( fraction(F),exponent(E),
      Nm = FT(n2float(I)+F)*n2float(E)
    | Nm = IN(I)).

tok(IN(N)) --> "0x", hexSeq(0,N).
tok(IN(C)) --> "0c", strChar(C), { C = __charCode(c)}.
...
numberSeq:[integer+,integer-]-->string.
numberSeq(I,N) --> [D],{__isNdChar(D)!},
        numberSeq(I*10+__digitCode(D),N).
numberSeq(N,N) --> "".

fraction:[float-]-->string.
fraction(F) --> ".", [C],{__isNdChar(C)},
        frSeq(10,1/__digitCode(C),F).
fraction(0) --> "".

frSeq:[float+,float+,float-]-->string.
frSeq(X,F,R) --> [C],{__isNdChar(C)},
        frSeq(X*10,F+__digitCode(C)/X,R).
frSeq(X,F,F) --> "".

exponent:[float-]-->string.
exponent(Ex) --> "E-", numberSeq(0,X),Ex = 10 ** (-X).
exponent(Ex) --> "e-", numberSeq(0,X),Ex=10 ** (-X).
exponent(Ex) --> "E", numberSeq(0,X),Ex=10 ** X.
exponent(Ex) --> "e", numberSeq(0,X),Ex=10 ** X.
exponent(1) --> "".
```

Apart from the normal decimal notation for numbers, Go! supports two additional notations: the hexadecimal notation and the character code notation. The hexadecimal number notation simply consists of a leading `0x` followed by the hexadecimal digits of the number. All hexadecimal numbers are integral.

The character code notation is used to construct numbers that correspond to particular characters. A sequence of characters of the form:

`0c`*<StrChar>*

denotes not a `char`acter literal but the Unicode value corresponding to that character. For example, the sequence

```
0c\n
```

is an `integer` literal – valued as 10 because the new-line character has a Unicode value of 10.

Notice that there is no definition of a negative numeric literal as a single token. Literal negative numbers are handled at a slightly higher level in the Go! language processing: the leading `-` character is interpreted as a prefix unary operator signifying arithmetic negation.

1.2.6 Operators and punctuation marks

Go! uses a number of special punctuation marks to signify specific syntactic features – such as lists, theta expressions and so on.

```
tok(LPAR) --> "(".
tok(RPAR) --> ")".
tok(LBRA) --> "[".
tok(RBRA) --> "]".
tok(LBRCE) --> "{".
tok(RBRCE) --> "}".
tok(CONS) --> ",..".
tok(COMMA) --> ",".
tok(ID('.␣')) --> ".", [X],{ __isZsChar(X)
                           | __isZlChar(X)
                           | __isZsChar(X)
                           | __isCcChar(X)}.
```

Note that the `.␣` symbol consists of a period followed by any kind of white space character. This distinguishes it from other uses of

the period; such as class body definition operator `..` or within a floating point number.

In addition to these punctuation marks, Go! uses a number of symbols for the prefined operators:

```
tok(ID('||') --> "||".
tok(ID('|') --> "|".
tok(ID('::') --> "::".
tok(ID('::=') --> "::=".
tok(ID(':=') --> ":=".
tok(ID(':-') --> ":-".
tok(ID(':--') --> ":--".
tok(ID(':') --> ":".
tok(ID('-->') --> "-->".
tok(ID('->') --> "->".
tok(ID('~') --> "~".
tok(ID('<~') --> "<~".
tok(ID('^') --> "^".
tok(ID('..') --> "..".
tok(ID('.')) --> ".".
tok(ID('?') --> "?".
tok(ID('!') --> "!".
tok(ID('==') --> "==".
tok(ID('=') --> "=".
tok(ID('\\=') --> "\\=".
tok(ID('!=') --> "!=".
tok(ID('$') --> "$".
tok(ID('\\+') --> "\\+".
tok(ID('+') --> "+".
tok(ID('-+')) --> "-+".
tok(ID('++')) --> "++".
tok(ID('*>') --> "*>".
tok(ID('**') --> "**".
tok(ID('*') --> "*".
tok(ID('/') --> "/".
tok(ID('\\/') --> "\\/".
tok(ID('/\\') --> "/\\".
tok(ID('\\') --> "\\".
```

```
tok(ID('<>') --> "<>".
tok(ID('=>') --> "=>".
tok(ID('<=') --> "<=".
tok(ID('>=') --> ">=".
tok(ID('=<') --> "=<".
tok(ID('.=') --> ".=".
tok(ID('%%') --> "%%".
tok(ID('#') --> "#".
tok(ID('@') --> "@".
tok(ID('@=') --> "@=".
tok(ID('@>') --> "@>".
tok(ID('@@') --> "@@".
tok(ID('>') --> ">".
tok(ID('<') --> "<".
tok(ID('-') --> "-".
tok(ID(';') --> ";".
```

Note that all the graphic identifiers are 'spelled out' here. This is indicative of one of the more subtle difference between Go! and `Prolog`. The reason that we can do so is that Go! does not permit the addition of so-called user-defined operators; and hence we know already all the operators in the language. In turn, explicitly enumerating all the possible graphical tokens significantly improves the programmer's experience of writing Go! programs – it is not necessary to separate sequences of characters such as:

```
A*-B
```

as the Go! parser can correctly parse this as:

```
*(A,-(B))
```

without requiring an explicit space between the `*` and `-` characters.

Any character not referred to explicitly here, and not referred to in any of the token rules above is not considered a legal character in a Go! program.

1.2.7 A Standard Tokenizer

Based on the productions for the `tok` introduces above, we can define a complete Go! tokenizer that consumes a string (i.e., a list

of characters) and produces a list of tokens.

Note that (as used in a compiler) a realistic tokenizer would collect additional information as it tokenizes the stream; in particular, the line number where each token occurs.

Our tokenizer rules can be easily extended to count line numbers and associate a line number and source file with each token. We use an auxilliary constructor to wrap each token with this information:

```
TokenType ::= tk(tokType,number).
```

The production below for `goTokens` also show how the treatment of white space and comments meshes with the treatment of the individual types of token:

```
goTokens:[list[(tokType,integer)-]]-->string.
goTokens([tk(Tok,Ln),..L],Lno) -->
    comment(Lno,Ln), tok(Tok), goTokens(L,Ln).
goTokens([],Lno) --> comment(Lno,_).
```

This requires a slight modification to the rules for comment handling, and furthermore assumes (correctly) that a Go! token cannot span across multiple lines of input.

1.3 Operator Grammar

The grammar of Go! specifies the rules for sequencing tokens into syntactically reasonable structures. However, a grammar parser is not itself capable of determining *semantically* meaningful program structures – that analysis requires the type inferencing system and other phases of a language processor.

The grammar of Go! is based on an *operator precedence grammar*. Although they might not be aware of it, most programmers are quite familiar with operator precedence grammars – it is typically the grammar used for arithmetic expressions in regular programming languages. In Go! – as in `Prolog` – we extend the use of operator-style grammars to cover the whole language.

An operator grammar allows us to write expressions like:

```
X * Y + X / Y
```

and to know that this means the equivalent of:

```
(X * Y) + (X / Y)
```

or more specifically:

```
+(*(X, Y), /(X, Y))
```

I.e., an operator grammar allows us to write operators between arguments instead of before them, and an operator precedence grammar allows us to avoid using parentheses in many common situations.

It is not necessary to restrict the scope of an operator grammar to arithmetic expressions – a fact used by the early originators of the **POP-2** and `Prolog` programming languages. We can also use an operator grammar for the whole of a programming language. For example, in Go!, an equation such as:

```
app([E,..X],Y) => [E,..app(X,Y)]
```

can be interpreted – by treating `=>` as an operator – as:

```
=>(app([E,..X],Y),[E,..app(X,Y)])
```

A somewhat more complicated example allows us to interpret a conditional equation in terms of operators:

```
sort([E,..X])::split(E,X,A,B) => sort(A)<>[E]<>sort(B)
```

as

```
=>(::(sort([E,..X]),split(E,X,A,B)),:-(<>(sort(A),<>([E],sort(B))))
```

Here, we have relied on the fact that `=>`, `<>` and `::` are *infix* operators, in addition to the 'normal' infix operators: `+` and `-` operators.

The major benefit of this sleight of hand is simplicity – it allows us to focus on the logical structure of a program fragment rather than the inessential details.

The major demerit is that, occasionally, syntax errors can be somewhat harder to interpret. Furthermore, a syntax error at this level tends to create a cascade of spurious error messages as the parser attempts to recover from the incorrect input.

The output of parsing a Go! text using the operator precedence grammar is a *parse tree*. This parse tree represents the syntactic structure of the program text in an abstract way.

1.3.1 Standard operators

The standard operators in Go! are listed in order of priority below in Section 1.2. Each operator has a priority, associativity and a role.

The priority of an operator is the indication of the 'importance' of the operator: the higher the priority the nearer the top of the abstract syntax tree the corresponding structure will be. Priorities are numbers in the range 1..2000; by convention, priorities in the range 1..899 refer to entities that normally take the role of expressions, priorities in the range 900..1000 refer to predicates and predicate-level connectives and priorities in the range 1001..2000 refer to entries that have a statement or program level interpretation. The comma and the semi-colon operators are the only one with a priority of exactly 1000.

Table 1.2: Go! standard operators

Operator	Priority	Assoc.	Description
`.␣`	1900	right	statement separator
`::=`	1460	infix	user type definition
`|`	1250	right	type union
`?`	1200	infix	conditional operator
`=>`	1200	infix	function arrow

continued...

Table 1.2 Go! standard operators (cont.)

Operator	Priority	Assoc.	Description
`:-`	1200	infix	clause arrow
`:--`	1200	infix	strong clause
`->`	1200	infix	process rule
`-->`	1200	infix	grammar rule
`*>`	1152	infix	all solutions
`;`	1150	right	action separator
`::`	1125	left	guard marker
`||`	1060	infix	bag of constructor
`,`	1000	right	tupling, conjunction
`onerror`	955	infix	error handler
`<=`	950	infix	class rule arrow
`<~`	949	infix	implements interface
`@`	905	infix	tau pattern notation
`@@`	905	right	suspension variable
`timeout`	900	infix	timeout clause
`=`	900	infix	variable declaration
`:=`	900	infix	variable assignment
`==`	900	infix	equality predicate
`\=`	900	infix	not unifyable
`!=`	900	infix	not equal
`<`	900	infix	less than
`=<`	900	infix	less than or equal
`>`	900	infix	greater than
`>=`	900	infix	greater than or equal
`.=`	900	infix	match predicate
`=.`	900	infix	match predicate
`..`	896	infix	list abstraction
`in`	895	infix	set membership
`\/`	820	left	set union
`\`	820	left	set difference
`/\`	800	left	set intersection
`<>`	800	right	list append
`#`	760	infix	package separator
`:`	750	infix	type annotation

continued...

Table 1.2 Go! standard operators (cont.)

Operator	Priority	Assoc.	Description
`$=`	731	infix	constructor type
`@>`	731	infix	constructor type
`@=`	731	infix	constructor type
`+`	720	left	addition
`-`	720	left	subtraction
`*`	700	left	multiplication
`/`	700	left	division
`quot`	700	left	integer quotient
`rem`	700	left	remainder function
`**`	600	left	exponentiation
`%%`	500	infix	grammar parse
`^`	500	infix	grammar iterator
`~`	935	infix	grammar remainder
`.`	450	infix	object access
`private`	1700	prefix	private program
`import`	900	prefix	import module
`case`	950	prefix	case analysis
`\+`	905	prefix	logical negation
`@`	905	prefix	tau pattern
`raise`	900	prefix	raise exception
`valis`	905	prefix	return value
`istrue`	905	prefix	return value
`$`	897	prefix	initialization
`:`	750	prefix	type annotation
`-`	300	prefix	arithmetic negation
`.␣`	1900	postfix	statement terminator
`;`	1150	postfix	action terminator
`!`	905	postfix	one solution operator
`+`	760	postfix	input mode
`-`	760	postfix	output mode
`-+`	760	postfix	bidirectional mode
`+-`	760	postfix	bidirectional mode
`++`	760	postfix	super input mode
`*`	700	postfix	action type

continued. . .

Table 1.2 Go! standard operators (cont.)

Operator	Priority	Assoc.	Description
`^`	500	postfix	string convertion

Representing operators In our standard definition of Go! grammar, we represent the different operators as clauses in the `preOp`, `infOp` and `postOp` relations. Each clause in the `infOp` predicate takes the form:

```
infOp:[string,integer,integer,integer]{}.
...
infOp("*",720,720,719).
```

where `"*"` is a left associative infix operator of priority 720. This is encoded in the three numbers: the first number represents the priority of the term expected on the left of the `*` expression, the second represents the priority of the `*` expression itself and the third number represents the expected priority of expressions to the right of the `*` operator. Since `*` is left associative, this implies that a `*` expression can appear to the left, i.e., the expected priority on the left is the same as the actual priority of the `*` operator. For example, an expression such as

```
A*B*C
```

is parsed as though it were

```
(A*B)*C
```

A non-associative operator, such as `!=`, is represented by a clause such as:

```
infOp("!=",899,900,899). -- not equal predicate
```

The Left and Right expected priorities are both less than the priority of `!=` itself; which implies that multiple occurrences of them must be correctly parenthesized. An expression such as

```
a != b != c
```

occuring in the program text would not be valid.

Prefix operators are represented in a similar way. The prefix operator `-` which is not associative and has priority 300 is represented by a clause in the `preOp` relation:

```
preOp:[string,integer,integer]{}.
...
preOp("-",300,299).
```

The first number is the priority of the `-` operator as a prefix operator, and the second number is the priority of the expected term that follows the operator. Note that the *prefix* priority of an operator can be different to its *infix* priority; however, if an operator's postfix priority is different to its infix priority then this may cause ambiguities. All of Go!'s operators have consistent infix and postfix priorities.

Postfix operators are represented as instances of the `postOp` relation. The postfix operator `!` which is not associative and has priority 905 is represented by a clause in the `postOp` relation:

```
postOp:[string,integer,integer]{}.
...
postOp("!",904,905);
```

In this case, the first number is the priority of the expected term to the left of the `!` operator as a postfix operator, and the second number is the priority of the `!` expression itself.

1.3.2 Parsing analysis

It is most convenient to consider parsing analysis as generating *parse trees* from streams of tokens. A parse tree is a tree-like structure that reflects operators such as `+`; as well as syntactic constructions such as lists, tuples and applicative expressions.

Abstract syntax type definition Parse trees are expressed in terms of an *abstract syntax*. An abstract syntax is a notation which has operators for core elements – such as characters, strings, symbols and numbers – and a way of combining these into

applicative forms – such as lists, tuples and function application. The type definition of the Go! abstract syntax is:

```
absTree ::=
    I(symbol)                        -- identifier
  | INT(integer)                     -- integer literal
  | FLT(float)                       -- floating point literal
  | SYM(list[char])                  -- symbol
  | CHR(char)                        -- character
  | STR(list[char])                  -- string literal
  | APPLY(absTree,list[absTree])     -- applicative
  | BRACE(absTree,list[absTree])     -- braced expression
  | SQUARE(absTree,list[absTree])    -- bracket expr.
  .
```

where the `IDEN`, `SYM`, `CHR`, `NM` and `ST` forms correspond to identifiers, symbols, characters, numbers and literal strings respectively. Note that the type definition used here is simplified compared to that used in a compiler: a 'real' version would include line number information that would help to identify the location of terms for reporting errors and for support for run-time debugging. We omit this for the sake of clarity.

The `BRACE` form is used for syntactic forms such as:

`sync(L){` *Action* `}`

which is represented as:

`BRACE(APPLY(I('sync'),[`*L*`]),[`*Action*`])`

and

`(integer,symbol){}`

which becomes:

`BRACE(APPLY(I(','),[I('integer'),I('symbol')]),[])`

The `BRACE` constructor is declared with a list of argument subtrees; however, the list will always either be empty or have exactly one element in it.

The `SQUARE` form is used for polymorphic type forms such as:

```
list[integer]
```

which is represented in the abstract syntax using

```
SQUARE(I('list'),[I('integer')])
```

Parsing over streams of tokens Note that the grammar rules for parsing Go! are expressed in terms of streams of `tokType` expressions rather than streams of characters. This is not a problem for the grammar formalism itself as the grammar notation is inherently polymorphic. For example, the grammar rule for a number in the token stream is:

```
term0:[absTree-]-->list[tokType].
...
term0(INT(N)) --> [IN(N)].
```

The basic rules for parsing Go! programs revolve around the notion of a *primitive* expression and an *operator* expression. These distinctions have nothing to do with the semantics of Go! programs; they only relate to the syntactic relationships of elements of the language. In Chapter 3 we provide an analysis of the elements of Go! that more closely relates to the meaning of a program.

As with the standard rules for Go! tokens, we give the rules for parsing sequences of tokens into parse trees as Go! grammar rules. For readability, we assume the input is parsed as a sequence of `tokType` entries (see section 1.2 on page 6), rather than the more realistic `tokenType` entries which carry line number information.

Primitive parse tree

A primitive parse tree can be a literal (such as a number, symbol, character, string or regular identifier), an applicative expression (such as a function application, or a rule head) or a bracketted expression (such as a list, or parenthesised expression.

Simple literals The main grammar production that corresponds to the primitive expression is `term0`. The first few cases of this are straightforward, and correspond to occurrences of simple literal values in the token stream:

```
term0:[absTree-]-->list[tokType].
term0(CHR(c)) --> [CH(c)].       -- A character literal
term0(STR(S)) --> [ST(s)],       -- A string literal
    stringSeq(s,S).
term0(INT(N)) --> [IN(N)].       -- An integer literal
term0(FLT(N)) --> [FT(N)].       -- A floating point literal
term0(SYM(S)) --> [SY(S)].       -- A symbolic literal
...
stringSeq:[string+,string-]-->list[tokType];
stringSeq(soFar,S) --> [ST(s)], stringSeq(soFar<>s,S).
stringSeq(S,S) --> "".
```

The production for string literals is worth noting here: a single string may be constructed by a sequence of string literals – they are all concatenated into a single string. This is the standard way that a Go! program may include long string literals spanning many lines: each fragment is placed as a separate string literal on each line; the parser concatenates them all into a string string.

Identifiers and applicative expressions An identifier may occur by itself, as in an occurrence of a variable, or it may signal an applicative expression, as in a function application. The various rules that capture these cases are amongst the most complicated in the entire Go! grammar. In part, this is to allow the grammar to parse expressions such as:

```
sync(X){
  X.Ok() -> stdout.outLine("Ok")
}
```

The core grammar rule for applicative expressions uses `term00` to express the rules for the *function* part of an applicative expression:

```
term0(T) --> term00(L), termArgs(L,T).
```

and `termArgs` to capture the possible forms of arguments – including none.

There are two main rules for `term00` – the identifier rule and the parenthesised expression rule. The first rule says that an identifier is a `term00` expression, provided that it is not also a standard operator. I.e., it isn't one of the symbols referred to in Table 1.2 on page 19.

```
term00:[absTree-]-->list[tokType].
term00(I(S)) --> [ID(S)],
    {\+isOperator(S)}.
term00(T) --> parenTerm(T).
```

The `termArgs` grammar production either encounters an opening parenthesis – in which case the arguments of an applicative expression are parsed – or not – in which case the leading 'function symbol' is interpreted as is.

The rules for `termArgs` are:

```
termArgs:[absTree+,absTree-]-->list[tokType].
termArgs(Pref,Term) --> [LPAR,RPAR],
    termArgs(APPLY(Pref,[]),Term).
termArgs(Pref,Term) --> [LPAR],term(A1,999),
    (COMMA,term(Ax,999))*Axˆ Args,[RPAR],
    termArgs(APPLY(Pref,[A1,..Args]),Term).
termArgs(Pref,Term) --> [LBRCE,RBRCE],
    termArgs(BRACE(Pref,[]),Term).
termArgs(Pref,Term) -->
    [LBRCE],term(Arg,2000),[RBRCE],
    termArgs(BRACE(Pref,[Arg]),Term).
termArgs(Pref,Term) --> [LBRA,RBRA],
    termArgs(SQUARE(Pref,[]),Term).
termArgs(Pref,Term) --> [LBRA],term(Arg,2000),
    (COMMA,term(Ax,999))*Axˆ Args,[RBRA],
    termArgs(SQUARE(Pref,[Arg,..Args]),Term).
termArgs(Pref,Term) --> [ID('.'),ID(Fld)],
    termArgs(APPLY(I('.'),[Pref,I(Fld)]),Term).
termArgs(T,T)-->[].
```

In effect, an applicative term consists of a 'function' applied to a sequence of expressions. This rule is iterative, allowing expressions of the form:

```
f[A](B,C)
```

whose parse tree representation is:

```
APPLY(SQUARE(I('f'),[I('A')]),[I('B'),I('C')])
```

There are other kinds of `APPLY` terms, arising from operator expressions, in addition to expressions using `termArgs` production.

Note that we have a special rule for dealing with the dot operator which enforces the requirement that the right hand side must be an identifier. This reflects the fact that method access in an object is always tightly bound: an expression such as

```
O.f(A,B)
```

is parsed as though it were:

```
(O.f)(A,B)
```

Parenthesised expressions Parenthesised expressions are enclosed by bracket characters. There are three such groups of characters – parentheses `()` which indicate tuples as well as operator overriding, square brackets `[]` which indicate list expressions and braces {} which typically indicate program structure such as theta expressions and classes.

```
parenTerm:[absTree-]-->list[tokType];
parenTerm(I('()')) --> [LPAR,RPAR].
parenTerm(T) --> [LPAR],term(T,2000),[RPAR].
parenTerm(I('[]')) --> [LBRA,RBRA].
parenTerm(APPLY(I(',..'),[L,R])) -->
    [LBRA],term(L,999), tList(R).
parenTerm(I('{}')) --> [LBRCE,RBRCE].
parenTerm(APPLY(I('{}'),[T])) -->
    [LBRCE],term(T,2000),[RBRCE].
```

List expressions The rules for `tList` cover the list notation. A Go! list expression is a sequence of terms separated by `COMMA`s, and enclosed in square brackets. If the last element of a list expression is separated by a `,..` (`CONS`) token then it denotes the remainder of the list rather than the last element.

```
tList:[absTree-]-->list[tokType].
tList(I('[]')) --> [RBRA].
tList(T) --> [CONS], term(T,999), [RBRA].
tList(APPLY(I(',..'),[L,R])) -->
    [COMMA],term(L,999),tList(R).
```

The abstract syntax of a list literal is based on the `,..` applicative expression. Thus the parse tree corresponding to the list literal `[A]` is:

```
APPLY(I(',..'),[I('A'),I('[]')])
```

Note that this refers only to the abstract parse tree representation of list pairs; the run-time representation of a list such as this may be considerably more succinct.

Operator expression

An operator expression can be an infix, prefix or a postfix operator expression. Most operators are infix; however it is possible for an operator to be simultaneously an infix and/or a prefix and/or a postfix operator. The grammar requires some disambiguation in the case that an operator is both infix and postfix.

Our productions for operator expressions are split into two fundamental cases: prefix expressions – handled by the `termLeft` grammar rules – and infix and postfix expressions – handled by the `termRight` grammar rules.

Prefix operator expressions A prefix operator expression consists of a prefix operator, followed by a term. The priority of the prefix operator should not be greater than the 'allowed' priority; and that the expected priority of the term that follows the prefix operator is based on the priority of the prefix operator itself.

The `termLeft` grammar does not report an error if it encounters an out-of-range prefix operator as the `termLeft` production may have been invoked recursively; and the prefix operator expression *may* have significance at an outer level. We rely on backtracking within the parser to resolve this particular conflict.

```
termLeft:[absTree-,integer+,integer-]-->list[tokType].
...
termLeft(APPLY(I(Op),[Right]),P,Oprior) -->
    [I(Op)],
    {isPreOp(Op,Oprior,OrPrior), P>=Oprior},
    term(Right,OrPrior).
```

The second rule for `termLeft` defaults to the primitive term expression in the case that the lead token is not a prefix operator:

```
termLeft(Term,P,0) --> term0(Term).
```

Infix and postfix operator expressions The `termRight` grammar production defines how infix and postfix operators are handled.

The 'input' to `termRight` includes the term encountered to the left of the infix or postfix operator. If the next token is an infix operator, then the right hand side term is parsed – with appropriate expected priorities – and we recursively look again for further infix and/or postfix operators:

```
termRight:[absTree+,integer+,integer+,integer-,absTree-]-->
    list[tokType].
...
termRight(Left,prior,lprior,aprior,Term) -->
    [I(Op)],
    { isInfOp(Op,L,O,R),O=<prior,L>=lprior },
    term(Right,R),
    termRight(APPLY(I(Op),[Left,Right]),
          prior,O,aprior,Term).
```

The input to the recursive call to `termRight` includes the infix term just discovered.

Postfix operators are treated in a similar way to infix expressions, except that postfix operators do not have a 'right hand' expression:

```
termRight(Left,prior,lprior,aprior,Term) -->
    [I(Op)],
```

```
    { isPstOp(Op,L,O),O=<prior,L>=lprior },
    termRight(APPLY(I(Op),[Left]),prior,O,aprior,Term).
```

Note that we make use of Go!'s backtracking to help disambiguate the case where an operator is simultaneously an infix and a postfix operator. There are several such operators, for example: `.␣`, `+` and `^`. When encountering such dual-mode operators, the first interpretation as an infix operator is tried first; and if that fails then the postfix interpretation is used.

Special infix operators The `.␣(TERM)` operator is a special operator in that it can serve both as punctuation and as an operator. As punctuation, it serves as an expression terminator and it serves as a separator operator when encountered in a class body or package.

These punctuation symbols' roles as operators are captured via additional rules in the `termRight` grammar:

```
termRight(Left,_,lprior,lprior,Left),[TERM] --> [TERM].
termRight(Left,prior,lprior,aprior,Term,true) -->
    [TERM],
    { 2000=<prior,1999>=lprior },
    term(Right,2000),
    termRight(APPLY(I('. '),[Left,Right]),
                prior,2000,aprior,Term).
termRight(Left,prior,lprior,aprior,Term) -->
    [TERM],
    { 2000=<prior,2000>=lprior },
    termRight(APPLY(I('. '),[Left]),
      prior,2000,aprior,Term).
```

We also need a special rule to deal with tupling operator `,` (`COMMA`):

```
termRight(Left,prior,lprior,aprior,Term) -->
    [COMMA],
    { isInfOp(',',L,O,R),O=<prior,L>=lprior },
    term(Right,R),
    termRight(APPLY(I(','),[Left,Right]), prior,O,aprior,Term).
```

The final rule for `termRight` simply returns the left-hand expression and does not consume further tokens:

```
termRight(Left,_,prior,prior,Left) --> [].
```

Top-level term parse tree Our final production, for `term` itself, invokes the grammar for prefix/simple expressions and the grammar for infix and postfix expressions:

```
term:[absTree-,integer+]-->list[tokType].
term(T,prior) --> termLeft(Left,prior,Lprior),
    termRight(Left,prior,Lprior,_,T).
```

1.3.3 Example parse tree

The abstract parse trees of even simple expressions can be quite large. For example, the abstract tree corresponding to:

```
app([E,..X],Y) => [E,..app(X,Y)]
```

is

```
APPLY(I('=>'),
      [APPLY(I('app'),
             [APPLY(I(',..'),
                    [I('E'),I('X')]),
              I('Y')]),
       APPLY(I(',..'),
             [I('E'),
              APPLY(I('app'),
                    [I('X'),I('Y')])])])
```

Moreover, abstract parse trees only represent one intermediate kind of structure in a typical Go! compiler. However, by the time that the above equation is compiled, it will have been reduced to just a few instructions.

Types 2

Go! is a statically checked strongly typed language. Strong typing means that every variable and every expression has a single type associated with it, and that the uses of these expressions are consistent with expectations. Static type checking means simply that types are checked at compile-time rather than at run-time. Type errors arise when type checking detects an inconsistency. For example, if a function `foo` is defined over lists, then passing a numeric valued expression to `foo` is inconsistent because no list is equal to any number.

2.1 Go!'s type language

Go!'s type language is founded on four key concepts. The *type expression* is a term that denotes a type. However, although logically type terms are terms, they not normal logical terms in that they are not manipulable by Go! programs: their meaning and existence is strictly compile-time. To help distinguish type terms from regular terms we use square brackets for type constructor terms – as opposed to round parentheses for normal constructor terms.

Type terms are related to each other by the *sub-type* relation: which represents a partial ordering on type terms. The sub-type relation is indicated by the programmer explicitly declaring which type terms are sub-types of other type terms – as part of type definition statements.

The third key concept is the *type interface*. A type interface defines the set of queries and operations that may be performed relative to a labeled theory – specifically, it defines the kinds of

'dot' references that are supported by a given type of value. All named types, including system types, may support an interface. If the type term defines what kind of value an expression has, the type interface defines to a large extent what you can do with the value.

Finally, *type inference* is used to reduce the burden of annotating a program with type expressions. Programs and top-level names are *declared*, but type inference is used to verify that programs *conform* to the declared type, and is also used to automatically determine the types of rule argument variables and pattern variables; a great saving in a rule-oriented language.

Every expression in a Go! program is associated with a type term which is called the expression's *type assignment* or *type denotation*. At the start of the type inference process all unknown identifiers are assigned an unbound type variable as their type. Type inference uses declared program types, and the types of the program's literal values, to infer a type value or a set of consistent sub-type constraints for each of these type variables.

In addition to the type assignment, there are a number of *type constraints*. Type constraints encode the rules for type safety in programs. There are two kinds of type constraints, type *inequality* constraints that reflect the sub-type relationship and *program* constraints that reflect the program being type checked. Typically, program constraints are constraints on the arguments of functions and other programs; for example, that the type of a function argument is the declared argument type, or a sub-type of the declared type. A program is *type safe* if there is a single consistent type assignment for all the identifiers in the program and the set of all the inferred type predicates are consistent.

2.1.1 Sub-type constraint

A sub-type constraint is a statement of the form:

```
T <~ Tp
```

where `T` is a named type and `Tp` is a named type or a type interface. This statement declares that `T` is a sub-type of `Tp`. For example, the statements:

```
student <~ person.
marriedStudent <~ student.
```

declare that `student` is a sub-type of `person`, and that `marriedStudent` is a sub-type of `student` and hence of `person`. So the constraint:

```
T <~ person
```

is satisfied if `T` is `person`, `student` or `marriedStudent` – or some other sub-type of the `person` type.

In the case that `Tp` is an interface, as in, for example,

```
person <~ { age:[]=>integer}
```

then this means that the `person` type implements an interface that includes an `integer`-valued function `age`.

Note how we use square brackets here for the list of arguments. This is to further reinforce the distinction between type expressions and normal terms.

We can combine, for a given type, the two forms of type statement:

```
student <~ { studies:[]=>string}.
student <~ person.
```

This means that `student` is a sub-type of `person` and that it also implements an interface with a `studies` function, the different interfaces are combined. Given only these two type statements, `student`'s full interface would be

```
{ studies:[]=>string. age:[]=>integer}.
```

A lattice of types

The set of types forms a *type lattice*. A lattice is simply a set with a partial ordering associated with it; together with a top element (`top` in Go!'s case) which is larger than all other values and a bottom element (`void`) which is smaller than all others. A type lattice, as in figure 2.1 on the next page, is a kind of lattice where

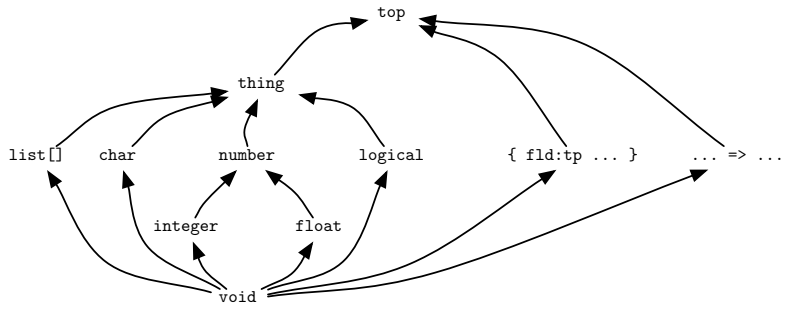


Figure 2.1: Part of Go!'s standard type lattice

the elements are type terms; and the partial ordering is, in fact, the *sub-type* relation.

In the sub-type partial ordering, higher in the order means more general and lower in the ordering means more specific. Thus `top` type is the most general type (and therefore the least is known about values of type `top`) and `void` is so specific that there are *no* legal `void` values.

The significance of `top` and `void` is largely technical, however, a function that accepts `top` arguments will accept anything and a `void` value is acceptable for all functions. On the whole, if you see either `top` or `void` in a type expression in an error message, you are likely to be in trouble!

You will notice that Go!'s type lattice is wide and shallow – that for the most part there are few significant *chain*s in the lattice. This is in the nature of type lattice systems. However, when defining types, particularly in terms of type inheritance, then we do get a richer network. For example, figure 2.2 on the facing page shows the lattice associated with the `number` type.

This graph highlights the fact that a lattice is not necessarily a simple *basket* or *chain*, but can have have branching elements in it. It cannot, however, have cycles – a lattice with a cycle is not permitted.

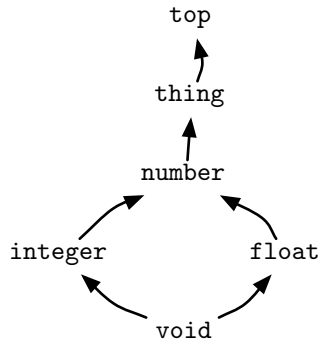


Figure 2.2: Go!'s `number` type lattice

2.1.2 Polymorphism

Types may be *polymorphic* and *recursive*. This polymorphism is reflected in the names of types – they can have type arguments. For example, the `char` type is the type of characters; and `list[char]` represents the type list of `char`acters – i.e., `string`s.

The list type (`list[`*`type`*`]`) is polymorphic: it requires a type argument which is the type of the elements of the list. It is also recursive because the type of a component element of a list term – namely the tail of the list – is also of the same type as the whole list. A recursive type is one whose values contain components that are of the same type as the whole. Although the `list[...]` type is built-in to Go!, it is straightforward to define new recursive types.

Recursive types have a particular hallmark: their type terms are *opaque* – it is not possible to infer solely from the type expression what values of that type look like. For example, although a list is necessarily defined as a term which includes a list as a component (the tail of a list is also a list of the same type) that recursion is not itself reflected in the `list` type term. In contrast with this, type expressions for functions are *transparent* – it is obvious from the type term what the structure of the values are – and conversely, transparent types may not be recursive. For tuples, that means, for example, that it is not possible to define a

tuple type in Go! that has the same tuple type as itself as one of the elements of the tuple:

```
( number, ( number, ( number, ...)))
```

Example 2.1: an impossible recursive tuple type

2.2 Type inference

Go! program expressions are only meaningful if a consistent type assignment or consistent sub-type constraints can be inferred for them using the declared program and class types. The process of inferring a consistent sent of type assignments and sub-type constraints to a Go! program is called *type inference*.

The fundamental predicate relating types to each other, and hence the key concept for understanding Go!'s type inference system is the subtype relation. In presenting the sub-type relation we use the notation:

$$T_1 \preceq T_2 \tag{2.1}$$

to denote that the type T_1 is a subtype of T_2. Implicit in this is always the possibility that the types T_1 and T_2 are *equal*.

As noted above, Go!'s types form a lattice with more general types being higher in the lattice and more specific types lower in the lattice.

To define the meaningful type associations we use a form of type inference rule notation. The general form of a type rule is:

$$\frac{Condition}{E \vdash_t X \rightsquigarrow T}$$

where the intended reading is:

> The type associated with the expression X, in the context of the environment E, is T providing that *Condition* holds. Often *Condition* is a conjunction of other type constraints mentioning some of the same variables as in consequent.

The environment E is a sequence of pairs, of the form `(`*`Name`*`,`*`Type`*`)`, which is used as a means of recording the type associated with the identifiers that may be in scope when assigning a type.

If we are inferring that one type is less than another, then we are likely to use an inference rule of the form:

$$\frac{Condition}{T_1 \preceq T_2}$$

since, in many cases, the subtype inference depends on preconditions being satisfied.

In addition to type associations and subtype determinations, we also need to make certain other kinds of type inference. For example, the $E \vdash_{safe} Goal$ predicate is satisfied if its goal argument is type consistent in environment E. Other predicates we will need include $E \vdash_{safe_A} Act$ which is satisfied for an action that is type safe, $E \vdash_{safe_G} G \rightsquigarrow S$ which is satisfied in context E if G is a grammar over streams of type S, .

2.2.1 Type variables

The process of type inference is complicated by the fact that it is often not known *what* a particular type actually is. For example, during the inference process associated with the expression:

```
O.f()
```

the type of `O` may not be known immediately. In which case the type inference process will be computing whether

$$\texttt{O}_t \texttt{ <~ \{ f:[]=>R}_t \texttt{ \}}$$

If the type of `O` is not known, then it must record this inequality against the type variable $\texttt{O}_t$; in case another step in the type inference process requires it or other constraints.

In general, in a type lattice, all type variables may be characterized by two types: an upper bound and a lower bound. The upper bound denotes that type which it known that the type variable is *lower* than, and conversely the lower bound determines a *minimum* type for the variable.

The type inference system goes one step further than this: it assumes that if the only thing known about two type variables is that one is greater than the other, it assumes that they are equal. A consequence of this is that the lower and upper bounds of a type variable are always known to be non-variable.

From a strict mathematical point of view, this is a strong inference that is not sanctioned by lattice theory: it is sound – in the sense that two such types may indeed be equal – but not complete, since it is quite possible for there to be a third type *between* the upper and lower types.

However, in practice this is not nearly as big an issue as it might seem. The reason is that the most common situation where variable-variable subtyping arises is in recursive programs. Again, in such situations, the most common scenario is:

`V`$_1$ `<~` `V`$_2$ `<~` `...` `V`$_1$

I.e., all of the type variables are, in fact equal!

Finally, the effect of this restriction is safe; in the sense that potentially type correct programs will be rejected – but it will not permit a type incorrect program to be accepted. Such false rejections can normally be resolved by additional type annotations on the text of the program that eliminate the implicit uncertainty.

The benefits of this form of type inference are two-fold: the type inference procedure itself is highly efficient and always terminates; and, in the case of erroneous programs, the error messages displayed by the type inference system are bounded.[1]

2.3 Standard Types

Go! has a range of built-in 'atomic' types: `integer`, `float`, `char` and `symbol`; and a range of built-in *type constructors*:

- `list[`*type*`]` (list of *type*),
- `[`*type*$_1$`,...,`*type*$_n$`]{}` (predicate type),

[1] Both of these are serious concerns for non-trivial programs.

- `[`$type_1$`,...,`$type_n$`]=>`*type* (function type),
- `[`$type_1$`,...,`$type_n$`]-->`*type* (grammar type),
- `[`$type_1$`,...,`$type_n$`]*` (action type), and
- { $field_1$`:`$type_1$`.` `...`$field_n$`:`$type_n$`.` } (type interface)

2.3.1 Type variables

Go! does not use any special lexical markers to distinguish type variables from other variables, or even other type names – the scope of the identifier serves to distinguish the cases. An identifier `foo` occurring in a type expression will refer to a type name if a type definition for `foo` is 'in scope'; otherwise it refers to a type variable.

The normal scope rules do not apply for certain of Go!'s built-in types; for example, the identifier `number` (say) is predefined in the language and always refers to the `number` type.

Universally quantified types A *universally* quantified type is written using the notation:

`[`s_1`,...,`s_n`]-`*Type*

this type binds the type variables s_i occurring in the *Type* expression. Go! supports nested quantification of type terms: if *Type* contains a type expression that is itself quantified and binds any or all of the s_i type variables then the inner occurrences refer to the inner-most quantification.

It is not normally required to manually identify the type variables in a type expression in this way; however, when type expressions are printed (in error messages typically) they will be displayed in fully quantified form.

2.3.2 Standard value types

Most of Go!'s standard types are defined in a standard library: `go.stdlib`. This library, which also includes a number of utility

definitions, is automatically `import`ed in every package.[1]

`thing` type

The `thing` type is the top of the value part of the type lattice – all values are a sub-type of `thing`. It is not the top of the type lattice itself – that is `top`. The program types – such as function type – are not a sub-type of `thing` – although they are sub-types of `top`.

The `thing` type has a simple type interface; it is declared to be:

```
thing <~ {
  show:[]=>string.
  meta:[]=>meta.
  }.
```

The `show` function is used to compute a printable `string` display of a term and `meta` is used to compute a meta-term representation of a term. Note that different types of terms may well use different methods for displaying themselves; it is only the *type* of `show` that is defined at this juncture.

`number` type

The `number` type symbol is used to denote the union type of integer and floating point values and expressions. The `number` type has the definition:

```
number <~ thing.
```

`integer` type

The `integer` type symbol is used to denote the type of integer values and expressions. The `integer` type has the definition:

```
integer <~ number.
```

[1] It is possible, if not advised, to suppress this behavior with a special compiler option.

`float` type

The `float` type symbol is used to denote the type of floating point values and expressions. The `float` type has the definition:

```
float <~ number.
```

Note that there is no subtype relationship between `integer`s and `float`s. They are, however, both subtypes of the standard type `number`. This can occasional quirks where it is necessary to specifically convert `integer` values to `float`s using the built-in function `n2float`.

`char` type

The `char` type symbol is used to denote character values and expressions. The `char` type has the definition:

```
char <~ thing.
```

`symbol` type

The `symbol` type symbol is use to denote symbol values and expressions. Note that this type refers to the 'general' class of symbols – literals of which are written as identifiers surrounded by single quote characters. The other main class of symbols – those introduced within user-defined type definitions are not covered by this type symbol; and nor are they written with single quotes.

The `symbol` type has the definition:

```
symbol <~ thing.
```

`logical` type

The `logical` type is used to denote truth values. Although a standard type, `logical` can be defined as a normal user-defined type, using the type definition:

```
logical ::= true | false.
```

`opaque` type

The `opaque` type symbol is used to denote certain 'internal' values that are managed by the Go! system. There is no written notation that corresponds to opaque literal values, and they will never be displayed in normal circumstances.

`meta` type

The `meta` type is used to represent a meta-level representation of a term. It is defined using the definition:

```
meta <~ thing.
```

Tuple type

The `,` type is used to denote pairs of values. Although a standard type, `,` can be defined as though it were a normal user-defined type, using the definition:

```
(u,v) ::= (u,v).
```

This rather bizarre type definition declares that the tuple type – denoted by the type term `(u,v)` has a single constructor, also `(u,v)`. This notation is a slight twist on the normal type notation – we are using the `,` operator as an infix type operator which is not actually permitted in programs.

`list` type

The list type is used to denote lists of values. The list type is written using the notation:

`list[`*type*`]`

For example, the type expression:

```
list[char]
```

denotes the type 'list of `char`'. This is the type assignment given to expressions which are inferred to be lists of `char`acter – including string literals. Go! supports the `string` synonym for the `list[char]` type term.

The `list` type has a definition equivalent to:

```
list[t] <~ thing.
list[t] <~ {
    head:[]=>t.                 -- head of the list
    tail:[]=>list[t].           -- tail of the list
    eof:[]{}.                   -- is the list empty
    cons:[t]=>list[t].          -- new list on the front
    tack:[t]=>list[t].          -- new list on the back
    hdtl:[t,list[t]]{}.         -- pick off the head and tail
    eq:[list[t]]{}.             -- is equal to this list
}
```

2.3.3 Program types

Type modes

Program types have argument types that denote the types of arguments to the program. However, the types of arguments are also associated with *modes* – which express constraints on the flow of information into or out of a program via its arguments.

There are four kinds of modes: input mode, super-input mode, output mode and bidirectional mode:

input mode If an argument is marked as being input mode then the expectation is that data flows into the program through that argument. This carries two implications: one for type checking and one dynamic data flow:

- An actual argument to a program that corresponds to an input moded argument may have a type that is a sub-type of the expected type.
- An actual argument to a program must not be *bound* as a result of the matching of input arguments to patterns. Where the actual argument is a non-variable, this is not

an issue; where the actual argument is an unbound variable then either the matching pattern also corresponds to an unbound variable or the match will *fail*.

- In order to explicitly note an argument type to be input, suffix the type of the argument with the `+` operator:

  `pp:[integer+]{}`

super input mode The super input mode is based on the input mode; with the additional operational characteristic that if a program is invoked where the super input moded argument is unbound then the call to the program is *delayed* until such time as the variable becomes bound.

If the variable is never instantiated, then the delayed program is never invoked.

The super input mode is marked by suffixing the type of the argument with the `++` operator:

`listOfInt:[list[integer]++]{}`

output mode An output moded argument is the converse of an input moded argument:

- An actual argument's type must be *equal* to the expected type for that program argument.
- at run-time an actual argument *must* be an unbound variable; otherwise the matching of patterns to values will *fail* even if the corresponding pattern is an unbound variable.
- In order to explicitly note an argument type to be output, suffix the type of the argument with the `-` operator:

  `pp:[integer-]{}`

bidirectional mode A bidirectional moded argument can be either input or output; and unification is used to match the actual argument to the pattern.

- The type of the actual argument must be equal to the expected type for that argument
- Unification is used to match the incoming value against the pattern; hence the flow of information may be either incoming, outgoing or a mixture.
- In order to explicitly note an argument type to be bidirectional, suffix the type of the argument with the `-+` operator:

  `pp:[integer-+]{}`

Although it is possible to associate modes with every argument type of every program type, program types have defaults that are intended to represent the normal mode of use for that kind of program.

Function type

The function type is used to denote function values. The function type is written:

`[`T_1`,...,`T_n`] => `T_R

where `[`T_1`,...,`T_n`]` is a list of the type expressions corresponding to the arguments of the function and T_R corresponds to the result of the function. For example, the type expression:

```
[list[char],list[char]]=>list[char]
```

denotes a function – from two string arguments to a string result.

The default mode for function arguments is *input*. However, it is occasionally useful to mark a function argument as output – as an out-of-band way of returning values from a function.

Predicate type

The predicate type is used to denote relational or predicate values. The predicate type is written:

`[`T_1`,...,`T_n`]{}`

where `[`T_1`,...,`T_n`]` is the list of type expressions of the arguments of the predicate. For example, the type expression:

```
[list[char],list[char],list[char]]{}
```

denotes the type of a ternary predicate where all the arguments are strings.

The default mode for predicate arguments is *bidirectional*. However, it is often useful to mark a predicate argument as input – to be explicit about the expect usage of the predicate.

Action type

The action type is used to denote procedure values. The action type is written using a postfix `*` operator:

`[`T_1`,...,`T_n`]*`

where `[`T_1`,...,`T_n`]` is the list of type expressions of the arguments of the procedure. For example, the type expression:

```
[list[char]]*
```

denotes the type of a unary procedure whose single argument is a string.

The default mode for action procedure arguments is *input*. However, it is occasionally useful to mark an argument as output – as that is the only way of arranging for output from an action procedure call.

Grammar type

The grammar type is used to denote grammar values. The grammar type is written as a `-->` mapping from the types of the arguments of the grammar to the type of the stream of values the grammar is defined over:

`[`T_1`,...,`T_n`] --> `T_S

where `[`T_1`,...,`T_n`]` is the list of type expressions of the arguments of the grammar and T_S is the type of the stream that the grammar is defined over – typically a list of some kind. For example, the type expression:

```
[number]-->list[char]
```

denotes the type of a grammar whose single argument is a number and which may be used to parse strings.

The default mode for grammar arguments is *bidirectional*. However, it is occasionally useful to mark a grammar argument as input – to be explicit about the expect usage of the grammar, and also to mark an argument as output – to be explicit about the outputs associated with parsing a stream.

Class type

There are two kinds of class type declarations: those that introduce a stateless class and those that introduce a stateful class.

The state-free class type is used to denote constructor classes, analogous to constructor functions. The stateful class type denotes classes that may carry state.

A state-free class type declaration takes the form:

class`:[`T_1`,...,`T_n`] @= `*Type*`.`

The only permitted mode for the label arguments of a state-free class label is *bidirectional*.

A stateful class type declaration takes the form:

class`:[`T_1`,...,`T_n`] @> `*Type*`.`

The default mode for label arguments of a stateful class label is *input*.

The reason that state-free label arguments must be bi-directional is that class labels are equivalent to constructor functions: i.e., they have an inverse. That means that it is always possible to recover the arguments of a 'call' to a constructor function.

> On the other hand, one of the differences between state-free class labels and statefull class labels is that the latter *do not* have inverses – they are semantically closer to regular functions.

Note that a class type statement does not define the class itself, it simply defines its type.

2.4 Algebraic type definitions

In addition to a type being defined using the sub-type statements, a type may also be introduced using an *algebraic type definition* statement. An algebraic type definition is one where the type is introduced at the same time as a set of enumerated symbols and constructors for the type:

UserT`[`T_1`,...,`T_n`] ::= ... | `*S*` | ....`

where T_i are identifiers indicating type arguments and the right hand side is a series of enumerated symbols and constructor function templates.

For example, the `tree` type defined below may be used to denote tree values:

```
tree[a] ::= empty | node(tree[a],a,tree[a]).
```

This statement defines a new type type constructor `tree` together with the enumerated symbols and constructor terms that make up values of the `tree` type.

2.4.1 Type parameters of a type definition

Where the template of a *UserType* takes the form:

UserType`[`T_1`,...,`T_n`] <~ ...`

the various arguments T_i are the *type parameter*s of the type definition. They must all be identifiers and they are interpreted as

type variables. Such a *UserType* is implicitly universally quantified with respect to the type parameters (hence the polymorphism of the type).

Go! imposes a restriction on type variables occurring in a type definition: all type variables appearing in the body of the type definition – i.e., appearing in templates for the type constructors in class definitions – must also appear in the type template expression itself.

2.4.2 Enumerated symbol

An *enumerated* symbol is equivalent to a zero-arity label term. For example, the introduction of `empty` in the definition of `tree` above is equivalent to the class definition:

```
empty:tree[a] .. {}.
```

The type analysis of a definition of an enumerated symbol may be captured via an 'introduced' type inference rule:

$$\forall \vec{V}. \left\{ \frac{}{E \vdash_t \mathtt{S} \rightsquigarrow \mathit{UserT}\,\mathtt{[}T_1\mathtt{,...,}T_n\mathtt{]}} \right\}$$

where $\vec{V}$ are all the type variables ocurring in the type definition and `S` is the newly introduced enumerated symbol.

Note that, in general, the type of a enumerated symbol can have type variables even when clearly the symbol itself has no variables. We can see this more clearly with the empty list case. Go!'s list notation is based on the `,..` constructor function and the `[]` enumerated symbol; which might be captured in the algebraic type definition:

```
list[T] ::= [T,..list[T]] | []
```

The type of an occurrence of the empty list is, then, an expression of the form:

$\mathtt{list[T}_i\mathtt{]}$

where $\mathtt{T}_i$ may or may not be known. The logic of this is that an empty list is always of a list of a specific type – even if we cannot determine what that type is in given circumstances.

2.4.3 Constructor functions

Constructor functions (a.k.a. class labels) are analogous to functors in `Prolog`: they fulfill very similar roles. However, their semantics are quite distinct to normal `Prolog` terms since they always are associated with logical theories – even when defined within an algebraice type definition.

Constructor functions are so-named because they are functions: with the particular property that every expression involving the constructor function has an exact inverse. This property allows constructor functions to be used as patterns as well as in other expressions.

The type assignment for constructor functions can be viewed as an additional type inference rule; where a type definition was of the form:

$$\mathit{Template}\ \texttt{::=}\ \ldots\ \texttt{|}\ F\texttt{(}T_1\texttt{,}\ldots\texttt{,}T_n\texttt{)}\ \texttt{|}\ \ldots$$

we introduce a new inference rule of the form:

$$\forall\vec{V}\ \frac{\mathrm{E}\vdash_t A_1\rightsquigarrow T_1\qquad\ldots\qquad\mathrm{E}\vdash_t A_n\rightsquigarrow T_n}{\mathrm{E}\vdash_t F\texttt{(}A_1\texttt{,}\ldots\texttt{,}A_n\texttt{)}\rightsquigarrow \mathit{Template}} \tag{2.2}$$

where $\vec{V}$ are the type variables ocurring in the type definition.

2.4.4 Type extension

One of the key features of Go!'s type system is its extensibility. In particular, because types are distinct from classes, there is always the potential to introduce new constructors for a type – including types that have been introduced using an algebraic type definition.

For example, given the `tree[]` type, we may decide that we need a new kind of node for a tree – perhaps a binary tree node that does not include a label. We can do so simply by defining a class for it:

```
bin:[tree[a],tree[a]]@=tree[a].
bin(L,R) .. { ... }
```

The effect of this is as though the original type definition were:

```
tree[a] ::= empty
          | node(tree[a],a,tree[a])
          | bin(tree[a],tree[a]).
```

We can even introduce this class in a different package than the one in which `tree[]` itself is defined:

```
foo{
  import tree.
  bin:[tree[a],tree[a]]@=tree[a].
  bin(L,R) .. { ... }
}
```

Functions and Expressions 3

This chapter is about the functional programming aspect of Go!. Go! has a rich range of expressions, one of its distinguishing characteristics when compared to `Prolog`. Part of this comes from the fact that Go! combines a functional programming notation, and part comes from Go!'s multi-paradigm style: so there are expressions that relate expressions to actions and predicates as well as regular evaluable forms.

3.1 Functions

Functions are defined using sequences of equations. For example, Program 3.1.1 shows how a list concatenation function may be written. Each equation is a *rewrite* equation that shows how to rewrite terms of one form – representing the function call – to terms of another form – representing the value. The type declaration says that `concat` is polymorphic, mapping a pair of lists of any type of element `T` to a list of elements of the same type.

Program 3.1.1 A list `concat`enation function

```
concat:[list[T],list[T]]=>list[T].
concat([],X) => X.
concat([E,..X],Y) => [E,.. concat(X,Y)].
```

Functions may be defined either within a class body – in which case they are specific to that class – or at the outer level of a package – in which case they are global to the package. In both cases all the equations for a given function must be grouped together.

The general form of an equation is:

Fun(*Ptn*$_1$,...,*Ptn*$_n$ :: *Condition* => *Exp*

The equations in a function are applied in a left-to-right order. There is no deep backtracking in function evaluation: once an equation has been found that matches then no other equations will be attempted. In the event that none of the equations match an error exception

```
error(...,'eFAIL')
```

will be raised.

Matching equations to arguments Note that, by default, the head arguments of an equation are *matched* against the tuple of arguments rather than *unified*. The distinction is that matching is not permitted to side-effect its input – in this case matching against the head of an equation *will not* side-effect the arguments to the function call. For example, given a function application:

```
F(X,2)
```

then if `F` is defined by the equation:

```
F('a',2) => 3
```

then the match will *fail* (and raise an error if there are no alternatives equations for `F`) since the only way that it could succeed would be by binding `X` to `'a'`.

The only exceptions to this matching semantics are where the argument's mode as declared in the function's type declaration is bidirectional – in which case the incoming argument will be *unified* – or if the argument is marked as output – in which case the incoming argument must be an unbound variable and a value will be returned in that argument.

For example, the type declaration:

```
F:[symbol-,integer]=>integer.
```

would require the above equation for `F` to be applied to a function call in which the first argument was unbound, and would permit equations that define `F` to instantiate the argument.

The match predicate `.=` (see Section 4.2.6 on page 84) may be used within a guarded term to achieve a similar matching semantics as for the head arguments themselves.

The type inference rule for the equations defining a function `F` is:

$$\frac{\mathrm{E} \vdash_t F \rightsquigarrow F_H \texttt{=>} F_R \qquad \mathrm{F}_H = \mathrm{T}_H \qquad F_R \preceq T_R}{\mathrm{E'} \vdash_{ok} H \texttt{::} G \texttt{=>} R} \tag{3.1}$$

I.e., the main requirement for a safe equation is that the head arguments have the same types as the declared argument types and the type of the right hand side expression of the equation is less than or equal to the function result type.

Note that the type rule for an equation is independent of the modes associated with the function type. In all cases, the type of the argument pattern must be the same as the type of the function. Modes affect calls to functions, and hence the right hand sides of equations, but do not affect the type analysis of argument *patterns*.

3.2 Atomic expressions

The atomic Go! expressions are the atomic or unstructured elements of the language: including characters, symbols and numbers. They are called atomic because, from the Go! programmer's perspective, they have no internal structure.

3.2.1 Symbols

As outlined in Section 1.2.4 on page 11, a symbol is written as a sequence of characters enclosed in single quotes. Symbols are one of the primitive data types of Go!, and have the type `symbol`.

We define the type inference rule for symbols thus:

$$\frac{}{E \vdash_t \texttt{'}S\texttt{'} \rightsquigarrow \texttt{symbol}} \tag{3.2}$$

which is interpreted as

> The type of an expression of the form `'S'` is `symbol`.

Logically, a symbol is a token that identifies an individual entity in the modeled world. Go! supports other kinds of symbols than quoted `symbol`s; user defined symbols as introduced via class definitions or in algebraic type definitions do not carry the `symbol` type: their type is that of the declared type. Such symbols are written using the normal identifier notation.

3.2.2 Characters

A character literal is written as a back-tick character followed by the character itself; which may be a string character reference. Characters are one of the core data types of Go!: lists of `char`acters form the basis of Go!'s string notation.

The type of a character expression is `char`:

$$\frac{}{E \vdash_t \texttt{`}C \rightsquigarrow \texttt{char}} \tag{3.3}$$

3.2.3 Numbers

Go! partitions numbers into two categories: `integer`s and `floating` point numbers. Both of these are sub-types of the `number` type; however, neither `integer` nor `float` are sub-types of each other. This reflects the common reality that integral and non-integral `number` values are represented differently.

There are various ways of writing integer literals: as decimal integers, hexadecimal numbers and character codes.

Integers

An integer is written as a sequence of decimal digit characters – with an optional leading minus sign to denote negative integers.

Although the standard ASCII digit characters are likely to be the most common digit characters used, there are approximately 250 decimal digit characters in the Unicode standard! The Go! parser supports the use of any Unicode decimal digit characters in constructing a number; however, the Go! system only uses ASCII digits when displaying numbers.

Hexadecimal numbers A hexadecimal number is written with a leading `0x` followed by hexadecimal digits. Note that, unlike regular integers, Go! requires that hexadecimal numbers use only the ASCII digit characters, together with upper-case or lower-case letters `A` to `F`.

```
0xffff 0xabd 0x0
```

are all integers, of type `integer`, written using hexadecimal notation.

Character codes It is also possible to specify an integer as the Unicode code value of a character. Such an `integer` is written with a leading `0c` followed by a single character reference. Thus, for example, the literal:

```
0cA
```

is the number 65 – as the Unicode code point value associated with upper-case `A` is 65. Similarly, `0c\n` is the `integer` 10, as the character `\n` refers to the new-line character – whose code value is 10. The value of

```
0c\+2334;
```

is, of course, the same as:

```
0x2334
```

$$\frac{X \text{ is an integer literal}}{E \vdash_t X \rightsquigarrow \texttt{integer}} \tag{3.4}$$

Floating point numbers

Go! distinguishes floating point numbers from integral values by the fact that floating point numbers always have a decimal point: i.e., a floating point number is written as a fractional number followed by a fractional part and an optional exponent expression. Example floating point numbers include:

```
34.56 2.0e45 2.04E-99
```

Since the period character has so many uses in Go!, there are some special rules for them. In the case of a floating point number there may be no spaces on either side of the period; i.e., the program text:

```
34 . 23
```

does *not* denote the number 34.23; it is three tokens: the integer 34, the rule terminator `.␣` and the integer 23.

Floating point literals are of type `float`, a sub-type of `number`.

$$\frac{X \text{ is a float literal}}{E \vdash_t X \rightsquigarrow \texttt{float}} \tag{3.5}$$

3.3 Variables

Variables are written as identifiers (see Section 1.2.2 on page 7). All identifiers which have not been defined as class labels or names of rule programs or declared as enumerated symbols or constructor functions in an algebraic type definition are considered to be variables. Go! does not have a variable convention to distinguish variables; instead we use the context of the identifier to distinguish them. Note that normal symbols (see Section 3.2.1 on page 57) are always surrounded by single quotes and cannot be confused with variables.

All variables have types, and each occurrence of a variable is type-checked – resulting in type constraints on its type: the constraints inferred with respect to each occurrence of a must be consistent.

A variable may not have a value and yet still have a type associated with it.

3.3.1 Scope of identifiers

Go! does not require that variables in rules be explicitly introduced – simply, an identifier occurring in a pattern or expression that does not have a prior declaration is assumed to be a variable. However, there are rules that define the *scope* of a variable: i.e., the textual range across which occurrences of the same name are considered to refer to the same variable.

For identifiers such as program names, type names and class names, introduced in the body of a class or package, they are in scope across the entire class body or package – there is no implied scope arising from the order of declarations. For variables in rules, the scope of the variable is the entire rule. Similarly, program names are in scope for their entire enclosing context, regardless of the order of declaration.

For a rule, such as an equation or a clause, any variables mentioned in the rule that are not defined in an outer scope – either in an enclosing class body or as package variables – are local to that rule.

A variable is *free* in a rule if its scope encompasses the rule as well as other, enclosing, program fragments. The main source of free variables in a rule are *label variables* of the class in which the rule is embedded. For example, in:

```
lbl(X)..{
  bar(X) :- X<10.
}
```

the variable `X` is *free* in the `bar` clause. The reason is that `X` is also a variable occurring in the label `lbl(X)` of the class that the `bar` clause is embedded.

Holes in the scope A hole in the scope of an identifier – i.e., where an inner use of an identifier can 'hide' an outer identifier with the same name – can occur when the inner use of the identifier

is as the name of a defined program of a class body. The inner variable masks out, throughout its natural scope, any variable of the same name defined in outer contexts.

Note that variable identifiers in rules *do not* mask out variables of the same name in outer scopes. Such a variable is a *free* variable of the rule. This is the principal mechanism in many cases whereby a program is referenced in function and predicate calls in the bodies of rules.

Consider the following definition of the state-free class:

```
funny:[T1,T2,T3] @= aType.
funny(A,Y,B)..{
  A(X).
  A([X,..Y]) :- B.is(X),A(Y).
};
```

The variable `B` introduced in the `funny` constructor is in scope across the whole of the class body defining `funny`. It is therefore a global variable of this class.

The variable `A` is in scope throughout the class as well. However, there is a definition of `A` in the class body that uses the same name `A`; this definition masks out the `A` that occurs as a parameter of the `funny` constructor. In effect, the `funny` class cannot use its `A` argument as it is masked by the inner `A` definition in the class body.

The `X` variable has two 'incarnations': in the first clause of the definition of `A` and in the second clause. These are separate variables. The `X` of the first clause is local to that clause.

The `Y` variable occurring in the head of second clause of `A` is *not* local – it is bound by an outer occurrence of the same variable in the argument list of the `funny` constructor. Like the `B` variable in the same clause, it is a *free* variable of the `A` program.

Singleton variables Note that the compiler will report a warning over any *singleton* occurrences of variables – unless the variable has a leading underscore in its name. Such singleton variables are often actually misspelt variables – hence the warning from the compiler. In this case, the compiler would report a warning over

both the `A` parameter of the function and the `X` parameter of the first `A` clause. The former helping to highlight the fact that `A` is effectively masked out by the theta environment.

Scope of type variables

Like regular variables, type variables also have an associated textual scope. The scope rules for type variables are essentially the same as for normal identifiers. Of particular interest are type variables occurring within rules that are *not* free – i.e., are not mentioned in an outer context. Such type variables have the same scope as a regular variable: the rule in which the variable is mentioned.

3.4 Standard structured terms

Go! has two standard data constructor types: lists and tuples. In addition, of course, it is possible to define additional data constructors using class definitions and algebraic type definitions.

In fact, both lists and tuples are definable using normal Go! mechanisms for defining constructor terms. They are highlighted because of their special role in the language.

3.4.1 Lists

Lists are a standard data type of Go!, the list data type is described in detail in Section 2.3.2 on page 44. The surface syntax of lists is defined in Section 1.3.2 on page 28.

A list is a sequence of values. The defining characteristic of a list is that the first element of the list – called the *head* of the list – and the remaining sequence of elements after the head – called the *tail* of the list – are available in a single constant step of computation. This distinguishes lists from arrays – where every element is available in constant time although arrays are not easily extended – and sets where each element may take $\log(n)$ time to access and there is only one occurrence of each element.

Lists are written as a sequence of comma-separated expressions enclosed in square brackets. For example, the list

```
[1,2,3]
```

is a list of three `numbers`: 1, 2 and 3, as is the expression:

```
[1,1+1,1+1+1]
```

List pattern notation There are two standard constructors for the list type: the empty list – written as `[]` – and the list cons constructor – written as the special infix operator `,..` consisting of a head element and a tail list:

```
[H,..T]
```

There is a direct correspondence between list patterns and list terms: the expression:

```
[1,[2,[3,..[]]]]
```

is equivalent to the list `[1,2,3]`.

Other forms of list pattern are also interesting. For example, the list pattern:

```
[1,2,3,..X]
```

denotes a list whose first three elements are known but whose tail is represented by the variable `X`. It is one of the magic aspects of logic programming that unifying `X` with `[4,5]` say, will have the effect of completing the above list pattern to

```
[1,2,3,4,5]
```

The type rules for the standard list constructors are

$$\frac{}{\forall T.E \vdash_t \texttt{[]} \rightsquigarrow \texttt{list[}T\texttt{]}} \tag{3.6}$$

and

$$\frac{E \vdash_t \texttt{H} \rightsquigarrow T_t \qquad E \vdash_t \texttt{T} \rightsquigarrow \texttt{list[T}_t\texttt{]}}{E \vdash_t \texttt{[H,..T]} \rightsquigarrow \texttt{list[T}_t\texttt{]}} \tag{3.7}$$

We can concatenate type inference rules to infer the types of more complex expressions. For example, we can use the type inference rules 3.6 and 3.7 to derive the type of this expression:

```
[1,2]
```

which gives the type derivation:

$$\dfrac{\dfrac{}{E \vdash_t \texttt{1} \rightsquigarrow \texttt{number}} \qquad \dfrac{E \vdash_t \texttt{2} \rightsquigarrow \texttt{number} \qquad E \vdash_t \texttt{[]} \rightsquigarrow \texttt{list[T]}}{E \vdash_t \texttt{[2]} \rightsquigarrow \texttt{list[number]}}}{E \vdash_t \texttt{[1,2]} \rightsquigarrow \texttt{list[number]}}$$

3.4.2 Strings

Go! string literal values are synonyms for lists of `char`s; i.e., a string literal such as `"foo"` is equivalent to the list:

```
[‘f,‘o,‘o]
```

and the empty string `""` is equivalent to

```
[]:list[char]
```

i.e., an empty list with the added type annotation that it's type is list of `char`s (see Section 3.5.1 on page 67).

3.4.3 Tuples

Like lists, tuples represent a way of aggregating values in a sequence. A tuple is written as a sequence of elements, separated by `,` and enclosed in parentheses:

```
('foo',23,['bar'])
```

Unlike lists, the elements of a tuple do not need to be of the same type.

As used in a tuple, the comma (`,`) is an infix constructor operator. The tuple type is provided as a convenient way of grouping values together; but it has no interface per se. The tuple type is defined as though by the type definition:

```
(,)[s,t] ::= (s,t)
```

where `s` and `t` represent the two degrees of polymorphic freedom available in a tuple.

This, rather circular, definition highlights the fact that, in Go!, the `,` infix operator is used both as the name of the tupling type as well as the tupling operator itself. It is, however, semantically, simply a predefined type with a single constructor that is very similar to a user-defined type.

The `,` operator is right associative, so it is possible to combine it into longer sequences:

```
('joe',23,"his place",tonight)
```

is equivalent to:

```
('joe',(23,("his place",tonight)))
```

3.4.4 Function Call Expression

A function call is an expression of the form:

Fun`(`A_1`,...,`A_n`)`

Function call expressions are typed using a rule that maps the type of the function and the type of the argument to the type of the result. Assuming a normal mode assignment for the type of a function `F`, the type rule for a function application is:

$$\frac{E \vdash_t F \rightsquigarrow \vec{L}\texttt{=>}R \qquad E \vdash_t \vec{A} \rightsquigarrow \vec{A}_t \qquad \vec{A}_t \preceq \vec{L}}{E \vdash_t \texttt{F(}\vec{A}\texttt{)} \rightsquigarrow R} \tag{3.8}$$

If there is one or more bidirectional or output moded arguments associated the function then the condition:

$$\vec{A}_t \preceq \vec{L}$$

becomes type equality:

$$\vec{A}_t = \vec{L}$$

For constructor functions, type equality is also required.

Note that if an applied function fails, if none of the function's equations apply to the arguments of the application, then an `'eFAIL'` error exception is raised. This may be caught using an `onerror` clause.

3.5 Special expressions

There are a number of expressions which are syntactic in nature, these include guarded expressions, action expressions, type annotated expressions and expressions relating to attribute sets.

3.5.1 Type annotation

A *type annotated expression* takes the form:

Ex : *Type*

A type annotated expression has the same value as its non-annotated component. The only effect of the type annotation is to add a type constraint to the expression:

$$\frac{E \vdash_t Ex \rightsquigarrow Type}{E \vdash_t Ex\colon Type \rightsquigarrow Type} \tag{3.9}$$

Type annotations are only rarely required within a normal Go! program. However, they can serve as useful documentation and in those circumstances where there is a hard to track down type error.

3.5.2 Bag of expression

The *bag of* a list, expression is a 'cousin' of the guarded expression – instead of a single expression which is defined if a goal is satisfied, a bag of expression represents the list of all the possible answers to a question. The bag of expression is written:

{ *Ex* || *Goal* }

The value of a bag of expression is a list consisting of a copy of the value of *Ex* for each way that *Goal* can be satisfied. The order of elements in the resulting list is the same as the order that *Goal* gives rise to possible solutions. It is of course possible for there to be multiple occurrences of a given value (hence the term 'bag of').

The type inference rule for the bag of expression is:

$$\frac{\mathrm{E} \vdash_t Ex \rightsquigarrow T_{Ex} \qquad \mathrm{E} \vdash_{safe} Goal}{\mathrm{E} \vdash_t \{Ex \,||\, Goal\} \rightsquigarrow \texttt{list[}T_{Ex}\texttt{]}} \tag{3.10}$$

Variables in bags may arise when *Ex* is not completely ground for one or more solutions to *Goal*. The list returned will also contain variables. More precisely, the bag-of algorithm works as follows:

1. The *Goal* is evaluated. If no solution to *Goal* is possible, then the value of {*Ex* || *Goal*} is the empty list.

2. Each time a solution to *Goal* is found, a 'frozen copy' of *Ex* is computed – in the context of the solution to *Goal*. This involves taking copies of any variables in *Ex* to produce new variables.

   After the *Ex* value is computed, a failure is forced to attempt to find additional solutions to *Goal*.

3. After the last solution to *Goal* has been found, the list of solutions found is 'thawed' and returned as the value of the bag expression.

3.5.3 Bounded set expression

The *bounded set* expression is similar in form, and in some cases similar in use, to the *bag of* expression (see Section 3.5.2 on the previous page). However, it owes its origin to a different style of programming and has quite different semantics.

The general form of the bounded set expression is:

{ *Ex* .. *Ptn* in *List* }

The value of a bounded set expression is a list consisting of evaluating the expression *Ex* for each member of *List* that matches with *Ptn*.

The semantics of the bounded set expression can be given in terms of a defining mapping from such expressions into other Go! expressions. I.e., a bounded set expression is entirely equivalent to the expanded form:

bounded$_X$(*List*,*Free*)

where `bounded`$_X$ is a new identifier not occurring anywhere else and is defined as though by the program:

```
boundedX([],Free) => [].
boundedX([Ptn,..L],Free) => [Ex,..boundedX(L,Free)].
boundedX([_,..L],Free) => boundedX(L,Free).
```

where *Free* is the tuple of free variables in *Ptn* and *Ex*. I.e., a bounded set is evaluated by a recursive iteration through the elements of the bounding set, computing an output value for each succesful match of the list element.

Note that the *Ptn* pattern may have a guard associated with it, considerably increasing the potential power of the form. For example,

```
{ X*X .. (X::X<10) in [1,2,3,50,23,2] }
```

will return the list:

```
[1,4,9,4]
```

The type inference rule for the bounded set expression is:

$$\frac{\mathrm{E} \vdash_t Ex \rightsquigarrow T_x \qquad \mathrm{E} \vdash_t Ptn \rightsquigarrow T_S \qquad \mathrm{E} \vdash_t List \rightsquigarrow \texttt{list[}T_S\texttt{]}}{\mathrm{E} \vdash_t \{Ex\texttt{..}Ptn\ \texttt{in}\ List\} \rightsquigarrow \texttt{list[}T_x\texttt{]}} \tag{3.11}$$

3.5.4 Conditional expressions

A *conditional expression* takes one of two values depending on the outcome of a test. Conditional expressions are written:

```
(Goal?E1|E2)
```

Note that the parentheses are required as the `|` is also used to separate clauses in a lambda expression. The type derivation rule for conditional expressions is

$$\frac{\mathrm{E} \vdash_{safe} Goal \qquad \mathrm{E} \vdash_t E_1 \rightsquigarrow T_E \qquad \mathrm{E} \vdash_t E_2 \rightsquigarrow T_E}{\mathrm{E} \vdash_t (Goal? E_1 \mid E_2) \rightsquigarrow T_E} \tag{3.12}$$

If *Goal* succeeds, then the value of the conditional expression is the value of the 'then' branch – E_1 – otherwise it is the value of the 'else' branch – E_2. *Goal* is evaluated in a 'one-of' context – only one solution for *Goal* is attempted.

Note that there is a certain asymmetry about E_1 and E_2 – if *Goal* succeeds and it instantiates one or more variables as it does so, then these values are 'available' in evaluating E_1; but (clearly) they are not available to E_2

3.5.5 Object creation

An occurrence of a label term where the constructor has been defined to be a stateful class is interpreted as an *object creation* expression.

This represents a contrast with object languages such as Java™ which have specific operators for creating objects. Go! has no such artificial distinction.

Specifically, given an expression of the form:

```
Term
```

where *Term* is a class label of a stateful class, returns a new symbol:

$\texttt{obj}_{random}$

together with an implied definition:

$\texttt{obj}_{random}$ `<=` *Term*

which indicates that the new term is a new class label. This new 'object' has the same type as *Term* and is associated with the same class definitions as *Term* but has fresh copies of any variables and constants defined in the *Term* class body.

3.5.6 Dot expressions

A dot expression is a request to invoke some function of an object denoted by a term. The form of a dot expression is:

Exp.*att*(*A*$_1$,...,*A*$_n$)

Note there must be no spaces between the dot and the `att` name.

The value of *Exp* is a term – typically a label term or an object reference – instantiated from a Go! class. For example, given the `person` class in program 3.5.1, and the expression:

Program 3.5.1 A `person` class

```
person <~ { name:[]=>string, age:[float]=>number }.

person:[string,integer]@=person.
person(N,D)..{
  name() => N.
  age(Wh) => (Wh-D)/31536000.  -- seconds in a year
}

joe:[]@=person.
joe<=person("joe",1098899684). -- 10/27/04 10:15am
```

```
joe.age(now())
```

this denotes a request to invoke the `age` function within the class identified by the constructor `joe`.

The Go! compiler is able to infer type requirements from occurrences of the dot expression; i.e., given the expression:

```
O.age(now())
```

the compiler *infers* that `O` must be an object with an interface type that includes the function:

`age:[float]=>`T_{new}

and the type of the expression also becomes T_{new}. More formally, it infers the constraint:

`typeOf(O) <~ {age:[float]=>`T_{new}`}`

Note that the interface of an object is restricted to program values. This implies that the only *expressions* that access an object are function calls – there is no direct way of accessing a variable defined within a class.

There are two reasons for this: it is good practice to protect the variables in an object from arbitrary manipulation by programs that merely *use* the object and there are certain subtleties relating to class variables that would be difficult to capture in the general case.

3.5.7 Guarded patterns

A *guarded pattern* takes the form

`Ptn::Goal`

which has the associated type derivation rule:

$$\frac{E \vdash_t Ptn \rightsquigarrow T_P \qquad E \vdash_{safe} Goal}{E \vdash_t Ptn :: Goal \rightsquigarrow T_P} \tag{3.13}$$

Note that the priority of `::` means that in many cases the guarded pattern must be enclosed in parentheses, for example when it occurs as an argument of a function call. However, a major use of guarded patterns is in the left hand sides of equations and paction rules:

```
fact(N)::N>1 => fact(N-1)*N.
```

This use does not require parentheses.

Pragmatically, guarded patterns represent a way of augmenting unification with semantic conditions. For example, the clause:

```
sqrt((N::N>0),S) :- S*S=N.
```

uses a guarded expression in the head of the clause to express the semantic constraint that square roots of negative numbers make no sense.

Guarded patterns are often mappable to more 'normal' patterns with the guard expressed in a normal body goal when they

are used in the heads of clauses; however, they do capture the intended relationship between the guard and the pattern more effectively than a goal which may be interspersed with other goals.

Guarded patterns used in the heads of strong clauses (see Section 4.1 on page 79), equations and action rules (see Chapter 5 on page 93) cannot be mapped into body calls for it is essential that the guards are evaluated before a commitment to use the rule.

Note that the `::` operator is available as a *pattern* operator, not as an expression operator.

3.5.8 Tau pattern

The *tau* pattern is a shorthand for invoking a predicate from a class. Tau patterns take the form:

Var`@`*P*`(`A_1`,...,`A_n`)`

or, in the case that *Var* is not needed, simply:

`@`*P*`(`A_1`,...,`A_n`)`

A pattern of this form matches any object O for which *O*`.`*P*`(`A_1`,...,`A_n`)` holds. It is equivalent to the guarded pattern:

Var`::`*Var*`.`*P*`(`A_1`,...,`A_n`)`

Tau patterns represent a useful generalization for term matching – with the additional benefit that a semantic check rather than a strictly syntactic check may be performed.

Tau patterns are useful when wishing to match against objects that may not have a predetermined constructor. A common pattern is to include in a type interface a predicate that corresponds to a constructor:

```
foo <~ { ..., cons:[symbol,char]{}, ... }
```

Then any program that wishes to 'unify' with `foo` values can use:

```
bar(@cons(S,C)) :- ...
```

The `bar` predicate can be used against any value that implements the `cons` predicate interface.

One very important role for tau patterns is in abstract data types. In `Prolog`, the only way of constructing complex values is by using constructor terms. One problem with that is that the constructor term is, in a sense, a concrete implementation of an abstract concept. Program maintainability issues arise in `Prolog` programs when the abstract concept changes and all the constructor terms used for that concept have to be modified accordingly.

An abstract data type is known by its interface but whose implementation is opaque. When wishing to check for a particular instance, or class of instances of the abstract data type then a tau pattern can be safely used as it does not imply anything about the way the abstract data type is realized.

Using tau patterns makes it possible to structure concepts using classes in a way that insulates the users of those concepts from the details of the implementation. If the concept changes, and the class describing it also changes, then only those references to the class that are directly impacted by the change need to be adjusted: so long as the tau pattern's semantics does not also change.

3.5.9 Parse expression

Parse expressions can be used to apply grammars to streams – normally strings – and return the result of parsing the stream. An expression of the form:

NonTerminal `%%` *Stream*

denotes a request to parse the *Stream* using the grammar *Nonterminal*. *NonTerminal* must be a single argument grammar that is defined over the type of *Stream*. The value returned by the `%%` expression is the value found in *NonTerminal*'s single argument.

In effect, a parse expression it is equivalent to a guarded expression where the guard involves parsing the stream:

`(`*X* `:: NonTerminal(`*X*`) --> `*Stream*`)`

NonTerminal itself must be defined using grammar rules (see Chapter 6 on page 109).

The parse of *NonTerminal* only succeeds if the *NonTerminal* grammar successfully parses the entire contents of *Stream*. A variation of the grammar expression is useful for those cases where it is not necessary to parse the whole stream:

`NonTerminal %% Stream ~ Remainder`

In this case *Remainder* is unified with the remaining portion of *Stream* – assuming that *NonTerminal* successfully parses some part of the stream.

For example, the standard `floatOf` grammar (available from the standard package `go.stdparse`) parses strings and 'returns' in its single argument a floating point number. This grammar can be used to parse strings into floating point values:

```
X = floatOf%%"3.14"
```

3.5.10 Valof expressions

A *valof expression* is used to compute expressions whose values depend on a series of actions rather than applying operators to sub-expressions. A valof expression is written:

`valof{` A_1`;...;`A_{i-1}`;valis` *Ex*`;`A_{i+1}`;...;`A_n`}`

The `valis` action may occur anywhere within the body of the `valof`, it denotes the *value* of the `valof` expression. Typically the `valis` action is placed at the end of the action sequence. There may be more than one `valis` action in a `valof` body; however, all executed `valis` actions must all agree on their value as well as their type. In all cases, the `valof` expression terminates when the last action has completed.

Go! requires that the `valis` action is 'visible' in the action sequence: at least one of the actions A_i in the body of the `valof` must be a `valis` action.

The type of a `valof`/`valis` expression is the type of the expression evaluated by the `valis` action(s). If there is more than one `valis` in the body the they must all return expressions of the same type; and, if more than one is executed, they must also agree on their value.

The type derivation rule for `valof`/`valis` is:

$$\frac{\mathrm{E} \vdash_t Ex \rightsquigarrow T_{Ex} \qquad E \vdash_{safe_A} A_1; \ldots; A_{i-1}; A_{i+1}; \ldots; A_n}{\mathrm{E} \vdash_t \texttt{valof\{A}_1\texttt{;...;A}_{i-1}\texttt{;valis } \textit{Ex}\texttt{;A}_{i+1}\texttt{;...;A}_n\texttt{\}} \rightsquigarrow T_{Ex}} \tag{3.14}$$

3.5.11 Spawn Sub-thread

A *spawn expression* is used to spawn an action as a sub-thread. The form of a `spawn` expression is:

`spawn { `*`Action`*` }`

The value of a `spawn` is a `thread` value that represents the handle of the sub-thread created. The type inference rule for `spawn` is:

$$\frac{E \vdash_{safe_A} Action}{\mathrm{E} \vdash_t \texttt{spawn\{}\textit{Action}\texttt{\}} \rightsquigarrow \texttt{thread[]}} \tag{3.15}$$

The sub-thread executes its action independently of the invoking thread; and may terminate after or before the 'parent'.

3.5.12 Exception recovery expression

An *exception recovery expression* is one which includes a 'handler' for dealing with any run-time exceptions that may arise. The form of such an expression is:

`Ex` `onerror (`P_1 `=>` E_1 `|` `...` `|` P_n `=>` E_n`)`

In such an expression, the types of Ex, E_i should all agree, and the types of P_i is of the standard error type – `exception`:

$$\frac{\mathrm{E} \vdash_t Ex \rightsquigarrow T_E \qquad \mathrm{E} \vdash_t E_i \rightsquigarrow T_E \qquad \mathrm{E} \vdash_t P_i \rightsquigarrow \texttt{exception}}{\mathrm{E} \vdash_t \textit{Ex}\ \texttt{onerror (}P_1\ \texttt{=>}\ E_1\ \texttt{|}\ \ldots\ \texttt{|}\ P_n\ \texttt{=>}\ E_n\texttt{)} \rightsquigarrow T_E}$$

(3.16)

A typical example of an error recovery expression guards a call to a function and returns a fail-safe value if a run-time error is raised:

```
fact(X) onerror (error(_,'eINVAL') => 0)
```

Semantically, an `onerror` expression always evaluates to the 'head' expression *Ex*; unless a run-time problem arises in that evaluation. In this case, an error exception would be raised (of type `exception`); and the evaluation of *Ex* is terminated and one of the error handling equations is used instead. The first equation in the handler that unifies with the raised error is the one that is used; and the value of the expression as a whole is the value returned by that handler equation.

3.5.13 Raise exception expression

An *raise exception expression* takes the form:

`raise` ***Ex***

Exception expressions do not return a value; instead, the current evaluation is terminated with a raised error exception. The error value `Ex` evaluated by the `raise` expression must be caught by an enclosing `onerror` clause.

Note that the `onerror` clause need not itself be an recovery expression: there are recovery mechanisms for actions, goals and grammar conditions as well. The `rase`d expression is caught by the innermost `onerror` active at the time of the `raise`.

The type rule for a `raise` expression reflects the fact that it does not return a value; the type of a `raise` expression is unconstrained:

$$\frac{\mathrm{E} \vdash_t Ex \rightsquigarrow \texttt{exception[]}}{\mathrm{E} \vdash_t \texttt{raise}\, Ex \rightsquigarrow T_v} \text{ where } T_v \text{ occurs nowhere else} \quad (3.17)$$

3.6 Matching and Unification

As with most logic programming languages the fundation for equality in Go! is *unification*: two terms are equal if they are unifyable.

Added to this notion of equality is the stronger concept of *matchability* (sic). Matching is used in place of unification in a few key areas where it seems more pragmatic: in the heads of equations, action rules and message patterns.

When two terms are *unified*, variables in either term may be *bound* in order to ensure that the two terms can be made identical. When two terms are *matched* only one of the terms may have variables bound. For example, in the match condition:

```
foo(X,Y).=Z
```

only `X` or `Y` may be bound, the matched term `Z` may not be bound, nor may any variables within it be bound. I.e., if it is necessary to bind one or more variables in the matched term in order to make the two terms identical then the match will fail.

The `.=` test above (see Section 4.2.6 on page 84) is a special built-in relational equality that denotes matching as opposed to unification. It succeeds only if the two terms can be made equal *without* binding any variables in the right-hand side.

Goals and Predicates 4

In this chapter we define the legal forms of relational programs and the various forms of predicate query conditions that may occur in Go! programs.

A relational program is defined in terms of clauses. Go! has two forms of clauses: regular clauses and strong clauses. The latter carry an if-and-only-if interpretation; whereas regular clauses are normal horn clause-with-negation clauses.

Predicate query conditions may occur in the bodies of clauses, guard conditions of rules (i.e., after a `::` operator) and in the tests of conditionals.

4.1 Predicates

A *predicate* definition within a package or class body takes the form a sequence of clauses, grouped together by predicate symbol, separated by `.␣`'s. A clause may be an assertion of the form:

Name`(`A_1`,...`A_n`).`

a rule-clause of the form:

Name`(`A_1`,...`A_n`):-`*Goal*`.`

or a *strong clause* of the form:

Name`(`A_1`,...`A_n`)::`*Guard*`:--`*Goal*`.`

It is not permitted to mix strong clauses with other forms of clauses within the same definition; i.e., a predicate is either defined using regular clauses or using strong clauses, but not both.

The type inference rule for clauses is:

$$\frac{\mathrm{E} \vdash_t P \rightsquigarrow \texttt{[}T_1, .., T_n\texttt{]\{\}} \quad \mathrm{E} \vdash_t (A_1, .., A_n) \rightsquigarrow (T_1, .., T_n) \quad E \vdash_{safe} G}{\mathrm{E} \vdash_{ok} \texttt{P(}A_1\texttt{,..,}A_n\texttt{):-}G} \tag{4.1}$$

I.e., a clause is type-safe if the argument patterns of the clause are the same as the type of the predicate and the body of the clause (if non-empty) is also type-safe.

By default, the *modes* of a predicate type are *bidirectional*. This implies that the patterns in the head of the clause are unified with the arguments to a relational query and that the type of the arguments to a relational query must similarly unify with the types of the arguments of the predicate type.

Predicate definitions may appear within a class or at the top-level within a package. In the latter case, the relation defined by the predicate definition is available to all other programs in the same package (or to `import`ing packages if the relation is exported).

4.1.1 Strong clauses

The *strong clause* is a variation on the clause form. Syntactically, strong clauses differ from regular clauses simply by using a 'long arrow' instead of a normal clause arrow:

```
Name(A1,...,An) :-- B
```

or, if a guard is necessary,

```
Name(A1,...,An)::G :-- B
```

The type safety rule for strong clauses is the same as for regular clauses:

$$\frac{\mathrm{E} \vdash_t P \rightsquigarrow \texttt{[}T_1, .., T_n\texttt{]\{\}} \quad \mathrm{E} \vdash_t (A_1, .., A_n) \rightsquigarrow (T_1, .., T_n) \quad E \vdash_{safe} G}{\mathrm{E} \vdash_{ok} \texttt{P(}A_1\texttt{,..,}A_n\texttt{):--}G} \tag{4.2}$$

Where strong clauses differ from regular clauses is in their semantics: a strong clause has an if and only if interpretation: if the head of a strong clause unifies with the call arguments – and all embedded guards within the head are satisfied – then no other clauses in the same program will be considered. In effect, the 'if and only' meaning of a strong clause means that if the head of the clause matches, and if any guards associated with the head are satisfied then no other strong clauses need to be considered.

Operationaly, strong clauses offer shallow backtracking selection of the clauses in a program but do not offer deep backtracking once a clause has been committed to.

It is not permitted to mix strong clauses and regular clauses in the same program. I.e., a predicate definition such as:

```
p('foo') :-- true
p('bar') :- true
```

is not permitted since the `foo` clause is a strong clause but the `bar` clause is a regular clause – even though they may be type compatible.

Note that regular clauses can omit their body arrow (`:-`) if the body is empty:

```
pater('john','julius').
```

is equivalent to

```
pater('john','julius') :- true.
```

However, there is no equivalent assertional form for strong clauses.

4.2 Basic goal queries

There are several kinds of basic *goal query*: true/false, predication and a `logical` variable holding one of `true` or `false`. In addition to the basic goal queries, there are the compound goal queries such as conjunction, disjunction, conditional, negation and one-of. These, and other goal queries, are outlined below in Section 4.3 on page 86 and in Section 4.4 on page 88.

Unlike expressions, there often no obvious result – and therefore result type – associated with goal queries. Type inference rules for predicate queries are primarily rules about the $safe_P$ predicate: a goal query G, in a context E, is defined to be type safe if $E \vdash_{safe} G$.

4.2.1 True/false goal

The *true goal* (which always succeeds) is written as `true`; the *false goal* (which never succeeds) is written `false`. The type inference rules for these goals is:

$$\frac{}{\text{Env} \vdash_{safe} \texttt{true}} \tag{4.3}$$

and

$$\frac{}{\text{Env} \vdash_{safe} \texttt{false}} \tag{4.4}$$

respectively.

4.2.2 Predication

A *predication* consists of a predicate applied to an argument. Typically the argument is a tuple of arguments: `P(`$\texttt{A}_1$`,...,`$\texttt{A}_n$`)`. The type inference rule for predications is:

$$\frac{Env \vdash_t P \rightsquigarrow (t_1, \ldots, t_n)\{\} \qquad Env \vdash_t (A_1, \ldots, A_n) \rightsquigarrow (t_1, \ldots, t_n)}{\text{Env} \vdash_{safe} P(A_1, \ldots, A_n)} \tag{4.5}$$

The normal mode of use for a predicate argument is bidirectional; as a result, the arguments of a predication are unified with the parameter patterns in the clause and the *types* of the arguments of a predication must be equal to the types expected by the relational program.

Where an argument of a predicate type is marked as *input*, then the corresponding actual argument may be a subtype of the expected type. Input arguments to predications, like arguments to a function call, are matched rather than unified against.

4.2.3 Class relative predication

A variation on the regular predication is the 'dot'-predication – or class relative goal query. A goal of the form:

`O.P(A1,...,An)`

denotes a goal that is to be evaluated relative to the class identified by `O`. This object may be either a term denoting a state-free class or an object denoting a stateful entity: there is no distinction made between these two for a call.

Such a condition can be viewed as proving `P(A1,...,An)` relative to the theory identified by `O`.

Note that an explicit class relative goal like this is the point at which the `this` keyword is defined. Throughout the proof of `P(A1,...,An)` the value of `this` is set to O – unless, of course, a sub-goal of the proof is also a class relative goal.

The type inference for class relative goals is similar to a regular predication, generalized appropriately:

$$\frac{Env \vdash_t O <\sim \{P : [t_1, \ldots, t_n]\{\}\} \qquad Env \vdash_t (A_1, \ldots, A_n) \rightsquigarrow (t_1, \ldots, t_n)}{Env \vdash_{safe} O.P(A_1, \ldots, A_n)} \tag{4.6}$$

where $Env \vdash_t O <\sim \{P : [t_1, \ldots, t_n]\{\}\}$ means that we can infer from the context that the interface associated with the type of O includes a predicate of type `(t1,...,tn){}`.

4.2.4 Equality

The `=` predicate is a distinguished predicate that is also available as a goal condition. The form of an equality is:

`A = B`

(We mention this here as the `=` operator is also part of Go!'s syntax.)

An equality is type safe if the two elements have the same type:

$$\frac{Env \vdash_t A \rightsquigarrow T \qquad Env \vdash_t B \rightsquigarrow T}{Env \vdash_{safe} \texttt{A = B}} \tag{4.7}$$

4.2.5 Inequality

The `!=` predicate is satisfied if the left hand side does not unify with the right hand side. The form of an inequality is:

```
A != B
```

An inequality is type safe if the two elements have the same type:

$$\frac{Env \vdash_t A \rightsquigarrow T \qquad Env \vdash_t B \rightsquigarrow T}{Env \vdash_{safe} \texttt{A != B}} \tag{4.8}$$

4.2.6 Match test

The `.=` predicate is a distinguished predicate that mirrors the kind of *matching* that characterises the left hand sides of equations and other rules. The form of a match test is:

```
P .= T
```

A match test is type safe if the right hand element's type is a subtype of the left hand side pattern:

$$\frac{Env \vdash_t P \rightsquigarrow T_P \qquad Env \vdash_t E \rightsquigarrow T_E \qquad T_E \preceq T_P}{Env \vdash_{safe} \texttt{P .= T}} \tag{4.9}$$

A match test is similar to a unifyability test with a crucial exception: the match test will *fail* if unification of the pattern and expression would require that any unbound variables in the expression become bound. I.e., the match test may bind variables in the left hand side but not in the right hand side.

This can be very useful in situations where it is known that the 'input' data may have variables in it and it is not desireable to side-effect the input.

4.2.7 Identicality test

The `==` predicate is satisfied if the two terms are 'already' equal – without requiring any substitution of terms for variables. The form of a identicality test is:

```
A == B
```

An identicality test will fail if unification of the pattern and expression would require that any unbound variables in either the pattern or the expression become bound.

An identicality test is type safe if the two elements have the same type:

$$\frac{Env \vdash_t A \rightsquigarrow T \qquad Env \vdash_t B \rightsquigarrow T}{Env \vdash_{safe} \texttt{A == B}} \tag{4.10}$$

4.2.8 Element of test

The `in` predicate is Go!'s list membership test. It is completely definable as a normal Go! program:

```
(in):[T,list[T]]{}.
X in [X,..._].
X in [_,..Y] :- X in Y.
```

but due to its importance in other aspects of Go! (such as within the bounded set abstraction (see Section 3.5.3 on page 68)) it is defined to be part of the language.

The `in` predicate is satisfied if the first argument term is unifiable with an element of the second argument list.

The `in` test is type safe if the type of the second argument is a list of the type of the first argument:

$$\frac{Env \vdash_t A \rightsquigarrow T \qquad Env \vdash_t B \rightsquigarrow \texttt{list[}T\texttt{]}}{Env \vdash_{safe} \texttt{A in B}} \tag{4.11}$$

4.2.9 Sub-class of goal

The `<=` goal condition is used to verify that a given object expression is an 'instance of' a given label term:

```
Ex <= Lb
```

This goal succeeds if the value of the expression *Ex* is an object which is either already unifiable with *Lb*, or is defined by a class that inherits from a class *Sp* that satisfies the predicate

```
Sp <= Lb
```

Otherwise, the goal fails.

4.3 Combination goals

4.3.1 Conjunction

A *conjunction* is a sequence of goals, separated by `,`'s, possibly enclosed in parentheses.

The type inference rule for conjunction is:

$$\frac{\text{Env} \vdash_{safe} \texttt{G}_1 \qquad \ldots \qquad \text{Env} \vdash_{safe} \texttt{G}_n}{\text{Env} \vdash_{safe} \texttt{G}_1\texttt{,}\ldots\texttt{,}\texttt{G}_n} \tag{4.12}$$

4.3.2 Disjunction

A *disjunction* is a pair of goals, separated by `|`'s. It is good practice to parenthesize a disjunction – to surround the disjunction with parentheses. This is because the priority of the `|` operator is higher than the priority of the normal conjunction operator: `,`.

The type inference rule for disjunction is similar to that for conjunction:

$$\frac{\text{Env} \vdash_{safe} \texttt{G}_1 \qquad \text{Env} \vdash_{safe} \texttt{G}_2}{\text{Env} \vdash_{safe} \texttt{(G}_1\texttt{|G}_2\texttt{)}} \tag{4.13}$$

4.3.3 Conditional

A *conditional* is a triple of goals; the first goal is a test, depending on whether it succeeds either the 'then' branch or the 'else' branch is taken. Conditionals are written:

`(`T`?`G_1`|`G_2`)`

This can be read as:

> if T succeeds, then try G_1, otherwise try G_2.

Only one solution of T is attempted; i.e., it is as though T were implicitly a one-of goal.

The relative precedence of operators requires that the conditional is enclosed in parentheses.

The type inference rule for conditionals is:

$$\frac{\text{Env} \vdash_{safe} \texttt{T} \qquad \text{Env} \vdash_{safe} \texttt{G}_1 \qquad \text{Env} \vdash_{safe} \texttt{G}_2}{\text{Env} \vdash_{safe} \texttt{T?G}_1\texttt{|G}_2} \tag{4.14}$$

4.3.4 Negation

A *negation* for is a negated goal, prefixed by the operator `\+`. Go! implements negation in terms of failure to prove positive – i.e., it is negation-by-failure [**?**].

The type inference rule for negation is straightforward:

$$\frac{\text{Env} \vdash_{safe} \texttt{G}}{\text{Env} \vdash_{safe} \texttt{\textbackslash+G}} \tag{4.15}$$

4.3.5 One-of

A *one-of* goal is a goal for which only one solution is required. A one-of goal suffixed by the operator `!`.

The type inference rule for one-of is similar to that for negation:

$$\frac{\text{Env} \vdash_{safe} \texttt{G}}{\text{Env} \vdash_{safe} \texttt{G!}} \tag{4.16}$$

4.3.6 Forall

A *forall* goal takes the form:

`(`G_1 `*>` G_2`)`

Such a goal is satisfied if every solution of G_1 implies that G_2 is satisfied also. For example, the condition:

```
(X in L1 *> X in L2)
```

tests that for every possible solution to `X in L1` leads to `X in L2` being true also: i.e., that the list `L1` is a subset of the list `L2`

The relative precedences of operators requires that `*>` goals are enclosed in parentheses – particularly if they are surrounded by other goals in a clause.

The type inference rule for `*>` is also straightforward:

$$\frac{\text{Env} \vdash_{safe} \texttt{G}_1 \qquad \text{Env} \vdash_{safe} \texttt{G}_2}{\text{Env} \vdash_{safe} \texttt{G}_1\texttt{*>}\texttt{G}_2} \tag{4.17}$$

4.4 Special goals

As with expressions (see Section 3.5 on page 67), there are a number of special forms of goal. These allow a goal to be determined as a result of performing an action, a parse of a stream and allow goals to include exception recovery.

4.4.1 Action goal

An action goal is one which involves the execution of an action to succeed. It is analogous to the `valof` expression (see Section 3.5.10 on page 75). The form of an `action` goal is:

`action{` A_1`;...;`A_{i-1}`;valis` Ex`;`A_{i+1}`;...;`A_n`}`

where Ex is one of `true` or `false`. If `false` then the `action` goal is also false, if `true` then it is true.

Typically, if present, the `valis` action is placed at the end of the action sequence. There may be more than one `valis` action in a `action` body; however, all *executed* `valis` actions must all agree on their value as well as their type. In all cases, the `action` expression terminates when the last action has completed.

If there is no `valis` action within the sequence, then the `action` goal *succeeds* – i.e., as though there were a `valis true` action at the end of the action sequence.

The type derivation rule for the `action` goal is:

$$\frac{\text{Env} \vdash_t Ex \rightsquigarrow \texttt{logical} \qquad \text{Env} \vdash_{safe_A} \texttt{G}_1 \ \ldots \ \text{Env} \vdash_{safe_A} \texttt{G}_n}{\text{Env} \vdash_{safe} \texttt{action\{G}_1\texttt{;...;valis } Ex\texttt{;...;G}_n\texttt{\}}} \tag{4.18}$$

4.4.2 Delayed goal

A delay goal is associated with a variable that must be instantiated before the goal condition may be attempted. If the variable is not instantiated then the condition is suspended, and remaining conditions are attempted. If the variable becomes instantiated then the delayed goal will be attempted.

The form of the delayed goal is:

Var @@ Condition

The type inference rule for delayed goals is straightforward

$$\frac{\text{Env} \vdash_{safe} \texttt{G} \qquad \text{Env} \vdash_t V \rightsquigarrow T_V}{\text{Env} \vdash_{safe} \texttt{V@@G}} \tag{4.19}$$

Note that if the variable is never instantiated, then the delayed goal will not be attempted.

Delayed goals are an effective means of implementing constraint propagation.

4.4.3 Exception handler

A goal may be protected by an exception handler in a similar way to expressions. An `onerror` goal takes the form:

```
Goal onerror
 (P1 :- G1
 | ...
 | Pn :- Gn)
```

In such an expression, *Goal*, G_i are all goals, and the types of P_i is of the standard error type `exception`:

$$\frac{\text{Env} \vdash_{safe} \mathit{Goal} \qquad \text{Env} \vdash_{safe} G_i \qquad \text{Env} \vdash_t P_i \rightsquigarrow \texttt{exception}}{\text{Env} \vdash_{safe} \mathit{Goal}\ \texttt{onerror}(P_1\texttt{:-}G_1 \texttt{|} \ldots \texttt{|}\, P_n\texttt{:-}G_n)} \tag{4.20}$$

Semantically, an `onerror` goal has the same meaning as *Goal*; unless a run-time problem arose in the evaluation. In this case, an exception would be raised (of type `exception[]` – see section 21.1.1 on page 283); and the evaluation of *Goal* is terminated and one of the error handling clauses is used instead. The first clause in the handler that unifies with the raised exception is the one that is used; and the success or failure of the protected goal depends on the success or failure of the goal in the error recovery clause.

4.4.4 Raise exception goal

The `raise` goal neither succeeds nor fails. Instead it raises an error which should be 'caught' by an enclosing `onerror` form.

The argument of an `raise` goal is a `exception` expression:

$$\frac{\text{Env} \vdash_t Er \rightsquigarrow \texttt{exception}}{\text{Env} \vdash_{safe} \texttt{raise}\ Er} \tag{4.21}$$

The effect of a `raise` goal is to terminate the current computation and to send the *Er* exception to the nearest enclosing exception handler – which may be an exception handling goal (see section 4.4.3 on the previous page), but is not required to be.

4.4.5 Grammar goal

A grammar goal is an invocation of a grammar rule. There are two forms of the grammar goal, the simple form denotes a request to parse an entire stream:

```
(Grammar --> Stream)
```

This goal succeeds if the *Grammar* completely parses the *Stream*. Note that it is the responsibility of the *Grammar* to consume any leading and trailing 'spaces' (if the stream is a `string`). The *Grammar* may be a single call to a grammar non-terminal; or it may be a sequence of grammar non-terminals and terminals. In the latter case the sequence will need to be enclosed in parentheses.

The second form of grammar goal allows for a partial parse of the stream:

```
(Grammar --> Stream ~ Remainder)
```

This form of grammar goal succeeds if *Grammar* parses the *Stream* up to – but not including – the *Remainder* stream. *Remainder* must be a proper tail segment of the *Stream*. The simple form of grammar goal is equivalent to:

```
(Grammar --> Stream ~ [])
```

A grammar goal is type safe if the grammar is a grammar of the appropriate type:

$$\frac{E \vdash_{safe_G} G \rightsquigarrow S \qquad E \vdash_t Ex \rightsquigarrow S \qquad E \vdash_t R \rightsquigarrow S}{E \vdash_{safe} \texttt{(}G\texttt{ --> }Ex\texttt{ ~ }R\texttt{)}} \tag{4.22}$$

Procedures and Actions

5

Actions are central to any significant computer application; and Go! supports this reality. The behavioral component of a Go! program should be written using Go!'s action rule notation.

Actions may only be invoked in a limited set of circumstances and action rules look different to clauses. In addition, action rules may have access to system resources – such as the file and communications systems – which are not immediately available to predicate programs or function programs.

Go!'s action rules represent the primary tool for constructing behaviors. An *action procedure* consists of one of more action rules occurring contiguously in a class body or package separated `.␣` operator:

name`(`*Ptn*$_{11}$`,...,`*Ptn*$_{1n}$`) :: `*G*$_1$`-> `*Action*$_1$`.`
`...`
name`(`*Ptn*$_{k1}$`,...,`*Ptn*$_{kn}$`) :: `*G*$_k$` -> `*Action*$_k$`.`

where Ptn_i are pattern terms, G_i are guards (goal query conditions) and $Action_i$ are legal actions.

In a similar fashion to equations, action rules are applied in turn – from first to last in the action program. The first action rule that matches – both the argument and the guard – is used to reduce the call. No further action rule will be considered once a rule has successfully matched the call.

By default, the mode of use of an argument for an action rule in *input*. This means that the corresponding pattern is matched rather than being unified. However, by setting the mode of an action procedure's type to bidirectional or output, then the action rule can be used to return a result.

The type inference rules associated with actions are mostly about the $E \vdash_{safe_A} A$ predicate: an action A, in a context E, is type safe if $E \vdash_{safe_A} A$.

The key type inference rule for action rules is:

$$\frac{\mathrm{E} \vdash_t \mathtt{P} \rightsquigarrow \mathtt{[T_1,...,T_n]*} \quad \mathrm{E} \vdash_t \mathtt{(A_1,...,A_n)} \rightsquigarrow (T_1, \ldots, T_n) \quad E \vdash_{safe} G \quad E \vdash_{safe_A} A}{\mathrm{E} \vdash_{ok} \mathtt{P(A_1,...,A_n)\ ::\ G\ ->\ A}} \tag{5.1}$$

which simply declares that if a procedure type is known, and the types of the arguments are consistent with that type, any guard present is type safe and the actions in the body are type safe then the rule is type consistent.

5.1 Basic actions

Note that actions are not permitted to fail. If an action does cause a failure – for example by some internal goal query failing – then an unexpected failure – `'eFAIL'` – `error` exception will be `raise`d.

Apart from in action rules' bodies, actions can also be found in other kinds of rules – for example, actions can be embedded in the bodies of `valof` and `spawn` expressions.

5.1.1 Empty action

The empty action is written simply as an empty pair of braces: {}; and has no effect.

$$\frac{}{E \vdash_{safe_A} \{\}} \tag{5.2}$$

Empty actions are required for the base cases of recursive action procedures and for conditional actions where one or more of the arms of the conditional is empty.

5.1.2 Equality

An equality *action* has no effect other than to ensure that two terms are equal. Typically, this is done to establish the value of an intermediate variable:

```
Ex1 = Ex2
```

As with equality goals, an equality action is type safe if the types of the two expressions are the same:

$$\frac{Env \vdash_t Ex_1 \rightsquigarrow T_E \qquad Env \vdash_t Ex_2 \rightsquigarrow T_E}{\mathrm{E} \vdash_{safe_A} \texttt{Ex}_1 \texttt{ = Ex}_2} \tag{5.3}$$

Note, however, that unlike equality conditions, an equality action should not fail. A failed equality action will result in an `eFAIL` exception being raised.

5.1.3 Variable assignment

A variable assignment action assigns a new value to a re-assignable variable. It takes the form:

```
V := Ex
```

Note that V must be declared as a re-assignable variable in a package or class body; and that the assignment action must be *inside* the scope of the variable.

In addition to the scope restriction, the new value of the variable – *Ex* – must be *ground*. Any attempt to assign a variable to a non-ground value will cause an `'eINSUFARG'` error to be raised.

The type of an expression assigned to a variable must be a sub-type of the variable's type:

$$\frac{Env \vdash_t V \rightsquigarrow T_V \qquad Env \vdash_t Ex \rightsquigarrow T_E \qquad T_E \preceq T_V}{\mathrm{E} \vdash_{safe_A} \texttt{V := Ex}} \tag{5.4}$$

Note that although the re-assignable variable's value is always ground, the standard packages `dynamic` (see Section 13.3 on page 208) and `cell` (see Section 13.1 on page 202) do permit non-ground values to be stored.

5.1.4 Invoke procedure

The procedure invoke action acts as a call to an action procedure – which itself should be defined by action rules.

The rules for type checking the argument of an action procedure call depend on the modes of the arguments – as defined in the type of the procedure itself. Assuming that the type of a called procedure is known:

$$Env \vdash_t P \rightsquigarrow \texttt{[}t_1\texttt{mode}_1\texttt{,...,}t_n\texttt{mode}_n\texttt{]*} \tag{5.5}$$

then the rules for type checking an argument $\texttt{Ex}_i$ depends on the $mode_i$ associated with argument i:

input This is the default mode for an action procedure. An input argument is correctly typed if its type is a subtype of (or same as) the required type:

$$\frac{Env \vdash_t Ex_i \rightsquigarrow T_i \qquad T_i \preceq t_i}{\mathrm{E} \vdash_{ok} \texttt{Ex}_i} \tag{5.6}$$

`input` arguments are matched against: passing a variable into an action procedure in an input argument position will never cause that variable to become further instantiated.

In fact, attempting to match a non-variable pattern against a variable input will cause the match to *fail*.

super input As with `input` moded arguments, the input argument is correctly typed if its types is a subtype of the required type:

$$\frac{Env \vdash_t Ex_i \rightsquigarrow T_i \qquad T_i \preceq t_i}{\mathrm{E} \vdash_{ok} \texttt{Ex}_i} \tag{5.7}$$

`super input` arguments are also matched against. However, the additional run-time aspect of `++` moded arguments is that the action procedure will *suspend* if the actual argument is a variable.

output or bidirectional arguments must have the *same* type as denoted in $\texttt{t}_i$:

$$\frac{Env \vdash_t Ex_i \rightsquigarrow t_i \qquad T_i \preceq t_i}{\mathrm{E} \vdash_{ok} \texttt{Ex}_i} \tag{5.8}$$

`bidirectional` arguments are unified against. This means that the incoming argument may be side-effected as a result – if the rule pattern is more specific than the actual argument.

`output` arguments must be variable at the point of the call. The intention being that the called action procedure is responsible for determining its value. It is not guaranteed that the action procedure will instantiate the variable, only that it may.

Passing a non-variable to an output argument is considered an error, and the action procedure will fail to apply – resulting in an error exception being raised.

5.1.5 Class relative invocation

A variation on the regular invoke action is the 'dot'-invocation – or class relative action. An action of the form:

O`.P(`A_1`,...,`A_n`)`

denotes that the action:

`P(`A_1`,...,`A_n`)`

is to be executed relative to the class program identified by the term O.

The type inference process is not always able to infer the type of an object in a class relative action call.

Other than the source of the type information for the called action procedure, the type inference rules for a dot-invocation are the same as for the unadorned procedure invocation (see Section 5.1.4 on the facing page).

5.2 Combination actions

Combination actions are, as the name suggests, actions composed of other actions. Go! has a similar range of action combiners to other languages; although the forms may be a little different – particularly for iteration.

5.2.1 Action sequence

A sequence of actions is written as a sequence of actions separated by semi-colons:

```
A1;A2;...;An
```

The actions in a sequence are executed in order.

A sequence is type safe if all the actions are also type safe:

$$\frac{E \vdash_{safe_A} A_1 \quad \ldots \quad E \vdash_{safe_A} A_n}{E \vdash_{safe_A} A_1; \ldots; A_n} \tag{5.9}$$

5.2.2 Goal action

The goal action allows a goal to take the role of an action. It is written as the goal surrounded by braces:

```
{ G }
```

A goal is a type safe action if it is a type safe goal:

$$\frac{E \vdash_{safe} G}{E \vdash_{safe_A} \{G\}} \tag{5.10}$$

The G is expected to *succeed*; if it does not then an `'eFAIL'` error exception will be raised. In keeping with this, only the first solution to the G will be considered – i.e., it is as though the G were a 'one-of' G.

5.2.3 Conditional action

The conditional action allows a programmer to specify one of two (or more) actions to take depending on whether a particular condition holds.

A *conditional* is a test goal; depending on whether it succeeds either the 'then' action branch or the 'else' action branch is taken. Conditionals are written:

```
...;(T?A1|A2);...    -- Note the parentheses
```

This can be read as:

if T succeeds, then execute A_1, otherwise execute A_2.

Only one solution of T is attempted; i.e., it is as though T were implicitly a one-of goal.

Note that the precedences of operators is such that conditional actions must be surrounded by parentheses:

```
action(A,B) ->
    ( A > B ?
      stdout.outLine("A is bigger")
    | stdout.outLine("B is bigger or equal")
    ).
```

The type inference rule for conditional actions is:

$$\frac{Env \vdash_{safe} \texttt{T} \qquad Env \vdash_{safe_A} \texttt{A}_1 \qquad Env \vdash_{safe_A} \texttt{A}_2}{Env \vdash_{safe_A} \texttt{T?A}_1\texttt{|A}_2} \tag{5.11}$$

5.2.4 Forall action

The forall action is an iteration action that repeats an action for each solution to a predicate condition.

The form of the forall action is:

T`*>`*A*

This can be read as:

For each answer that satisfies T execute A.

The type inference rule for forall actions is:

$$\frac{Env \vdash_{safe} \texttt{T} \qquad Env \vdash_{safe_A} \texttt{A}}{Env \vdash_{safe_A} \texttt{T *> A}} \tag{5.12}$$

5.2.5 Case analysis action

The `case` action is a generalization of the conditional action supporting a range of possibilities. The form of the `case` action is:

`...;case` *Exp* `in (`*Ptn*$_1$ `->` *Act*$_1$ `| ...|` *Ptn*$_n$ `->` *Act*$_n$`);...`

The `case` action works by first of all evaluating the *Exp*ression, then then *matching* the value against the patterns Ptn_i in turn until one of them matches.

In fact, `case` can be given a semantics in terms of an auxiliary procedure. If $V_1,\ldots,V_k$ are the free variables in Ptn_i, Act_j, then a `case` action as above is equivalent to:

```
...;ActProc123(Exp,V1,...,Vk);...

ActProc123(Ptn1,V1,...,Vk) -> Act1.
...
ActProc123(Ptnn,V1,...,Vk) -> Actn.
```

where `ActProc123` is a new identifier not occurring elsewhere in the program.

A `case` action is type safe if the individual arms of the case match the governing expression; specifically if the pattern of case arms is the same as the governing expression's type and the case actions are also type safe:

$$\frac{\mathrm{E} \vdash_t \mathtt{P}_i \rightsquigarrow \mathtt{T}_{Ex} \qquad E \vdash_{safe} G_i \qquad E \vdash_{safe_A} A_i}{\mathrm{E} \vdash_{safe_A} \mathtt{case\ Ex\ in\ (...P}_i\mathtt{::}G_i\ \mathtt{->\ A}_i\mathtt{...)}} \tag{5.13}$$

where

$$\mathrm{E} \vdash_t \mathtt{Ex} \rightsquigarrow \mathtt{T}_{Ex} \tag{5.14}$$

is the type assignment for the governing expression.

5.2.6 `valis` Action

The `valis` action is used to 'export' a value from an action sequence. The argument of a `valis` action is an expression; and the type of the expression becomes the type of the `valof` expression that the `valis` is embedded in:

$$\frac{E \vdash_t X \rightsquigarrow T_X \qquad E \vdash_{safe_A} A_1 \quad \ldots \quad E \vdash_{safe_A} A_n}{E \vdash_t \mathtt{valof\{A_1;...;A}_i\mathtt{;valis\ X;A}_{i+1}\mathtt{;...;A}_n\mathtt{\}} \rightsquigarrow T_X} \tag{5.15}$$

The `valis` action is legal within an `action` goal or `valof` expression's action sequence; it has no valid semantics outside that context. There may be any number of `valis` statements in such a statement sequence; if there are none, or if no `valis` action is executed, then the value of the corresponding `valof` is an unbound variable.

Executing a `valis` action *does not* terminate the action body, furthermore, if more than one `valis` is executed then they must all agree (unify) on their reported value. If different `valis` actions do not agree on their values then an `eFAIL` exception will be raised.

The type of expression evaluate by the `valis` action should reflect its context: if the context is a `valof` expression, then the type of the `valof` expression is the same as the type of `valis` expression. If the context is an `action` goal, then the type associated with the `valis` should be `logical`; i.e., either `true` or `false`.

5.2.7 error handler

An action may be protected by an error handler in a similar way to expressions and goals. An `onerror` action takes the form:

Action `onerror (`P_1 `->` A_1 `|` `...` `|` P_n `->` A_n`)`

In such an expression, *Action*, A_i are all actions, and the types of P_i is of the standard error type `exception`:

$$\frac{E \vdash_{safe_A} Action \qquad E \vdash_{safe_A} A_i \qquad \mathrm{E} \vdash_t \mathtt{P}_i \rightsquigarrow \texttt{exception}}{\mathrm{E} \vdash_{safe_A} Action\ \texttt{onerror (}P_1\ \texttt{->}\ A_1\texttt{|...|}P_n\ \texttt{->}\ A_n\texttt{)}} \tag{5.16}$$

Semantically, an `onerror` action has the same meaning as *Action*; unless a run-time problem arises in the execution of the action. In this case, an error exception would be raised (of type `exception`); and the execution of *Action* is terminated and one of the error handling clauses is used instead. The first clause in the handler that unifies with the raised exception is the one that is used.

Note that it is quite possible for an exception to have been raised within the evaluation of *Action*.

5.2.8 `raise` action

The `raise` action raises an exception which should be 'caught' by an enclosing `onerror` form. The enclosing `onerror` form need not be an action: it may be within an `onerror` expression or goal.

The argument of a `raise` action is a `exception` expression:

$$\frac{\mathrm{E} \vdash_t Er \rightsquigarrow \texttt{exception}}{E \vdash_{safe_A} \texttt{raise}\ Er} \tag{5.17}$$

5.3 Threads

Go! is a multi-threaded and distributed programming language – in that there is support for `spawn`ing new threads of activity and there is support for coordinating threads.

Go! threads share program code and instances of classes; but do not share logical variables. In that sense, Go!'s thread system is not a replacement for coroutining or parallel execution of programs.

Go! has support for thread synchronization – see Section 5.3.2 on the next page – and there are also some higher-level support for thread *coordination* by means of message passing.

5.3.1 `spawn` Sub-thread

The `spawn` action spawns a sub-thread; it is similar to the `spawn` expression – see Section 3.5.11 on page 76 – except that the identity of the spawned sub-thread is not returned in the `spawn` action.

The form of a `spawn` action is:

`spawn` { *`Action`* }

The type inference rule for `spawn` action shows that a `spawn` action is type safe if the embedded actions are:

$$\frac{E \vdash_{safe_A} A_1; \ldots; A_n}{\mathrm{Env} \vdash_{safe_A} \texttt{spawn\{A}_1\texttt{;...;A}_n\texttt{\}}} \tag{5.18}$$

The sub-thread executes its action independently of the invoking thread; and may terminate after or before the 'parent'.

5.3.2 Process Synchronization

All multi-threaded applications need to solve two related but separate problems: sharing resources between the processes and coordination between processes.

The purpose of the `sync` action is to effectively *sequentialize* the actions that access and manipulate a shared resource; that way, the applications programmer can be guaranteed that while one thread is using the resource, other threads are locked out of accessing the resource. That, in turn, makes it easier to ensure that the resource's state is always well-formed.

Although synchronized access is a powerful technique that permits processes to share resources, two important caveats are required: for synchronized access to be effective *all* accesses to the shared resource must be synchronized and it is sadly easy to mis-program synchronized actions and end up with dead-lock situations.

Dead-lock occurs when more than one resource is being `synchr`-onized on, and there is contention amongst the sharing processes – some have one resource and others have the other resource and neither process can proceed. Such deadlocks can be avoided by ensuring that the *order* of synchronization on resources is the same for all processes.

We recommend the use of higher-level mechanisms, such as the message mailbox package `go.mbox` described in Chapter 15 on page 237.

The effect of the `sync` action is to block other `sync`hronized actions – on the same resource – from being entered for the duration of the `sync` action. Thus it can be used to allow multiple threads of activity to safely share resources: without an equivalent of `sync` two threads could simultaneously access and modify a shared object without being aware of each other.

Go! permits synchronization only on statefull objects. Unfortunately, it is not always possible to determine if an expression denoting an object is statefull at compile-time. This can lead to a run-time error exception.

The simplest form of the `sync` action is:

```
sync{ Action }
```

This form of action is valid *within* the definition of a method belonging to the shared object itself. The effect of this action is to ensure that only the current process is currently able to access `this` object in a synchronized mode during the course of executing *Action*.

If another process attempts to execute a `sync` action within the same object then it will be blocked until the *Action* has either terminated normally or terminated via an exception.

It is possible to `sync` on an 'external' object, either within a method or completely outside any classes, using the form:

```
sync(Object){ Action }
```

This version attempts to access the object *Object* in a synchronized way. Again, this will only be possible if neither an internal method, nor any other program on another process has already sucessfully entered a `sync`hronized action.

For example, the following action ensures that the `cell` variable `X` is only modified while the thread has exclusive access to it:[1]

```
...;sync(X){X.set(newValue)};...
```

Multiple choice guarded synchronized The 'full' version of the `sync` action allows for the possibility of multiple guards and a `timeout`. Within a class, this form of `sync` takes the form:

```
sync{
  Guard1 -> Action1
| Guard2 -> Action2
...
} timeout (timeExp -> TimeoutAction)
```

[1] This example is actually redundant as `cell` already implements this functionality.

and when attempting to synchronize an external object the form is:

```
sync(O){
  Guard1 -> Action1
| Guard2 -> Action2
...
} timeout (timeExp -> TimeoutAction)
```

A simple `sync` action

```
sync{ Action }
```

is, in fact, equivalent to:

```
sync(this){ true -> Action }
```

Note that the `timeout` clause is optional; if given the *timeExp* should refer to an absolute time – preferably in the future.

The operational semantics of the full form of `sync` action is somewhat complex. Associated with every `synchronized` object is an internal lock value which is the key structure needed to implement synchronized access. The process of entering and executing a `sync` action follows the algorithm:

1. Attempt to acquire the lock associated with the synchronized object. This attempt has three possible outcomes:
   (a) The attempt is successful, and the thread has the lock.
   (b) The lock is not available, and the requested `timeout` *timeExp* is in the future, in which case the thread suspends waiting for either the timeout to expire or for the lock to become available.
   (c) The lock is not available and the `timeout` has expired; in which case the attempt *fails*. This in turn causes the timeout action to be taken.

   The timeout value is not recalculated; it is calculated once and for all at the start of the `sync` action. The effect is that the `timeout` clause will be activated at a

fixed time no matter how many times the guards are attempted.

Note that the lock is *not* owned by the thread during the execution of the `timeout` action. Therefore, the timeout action should not attempt to modify or access the shared resource. A suitable action for the `timeout` clause is, in fact, to raise an exception.

2. If the lock has been acquired in step 1, then each of the guards associated with the `sync` choice is attempted in order.

3. If one of those guards succeeds, the corresponding action is executed and execution continues with step 5

4. If none of the guards succeeds, the lock is relinquished, and a fresh attempt is made to acquire the lock with a given timeout. This enables other processes to 'step in' and synchronize with the object – with the possibility of somehow adjusting the state of the object so that one of the guards will succeed on a subsequent attempt.

   The relinquishing thread is only rescheduled for execution when *another* thread releases the lock on the object. This ensures that the system is not kept in a busy cycle as it tests the predicates of the guards.

5. On successful completion of the selected synchronized action, the lock is released.

6. Finally, if an error occurs anywhere within the synchronized action that is not handled within the action then the lock is released before 'exporting' the exception to the outer context.

The overall effect of this complex operation is quite simple: one of the guarded actions in the `sync` action will be executed. The precondition being that the shared resource denoted by the lock is available and that the corresponding guard condition is satisfied.

Hint Using `sync` judiciously will enable the programmer to construct *thread-safe* libraries. Recall that all threads that are

`spawn`ed that have access to any read/write variables that are in scope effectively see the same read/variable. Since this can obviously lead to contention problems when two threads attempt to access and/or update a shared read/write variable, a key motivation of the `sync` operator is to prevent this.

A thread-safe library is a program that is written to ensure that any threads which can access shared read/write variables defined within the thread properly 'gate' access to the variables with `sync` statements.

Grammar rules 6

Go! supports a form of definite clause grammar notation which we call Go! grammars. The grammar notation can be used for parsing strings, but can equally be used for parsing (or generating) arbitrary streams of data. The most general form of a logic grammar takes the form:

$\mathtt{L}_1\mathtt{,...,L}_n$ `-->` $\mathtt{R}_1\mathtt{,...,R}_k$

where the $\mathrm{L}_i$and R_j are either *terminals* or *non-terminals*. The distinction is that terminal symbols may appear in the original input stream and non-terminal symbols may not.

The effect of applying a rule such as this is to replace a stream of the form:

$\mathtt{S}_1\mathtt{,...,S}_{i-1}\mathtt{,R}_1\mathtt{,...,R}_k\mathtt{,S}_{i+1}\mathtt{,...,S}_m$

with

$\mathtt{S}_1\mathtt{,...,S}_{i-1}\mathtt{,L}_1\mathtt{,...,L}_n\mathtt{,S}_{i+1}\mathtt{,...,S}_m$

Normally we consider the parsing process one of deriving a stream containing a single non-terminal from the original stream.

However, for a combination of pragmatic and utilitarian reasons, we normally restrict grammar rules to a more simplified form:

$\mathtt{N,T}_1\mathtt{,...,T}_n$ `-->` $\mathtt{R}_1\mathtt{,...,R}_k$

where N is a non-terminal, T_i are all terminal symbols and R_j are either terminals or non-terminals. Either of or both n and k may be zero, but every grammar rule must be 'about' a particular non-terminal.

Like other forms of logic grammars, Go! grammar's non-terminal symbols may take arguments, which are unified rather than simply input. This has great utility from a programmer's perspective as it allows a grammar to simultaneously parse a stream and to construct some kind of parse tree via 'back substitution' of the variables in the grammar rules.

The stream of data that is processed by a grammar rule is typically a `string` – i.e., a list of `char`acters. However, in general, the stream may be represented by any kind of list, or indeed any kind of stream.

Note that type inference applies to the bodies of grammar rules; generally, all the grammar rule conditions must agree on the type of the stream, as must the lookahead. The type of a grammar program is a term of the form:

`[`$\mathtt{T}_1$`,...,`$\mathtt{T}_n$`] --> `$\mathtt{T}_{stream}$

where $\mathtt{T}_i$ are the types of the arguments of the grammar program and $\mathtt{T}_{stream}$ is the type of the stream that the grammar can process.

6.1 Basic grammar conditions

6.1.1 Terminal grammar condition

A *terminal grammar condition* represents a term that is expected in the input stream – it corresponds to a terminal symbol; although Go! grammars may parse streams other than symbol streams. The general form of a terminal grammar condition is a list of terms – each of which represents a separate term that should appear in the input list for the grammar rule to be satisfied.

For grammar rules over strings, a `string` literal may act as a terminal grammar condition. The special case of the empty list, or empty string, is often used to denote a grammar condition which does not consume any input:

```
foo() --> [].
```

The type inference rule for a terminal grammar condition is:

$$\frac{\text{Env} \vdash_t L \rightsquigarrow \mathtt{list[}T\mathtt{]}}{Env \vdash_{safe_G} L \rightsquigarrow \mathtt{list[}T\mathtt{]}} \tag{6.1}$$

where the predicate $Env \vdash_{safe_G} L \rightsquigarrow \texttt{list[}T\texttt{]}$ means

> In the environment *Env* the symbol L is a grammar defined over lists of T

6.1.2 Non-terminal grammar call

A *non-terminal grammar call* is of the form ***nt(Args)*** and consists of the application of a grammar program to some arguments. It represents a 'call' to another grammar program (or even the same in the case of recursive grammars) and succeeds if the called grammar program can successfully parse the appropriate segment of the input stream.

The type inference rule for a non-terminal grammar call is:

$$\frac{E \vdash_t Nt \rightsquigarrow \texttt{[}t_1\texttt{,...,}t_n\texttt{]-->}T_s \quad E \vdash_t \texttt{(}A_1\texttt{,...,}A_n\texttt{)} \rightsquigarrow (t_1, \ldots, t_n)}{E \vdash_{safe_G} \texttt{Nt(}A_1\texttt{,...,}A_n\texttt{)} \rightsquigarrow T_s} \tag{6.2}$$

The default mode for passing arguments to a grammar non-terminal is *bidirectional* – arguments are unified rather than matched. However, it is possible to specify modes in the type of the grammar procedure. In detail, the type rules for type checking individual arguments are the same as for actions, functions and predicates (for example, see Section 5.1.4 on page 96).

6.1.3 Class relative grammar call

The class relative variant of the non-terminal invokes a grammar condition defined within a class.

$\texttt{}O\texttt{.Nt(A}_1\texttt{,...,A}_n\texttt{)}$

denotes a grammar condition where the grammar rules are defined within the class identified by the label term O.

The type inference for class relative grammar conditions is similar to a regular grammar call, generalized appropriately:

$$\frac{Env \vdash_t O \texttt{<-\{Nt:[}t_1\texttt{,..,}t_n\texttt{]-->}T_s\texttt{\}} \quad Env \vdash_t \texttt{(}A_1\texttt{,..,}A_n\texttt{)} \rightsquigarrow \texttt{(}t_1\texttt{,..,}t_n\texttt{)}}{Env \vdash_{safe_G} \texttt{O.Nt(}A_1\texttt{,..,}A_n\texttt{)} \rightsquigarrow T_s} \tag{6.3}$$

6.1.4 Equality condition

An equality definition grammar condition has no effect other than to ensure that two terms are equal. Typically, this is done to establish the value of an intermediate variable:

Ex_1 `=` Ex_2

As with equality goals, an equality grammar condition is type safe if the types of the two expressions are the same:

$$\frac{Env \vdash_t E_1 \rightsquigarrow T_E \qquad Env \vdash_t E_2 \rightsquigarrow T_E}{Env \ \vdash_{safe_G} E_1 \ \texttt{=} \ E_2 \rightsquigarrow T_{new}} \tag{6.4}$$

where T_{new} is a new type variable.

6.1.5 Inequality condition

The `!=` grammar condition is satisfied if the two expressions are *not* unifiable. The form of an inequality grammar condition is:

T_1 `!=` T_2

Like equality conditions, an inequality is type safe if the two elements have the same type:

$$\frac{Env \vdash_t E_1 \rightsquigarrow T_E \qquad Env \vdash_t E_2 \rightsquigarrow T_E}{Env \ \vdash_{safe_G} E_1 \ \texttt{!=} \ E_2 \rightsquigarrow T_{new}} \tag{6.5}$$

where T_{new} is a new type variable.

6.1.6 Grammar predicate condition

A *grammar predicate condition* is a query condition, enclosed in braces – $\{Goal\}$ – represents a predicate or condition to be applied as part of the parsing process. Grammar predicate conditions do not 'consume' any of the string input; nor do they directly influence the type of stream that the grammar rule consumes.

The type inference rule for a grammar goal is:

$$\frac{\text{Env} \vdash_{safe} Goal}{\text{Env} \ \vdash_{safe_G} \{Goal\} \rightsquigarrow T_s} \tag{6.6}$$

where T_s is a new type variable not occurring elsewhere.

6.2 Combined grammar conditions

6.2.1 Disjunction

A grammar *disjunction* is a pair of grammar conditions, separated by `|`'s. The disjunction is required to be parenthesised; since the priority of the `|` operator is higher than the priority of the normal conjunction operator: `,`.

A grammar disjunction succeeds if either arm of the disjunction is able to parse the input stream.

The type inference rule for a grammar disjunction is similar to that for conjunction:

$$\frac{\text{Env} \vdash_{safe_G} \texttt{G}_1 \rightsquigarrow T_S \qquad \text{Env} \vdash_{safe_G} \texttt{G}_2 \rightsquigarrow T_S}{\text{Env} \vdash_{safe_G} \texttt{(G}_1\texttt{|G}_2\texttt{)} \rightsquigarrow T_S} \tag{6.7}$$

6.2.2 Conditional grammar

A *conditional* grammar condition is a triple of a three grammar conditions; the first is a test, depending on whether it succeeds either the 'then' branch or the 'else' branch is taken. Conditionals are written: $T\texttt{?}G_1\texttt{|}G_2$ This can be read as:

> if T succeeds, then try G_1, otherwise try G_2.

Only one solution of T is attempted; i.e., it is as though T were implicitly a one-of grammar condition.

Note that the test may 'consume' some of the input; the 'then' branch of the conditional grammar only sees the input after the test. However, the else branch is expected to parse the entire input.

The type inference rule for conditionals is:

$$\frac{\mathit{Env} \vdash_{safe_G} \texttt{T} \rightsquigarrow T_S \qquad \mathit{Env} \vdash_{safe_G} \texttt{G}_1 \rightsquigarrow T_S \qquad \mathit{Env} \vdash_{safe_G} \texttt{G}_2 \rightsquigarrow T_S}{\mathit{Env} \vdash_{safe_G} \texttt{T?G}_1\texttt{|G}_2 \rightsquigarrow T_S} \tag{6.8}$$

6.2.3 Negated grammar condition

A *negated* grammar condition is a grammar condition, prefixed by the `\+` operator (negation-as-failure). A negated grammar condition succeeds if the grammar condition is not able to parse the input stream. The effect of a successful negated grammar condition is to parse none of the input stream; in effect, the negated grammar condition is a kind of negative look-ahead – it succeeds if the input does *not* start with the negated grammar.

Go! implements negation in terms of failure to prove positive – i.e., it is negation-by-failure [**?**].

The type inference rule for a negated grammar condition is:

$$\frac{\text{Env} \vdash_{safe_G} \texttt{G} \rightsquigarrow T_S}{\text{Env} \vdash_{safe_G} \texttt{\textbackslash+G} \rightsquigarrow T_S} \tag{6.9}$$

6.2.4 Iterated grammar

The iterator grammar condition adresses a common problem with logic grammars: how to encapsulate a sub-sequence; without the relatively tedious requirement of writing a special grammar non-terminal to handle it.

For example, the common definition of a program identifier reads something like:

> the first character must a letter, then followed by an arbitrary sequence of letters and digits

Such a definition is hard to capture in regular logic grammars without an explicit recursion and worse, often requires a cut to get the correct semantics.

A grammar iterator applies a grammar to the stream repeatedly, and returns the result as a list. It is written:

```
Gr * Exp ^ LstVar
```

This grammar succeeds if the grammar *Gr* – which may be any combination of terminals and non-terminals – successfully parses some portion of the stream any number of times. The 'result' is returned in *LstVar* – which consists of a list constructed from *Exp*.

The meaning of a grammar iterator is:

`...,iter`$_{new}$`(LstVar,`F_1`,..,`F_n`),...`

where F_i are the free variables of ***Gr*** and ***Exp*** and `iter`$_{new}$ is a new program defined:

```
iter_new([Exp,..L_x],F_1,..,F_n) --> Gr!,iter_new(L_x,F_1,..,F_n).
iter_new([],F_1,..,F_n) --> [].
```

with the additional property that ***LstVar*** will be bound to the list corresponding to the longest possible repetition of ***Gr***.

The informal rule for identifiers we described above can be captured using the grammar iterator illustrated in program 6.2.1.

Program 6.2.1 A grammar for identifiers

```
identifier(ID([C,..L])) --> letter(C),
      (letter(X)|digit(X))*X^L
```

The grammar iterator captures one of the key essentials of regular expression notation – the star iterator.

Type inference for an iterated grammar condition is fairly straightforward:

$$\frac{\text{Env} \vdash_{safe_G} \textit{Gr} \rightsquigarrow T_S \qquad \text{Env} \vdash_t \textit{Exp} \rightsquigarrow T_E \qquad \text{Env} \vdash_t L \rightsquigarrow \texttt{list[}T_E\texttt{]}}{\text{Env} \vdash_{safe_G} \textit{Gr} \texttt{ * } \textit{Exp} \texttt{ \^{} } L \rightsquigarrow T_S} \tag{6.10}$$

6.3 Special grammar conditions

6.3.1 error handler

A grammar condition may be protected by an error handler in a similar way to expressions and goals. An `onerror` grammar condition takes the form:

G `onerror (`P_1 `-->` G_1 `|` ... `|` P_n `-->` G_n`)`

In such an expression, G, G_i are all grammar conditions, and the types of P_i is of the standard error type `exception`:

$$\frac{E \vdash_{safe_G} \texttt{G} \rightsquigarrow \texttt{T}_S \qquad E \vdash_{safe_G} \texttt{A}_i \rightsquigarrow \texttt{T}_S \qquad \mathrm{E} \vdash_t \mathrm{P}_i \rightsquigarrow \texttt{exception}}{\mathrm{E} \vdash_{safe_A} \texttt{G onerror (P}_1 \texttt{ --> A}_1\texttt{|...|P}_n \texttt{ --> A}_n\texttt{)T}_S} \tag{6.11}$$

Semantically, an `onerror` grammar condition has the same meaning as the 'protected' condition G; unless a run-time problem arose in the evaluation of G. In this case, an error exception would be raised (of type `exception`); and the parse of G is terminated and one of the error handling clauses is used instead. The first clause in the handler that unifies with the raised error is the one that is used; and the success or failure of the protected grammar depends on the success or failure of the selected grammar rule in the error recovery clause.

Error handler protected grammar conditions are especially useful in coping with errors in the source stream. I.e., when parsing a stream which may have syntax errors in it – such as a programming language – then the error protected grammar condition represents one way of recovering from syntax errors. However, error protecting a grammar does not itself handle the processing of the input stream – the error handler's grammar conditions are processing the *same* input as the protected condition itself.

6.3.2 Raise exception

The `raise` exception grammar condition does not parse any input; it terminates processing of the input and raises an exception. If the exception is caught by an `onerror` grammar condition then parsing is continued at that level; otherwise the entire parse is aborted and the exception is caught at a higher level.

The argument of an `raise` grammar condition is a `exception` expression:

$$\frac{\mathrm{Env} \vdash_t Er \rightsquigarrow \texttt{exception}}{\mathrm{Env} \vdash_{safe_G} \texttt{raise } Er \rightsquigarrow T_S} \tag{6.12}$$

where T_s is a new type variable not occurring elsewhere.

Judicious use of the `onerror` grammar form can greatly ameliorate one of logic grammar's greatest practical difficulties: recovering from errors. Logic grammars are based on the normal backtracking behavior of clauses; and this can be a very powerful tool for expressing grammars in a high-level style.

However, backtracking across 'correct' input is perhaps legitimate, backtracking across erroneous input (as in the case of parsing a program with a syntax error in it) can cause a great deal of unnecessary backtracking. Adjusting a normal logic grammar to gracefully handle erroneous input is a tedious and ugly process: typically involving the use of cuts in a Prolog-based system.

By explicitly raising an exception when erroneous input is detected, and by catching it with an appropriate `onerror` clause, we can add error handling and recovery to a Go! grammar in a more accurate and succinct manner.

6.3.3 End of file

The `eof` grammar condition is satisfied only at the end of the input. For example, the grammar rule:

```
tok(TERM) --> ".", eof.
```

is satisfied only if the last character in the string being parsed is a period.

The type inference rule for the `eof` test is:

$$\frac{}{\text{Env} \vdash_{safe_G} \texttt{eof} \rightsquigarrow T_s} \tag{6.13}$$

where T_s is a new type variable not occurring elsewhere.

Logic and Objects 7

Go!'s object oriented notation is based on 'Logic and Objects' ([**?**]) with some simplifications and modifications to incorporate Go!'s type system and the notion of an *instance* as well as a class. It provides a straightforward technique to build large scale systems and to represent knowledge.

The fundamental concept behind Go!'s object notation is the *labeled theory*. A theory is simply a set of facts (presented as rules of various kinds) that is known about some concept. A label is a term that represents a handle on the theory that allows the knowledge contained in multiple theories to interact in structured ways – in effect, the label term might be viewed as the concept identifier and the theory itself a view on what is known about the concept.

Go!'s classes reflect the notational conventions of object oriented programming: theories' knowledge can be inherited from other theories, and theory elements – a.k.a. methods – can be referenced from outside a theory using the theory label.

A *class* is defined with a combination of a *class body* and zero or more *class rules*. Program 7.0.1 on the following page demonstrates a simple example of a labeled theory, where the `bird` class has a class body and a class rule relating birds to `animals`.

7.1 Class types

Like other kinds of programs, a class requires a *type declaration*. The type declaration for the class constructor serves as an introduction to the labeled theory that is associated with the constructor.

Program 7.0.1 A `bird` class

```
birdness <~ {no_of_legs:[integer]{}, mode:[symbol]{}}.

bird:[]@=birdness.
bird<=animal.
bird..{
  no_of_legs(2).
  mode('fly').
}.
```

There are two styles of class definition, corresponding to *state-free* classes and *state-full* classes. The former are very close to regular declarative theories, the latter are intended to capture state as well as knowledge.

The type declaration for a state-free class looks like:

label`:[`$\mathtt{T}_1$`,...,`$\mathtt{T}_n$`] @=` *Type*`.`

This type declaration introduces the constructor *label* as a function symbol for the *Type* – of arity n. If n is zero then the constructor may be written without arguments.

The *Type* expression declares what the type of instances of the class are; in the case of state-free classes this is equivalent to giving the type of the class label term. In the case of a state-full class, the type is the type of objects that are instances of this class.

The *Type* also defines the *interface* of the class – the functions and other elements that must be defined within the class and which are accessible by users of the class. This is because all user-defined types have an interface.

The type declaration for a state-full class is very similar to the state-free class type declaration, albeit with a different operator:

label`:[`$\mathtt{T}_1$`,...,`$\mathtt{T}_n$`] @>` *Type*`.`

Note that, unlike with state-free constructors, a zero-arity state-full constructor must always use the empty argument tuple `()`. I.e., given

```
foo:[]@>SomeType.
```

any instance of `foo` must include the empty argument tuple:

```
foo()
```

Polymorphic classes Go! supports polymorphic classes; however, the polymorphism of a class is reflected in the initial class label given with the class body (and any class rules).

In particular, a class may not be 'more polymorphic' – i.e., polymorphic in additional type variables – than the *Type* it is associated with.

7.2 Class body

Class bodies give the implementation of methods and other exported values and class rules express the inheritance relationships. As with other programs, the elements that make up a given class must be contiguous within a package body.

A class body consists of a structure of the form:

```
label(A1,...,An)..{
  local definitions
}.
```

where

```
label(A1,...,An)
```

is the *class label* and the definitions in

```
{
  local definitions
}.
```

form the *class theta environment*.

7.2.1 Class labels

The label of a class is a term that identifies the class. In a class definition, a *class label* takes the form:

$label(A_1, \ldots, A_n)$

where all of A_i are *variables*. This is also a constructor function – in Go! all constructor functions are class labels.

The class label is the key to understanding Go!'s object notation: class labels denote the set of axioms and other definitions in much the same way that predicate symbols denote relations and function symbols denote functions. The main distinction between such symbols and class labels is that the latter may themselves be structured, and that class labels identify *sets* of relations, functions and so on.

The types of the arguments of the label are matched up with the types of the arguments in the label's type declaration. Thus, $\mathtt{A}_i$ has type $\mathtt{T}_i$.

7.2.2 Constructors, patterns and modes of use

Semantically, constructors are a kind of *function*. State-free constructors are *bijections* (i.e., one-to-one and onto) where state-full constructors are not.

The critical property of a bijection is that it is guaranteed to have an *inverse*; which leads to their use in *patterns*. When we use a state-free constructor to match against an input term, we are effectively using the constructor function's inverse to recover the arguments of the expression. On the other hand, because state-full constructors do not have inverses, they cannot be used in patterns.

Because of the inherently bi-directional nature of state-free constructor functions, they are *not* associated with modes of use – it is always bi-directional. This also means that the type of an argument of a constructor function must be *equal* to the type declared for that argument – it may not be a sub-type or a super-type of the declared argument type.

However, a state-full constructor's default mode of use is *input*; much like a regular function. The other modes of use are

theoretically available for state-full constructors but they are not all that useful – because the parameters of the label in a class definition must consist of variables.

Recall that an input-moded parameter is permitted to have an actual argument that is a strict sub-type of the expected type. This is not the case for either bidirectionally-moded parameters nor output-moded parameters.

Although the rules for legal state-free and state-full classes are different – they can contain different kinds of definitions – a constructed value is *accessed* in the same way whether it is defined in a state-free or a state-full manner.

7.2.3 Class theta environment

The body itself consists of a set of definitions – called the class *theta environment*. There are slightly different constraints for state-free classes and state-full classes: the former is permitted only to contain rules of various kinds, whereas a state-full class body may also contain variables and constants.

Both state-free and state-full class theta environments may contain inner classes, but a state-free class may not contain any state-full inner classes.

Finally, both types of classes may contain inner type definitions; although any *exported* program may not reference the inner type in the types of its arguments or returned result.

Any variables mentioned in the constructor arguments are in scope across the entire class body – as are special variables denoting the super classes and `this` which is the finally constructed object.

The class theta environment must contain definitions for each of the methods declared in the `Type`'s interface – except for definitions that are inherited. There is no equivalent, in Go!, of the *abstract class* found in some object oriented languages.

Private definitions Any variables or other definitions that are defined within a state-full class body are *private* to the class: they

may not be referenced either by any sub-class or by any external query.

The fact that only rule types may to be included in a type interface prevents variables being referenced directly externally.

Of course, any additional programs must be *declared* – just as programs in the package are declared.

7.2.4 Special elements in state-full class theta environments

A state-full class may include, in addition to those elements permitted in a state-free class body, object *constants* and object *variables*.

Object Constant

An *object constant* is a symbol that is given a fixed value within a class body. Object constants are introduced using equality statements within the class body. Note that constants are, by definition, restricted to being *private* to the class body in which they are defined.

Rules for evaluation An object constant is evaluated when an instance of the class is created – when its constructor function is invoked.

Groundedness Object constants may not be nor include unbound variables in their value.

Object Variable

An *object variable* is a symbol that is given a reassignable value within a class body. Object variables are introduced using `:=` statements within the class body. Variables can be re-assigned by rules – primarily action rules – that are located *within* the class body that they are defined in.

Like constants, object variables are always private to the class body: they may not be referenced either by any sub-class or by any external query.

Note that object variables and constants also require type declarations; which, in the case of constant and variable definitions, can be included in the defining statement:

```
...{
  iX:integer := 0.
...
}
```

is equivalent to:

```
...{
  iX:integer.
  iX := 0.
...
}
```

Like object constants, object variables may not be unbound, nor may their values contain any unbound elements: they must be *ground*.

The `queue` class, shown in Program 7.2.1 on the next page shows a variable being reassigned by the action rules for `push` and `pull`. Should there be a sub-class of `queue`, no rules defined within that sub-class are permitted to re-assign the `Q` variable.

the `queue` type is explicitly polymorphic, and the `queue` class is similarly polymorphic – `queue`s can be queues of any kind of value.

Note that the variable `Q` in the `queue` class body has a type declaration associated with it. Note also that although it is declared to be a `list`, its not further constrained. In fact, the other occurrences of `Q` constrain its type to be the same as that of the label argument `I`. We could have made this explicit by annotating the argument:

```
queue(I:list[t])..{
  Q : list[t] := I.
```

Program 7.2.1 A simple `queue` class

```
queue[T] <~ { push:[T]*, pull:[T]* }.

queue:[list[t]] @> queue[t].
queue(I)..{
  Q : list[_].
  Q := I.

  push(e) -> Q := Q<>[e].

  pull(e) -> [e,..R].=Q; Q := R.
}
```

```
  ...
}
```

The use of the same type variable `t` as in `queue`'s type declaration is coincidence, the same effect would be had by using a type variable `alpha`. The annotation in the label – together with the type declaration for `Q` – are there to bind the type of `Q` to that of `I`.

Rules for evaluation An object variable is initialized when an instance of the class is created.

The order of evaluation between different variables and constants is not fixed by their order of appearance within the class theta environment. Instead, they will be evaluated in such a way that constants and variables are evaluated *after* any variables and constants they depend on.

Such an ordering is not possible in general, in which case, the result is not defined (the compiler may issue an error in this case).

Static initialization

For those situations where the initialization of an object is more involved, Go! supports a special initialization construct within class bodies. An *`InitAction`* of the form:

```
label..{
  ...
  ${
    InitAction
  }
}
```

is executed – after the initialization of variables and constants defined in the class. This *InitAction* may perform any action that is legal within the context of the class. If a class inherits from another class then the super-classes initialization actions are performed before the sub-class'es initialization actions.

7.3 Inheritance and Class rules

Inheritance is expressed via the use of *class rules*. A class rule is a rule that defines how a sub-class inherits from a super-class:

label(*A*$_1$,...,*A*$_n$) <= *mabel*(*E*$_1$,...,*E*$_m$).

where the parentheses may be dropped if n is zero.

For example, to denote the fact that birds are animals, we can use the class rule:

```
bird <= animal.
```

From a labeled theoretic perspective, the (informal) semantics of a class rule such as this is:

> all the consequences of the `animal` theory are also consequences of the `bird` theory.

reflecting the intuition that inheritance is specialization, and specialization generally consists of refining and adding to knowledge.

A class rule has the effect of defining within the scope of the sub-class all the elements – except types – that are present in the super-class. However, if an element is defined both within the super-class and the sub-class, then the sub-class'es definition *overrides* the inherited definition *within the sub-class*. The simple

rule is that if its defined locally, then the inherited definition is masked – much like a local variable in a theta expression can mask a variable from an outer scope.

However, it is still possible to access any public element from any inherited class – via the super mechanism (see Section 7.4.3 on page 133).

There is a subtle – though important – difference between the way that Go! treats inheritance and that found in other object oriented languages. Within a class body, unless explicitly marked with the `this` keyword (see Section 7.4.2 on page 132) any references to programs from within a class body refer to other programs *either in the same class body or inherited definitions*. In particular, there is no automatic 'down-shifting' to definitions found in sub-classes.

This is important because if you wish to use the 'current' version of a program then you will need to use the `this` keyword to do so. It is also important for security of programs: it becomes impossible to pervert the programmers intentions in a program simply by sub-classing and overriding a definition.

Inheritance and types

If a class includes any class rules, then the type of the class must be a sub-type of the inherited types. Often the inheritance in a class hierarchy reflects a similar inheritance in the type hierarchy.

Note that it is not permitted for a state-free class to inherit from a state-full class.

The main type inference rule for class rules expresses the main constraints on safe inheritance for state-free classes:

$$\frac{E \vdash_t \texttt{C} \rightsquigarrow \texttt{[T}_{C_1}\texttt{,...,T}_{C_n}\texttt{]@=}T_C \quad E \vdash_t \texttt{S} \rightsquigarrow \texttt{[T}_{S_1}\texttt{,...,T}_{S_m}\texttt{]@=}T_S \quad T_C \preceq T_S}{E \vdash_{ok} C(C_1, \ldots, C_n)\texttt{<=}S(S_1, \ldots, S_m)} \tag{7.1}$$

where

$$E \vdash_t C_i \rightsquigarrow T_{C_i}$$

and

$$E \vdash_t S_i \rightsquigarrow T_{S_i}$$

For example, we might have a type defining the interface for `animal`s:

```
animal <~ { mode:[symbol]{}, eats:[]=>string }.
```

and a sub-type of `animal` – `bird` – which refines it with aspects of birdness:

```
birdness <~ {no_of_legs:[integer]{}}.
bird <~ animal.
```

When it comes to defining classes that implement the `animal` and `bird` interfaces we may see a similar hierarchy. For example, Program 7.3.1 defines a prototypical animal and Program 7.0.1 on page 120 defines a prototypical bird in terms of the prototypical animal.

Program 7.3.1 An `animal` class

```
animal:[]@=animal.      --                                  la-
bels and types can have same name
animal..{
  mode('walk').
  mode('run').

  eats()=>"grass"
}
```

Note that Go!'s class system allows for alternate implementations of the type. Thus one might have another kind of bird, an `ostrich` for example, that derives its implementation completely independently from `animal` – as in Program 7.3.2 on the following page.

Program 7.3.2 An `ostrich` class

```
ostrich:[]@=birdness.
ostrich..{
  no_of_legs(2).
  mode('walk').
  mode('run').

  eats()=>"sand".          -- Why else do they bury their head in it?
}.
```

7.3.1 Multiple inheritance

Go!'s object notation permits *multiple inheritance* – with some simple restrictions. If a given element can be inherited from more than one super class, only one of the super elements will be used: they will *not* be unioned.[1] Which of the available definitions used is *not* defined; and so they had better refer to the same actual definition. However, see below for techniques for accessing particular elements of a super class.

7.4 Accessing and using classes

The fundamental operator used in accessing the definitions of a labeled theory is the dot operator:

$Exp.M(X_1,\ldots,X_n)$

which means

> invoke $M(X_1,\ldots,X_n)$ from the definitions in the theory identified by the label *Exp*.

The query $M(X_1,\ldots,X_n)$ may be a goal (see section 4.2.3 on page 83), action (see section 5.1.5 on page 97), grammar call (see section 6.1.3 on page 111) or function call (see section 3.5.6 on page 70), depending on the context and type integrity.

[1] This avoids one of the classic problems of multiple inheritance where a method can be inherited more than once.

What the query *cannot* be is a reference to an object variable (a.k.a. instance variable). Go! does not support accessing object variables and constants from outside the class body in which they are defined.

The main issue to remember here is that only those interface elements that are associated with the type of *term* may be accessed using the dot operator. The type gives the interface, and the interface determines the legal accesses.

It is possible to give a formal set of inference rules for proving theorems in the context of multiple labeled theories: essentially reflecting the two choices of reducing a condition of the form: $L.P$ by reducing the P part – with a rule from the theory identified by L – and reducing the L part – with a class rule to replace L with another label term M (say).

7.4.1 Creating objects

An instance of a class corresponds simply to an occurrence of its constructor term. For state-free classes, these constructor terms are directly analogous to `Prolog` terms: two occurrences of expressions for state-free class labels that are unifyable refer to the same class. For example,

```
bird=bird
```

is true because `bird` is a class label for a state-free class.

However, given a state-full class, such as the `stack` class:

```
stack:[list[t]]@>stack.
stack(I)..{
  S:list[t] := I.

  push(E) -> S:=[E,..S].

  ...
}
```

two occurrences of a `stack` terms are *not* equal, even if they are unifyable:

```
\+ stack([2])=stack([2])
```

This is because a state-full constructor's value is not the expression itself but a new object – each evaluation of the constructor will yield a different object.

This reflects, of course, the intended semantics of a state-full class where the object may evolve over the course of a computation.

7.4.2 `this` object

Under normal circumstances, within a class body references to names either refer to elements defined within the same class body or to elements that are defined in a super class. Occasionally, it is necessary to be more explicit about the appropriate source of an element.

The `this` keyword – which only permitted in a definition in a class body – refers to the object as created. The object might not have been created directly as an object of the 'current' class – the class may have been sub-classed and an object of the sub-class created.

For example, in the `animal` class, we might have a rule for mode of travel involving running:

```
animal:[]@=animal.
animal..{
  mode('run') :-
    this.no_of_legs(2).
  ...
  no_of_legs(4).       -- by default, animals have 4 legs
}
```

The `mode` clause references the `no_of_legs` predicate relative to the `this` keyword. This will always refer to the `no_of_legs` definition as it is defined in the object actually created. If we reference an `animal` object directly, then `this` refers to an object of type `animal`. If we sub-class `animal`, and reference an instance of that sub-class, then `this` will refer to the sub-classed object.

So, for example in the definition of `bird` in Program 7.0.1 on page 120, it is declared that a `bird` has 2 legs. If we evaluate `mode` relative to a `bird` object, then `mode('run')` will be satisfied; because even though `animal` defines `no_of_legs` to be four,

```
this.no_of_legs(2)
```

is true due to the definition in `bird`.

Normally, even when sub-classed, methods and other elements in a class body do not access the 'leaf' methods of the class associated with the object. The `this` keyword is useful for those occasions where a definition in a class body requires access to overridden methods rather than locally defined methods.

7.4.3 Super and inherited definitions

For the most part – where a method is not defined in a class and it is defined in a super class – super class methods are automatically 'in scope' in a class body.

Since Go! permits multiple inheritance there may be more than one super class that defines a given method. Furthermore, it is possible to get a lattice-like structure where a single definition may be inherited multiple times from a single ancestor class.

To avoid problems associated with such multiple definitions, only *one* definition of a class's super classes is used. Which one used is left undefined in the definition of Go!. Thus, even if multiple definitions of a method might be available through different inheritance routes, only one 'copy' of the definition will ever be used. As a result it *should* be the case that if a given definition is multiply defined then it shouldn't matter which definition is used.

It is possible, however, to explicitly *program* using inherited definitions from more than one super class. To directly access definitions associated with super classes – even if the methods have been overridden – Go! introduces 'into scope', within the class body, identifiers that denote each of the super classes of that class. The identifiers used are the class names of the super classes.

For example, in the `bird` class, we might wish to redefine `mode` using `animal`'s `mode` with a modification. We can do this explicitly by using `animal.mode`:

```
bird<=animal.
bird..{
  mode('fly').
  mode('run') :- animal.mode('walk').
  ...
}
```

The second rule for `mode` bypasses the local definition of `mode` and uses the definition from `animal`.

Using explicit super calls such as in `mode` above can be used to deliver a kind of *inheritance union*. The normal interpretation of inheritance does not allow a sub-class to *extend* an inherited definition – only to replace it. However, we can use explicit *super* references to extend an inherited definition and also to access all available definitions from super classes:

```
pred(x) :- super_1.pred(x).
...
pred(x) :- super_n.pred(x).
```

where $\texttt{super}_i$ are the super-classes of the class in which `pred` is defined. Of course, this kind of definition is not especially elegant.

7.5 Inner Classes

An *inner* class is on that is defined within a class body. For example, in Program 7.5.1 on the facing page we have an inner `parasite` class that is defined in the `bird` class. Inner classes represent a particular form of aggregation: the inner theory is defined inside and is part of the outer theory.

An inner class may be *exported* by a class if the class type signature is part of the class's type signature. For example, Program 7.5.2 on page 136 is very similar to Program 7.5.1, except that the inner class type is now part of `bird`'s type.

Program 7.5.1 An inner parasite

```
bird:[]@=birdness.
bird..{
  no_of_legs(2).
  mode('fly').

  para<~{ eat:[]=>string. }.
  parasite:[string]@=para.
  parasite(Where)..{
    eat()::mode('fly')=>"wings".
     eat()=>Where.
  }.
}.
```

Inner classes are not needed that often; but when they are, there is no alternative! The key is that variables and programs that are defined in an enclosed class are in scope in the inner class.

Once exported, the inner constructor can be used in the same way that other programs are referenced from a class:

```
Tweety = bird;
TweetyParasite = Tweety.parasite("stomach")
```

The type of `TweetyParasite` is `para` – this type had to be declared in the same level as `bird` because the `birdness` type references it.

Where constructors for a top-level class are directly analogous to normal `Prolog` terms, the same is not precisely true for constructors for inner classes. An inner constructor is a term but it has hidden extra arguments that are added as part of the compilation process.

7.5.1 Anonymous classes

An anonymous class is a particular kind of class which is defined and used at once. Anonymous classes are *expressions* that define

Program 7.5.2 An exported inner parasite

```
birdness <~ { no_of_legs:[number]{}. mode:[symbol]{}.
    parasite:[string]@=para. }.
para<~{ eat:[]=>string. }.

bird:[]@=birdness.
bird..{
  no_of_legs(2).
  mode('fly').

  parasite(Where)..{
    eat()::mode('fly')=>"wings".
     eat()=>Where.
  }.
}.
```

both the class and the single instance of that class. There is no constructor defined for this class – its occurrence also defines the only instance of the class.

There are two variants of the anonymous class, either a template type is specified, or a label term is given which the anonymous class is sub-classing:

- If the anonymous class takes the form:

  `(`*label*`..{` *definitions* `})`

  then this defines a new object whose type is the type of *label* with *definitions* being used to override inherited definitions from the *label* class. This is equivalent to the expression:

  NewLbl`(`F_1`,...,`F_n`)`

  together with a new class definition:

  NewLbl`:[`T_{F_1}`,...,`T_{F_n}`]@>`$Type_{label}$`.`
  NewLbl`(`F_1`,...,`F_n`) <=` *label*`.`

```
NewLbl(F1,...,Fn)..{
  definitions
}
```

where *NewLbl* is a new constructor symbol not occurring elsewhere in the program, F_i are the free variables occurring in the *definitions* that are defined in an outer context, and T_{F_i} are the corresponding types of F_i.

- If the anonymous class expression is of the form

```
(:type..{ definitions })
```

then this defines an object of type *type* which does not inherit from any existing class. It is the responsibility of the programmer to ensure that the enclosed *definitions* correctly implements the interface associated with *type*. This form of anonymous class is equivalent to the expression:

```
NewLbl:[TF1,...,TFn]@>type.
NewLbl(F1,...,Fn)
```

together with the new class definition:

```
NewLbl(F1,...,Fn)..{
  definitions
}
```

where *NewLbl* is a new constructor symbol not occurring elsewhere in the program.

Anonymous classes are useful for providing implementations of callbacks as well as acting as a more general form of lambda closure. For example, the `sort` function in Program 7.5.3 on the following page takes as argument a list and a theory label that implements the `compare[]` interface, as defined in:

```
compare[T] <~ { less:[T,T]{} }
```

Program 7.5.3 A `sort` function

```
sort:[list[t],compare[t]]=>list[t].
sort([],_) => [].
sort([E],_) => [E].
sort([E,..L],C)::split(L,C,E,S1,S2) =>
    sort(S1,C)<>[E]<>sort(S2,C).

split:[list[t],compare[t]+,list[t]-,list[t]-]{}.
split([],_,[],[]).
split([D,..L],C,E,[D,..S1],S2) :-
  C.less(D,E),
  split(L,C,E,S1,S2).
split([D,..L],C,E,S1,[D,..S2]) :-
  \+ C.less(D,E),
  split(L,C,E,S1,S2).
```

We can use an anonymous class in a call to `sort` that constructs a specific predicate for comparing `integer`s:

```
sort([1,2,0,10,-45],(:compare[integer]..{
  less(X,Y) :- X<Y.
}))
```

Free variables The definitions within an anonymous class may *share variables* with other expressions that are in scope. Such variables are *free* variables of the anonymous class. However, there is an important caveat, the value recorded within the anonymous class is a copy of the values of those free variables – unifying against a free variable within the anonymous class cannot affect outer instances of the variable; conversely if the outer instances are unified against that will not affect the inner occurrences.

However, where the free variables represent objects or read/write variables then the free variables within the anonymous do directly reflect the value of the original variables (such variables cannot be meaningfully be unified against).

Packages 8

Go! programs revolve around three principal constructs: rules, classes and packages – in increasing order of granularity. In this chapter we focus on the larger scale aspects of Go! programs – namely *package*s. Each Go! source file makes up a package.

Packages represent Go!'s equivalent of modules; Go! has a simple but effective package system that allows programs, classes and types to be defined in one file and re-used in others.

Packages may contain type definitions, classes, rules, variables and constants. A top-level program also takes the form of a package – with the addition of a standard action rule defined for the `main` symbol.

Packages may `import` other packages, in which case they are loaded automatically whenever the referring package is loaded. The Go! engine guarantees that a given package will only ever be loaded once; although it does not necessarily guarantee the *order* of loading, the system tries to ensure that packages are loaded in a way that dependent packages are loaded after the packages they depend on.

The form of a package source file is:

```
packagename{
  ...
  Definitions
  ...
}
```

The *packagename* is either a single identifier, or a sequence of identifiers separated by periods.

Note that the *packagename* must reflect the name of the file containing it. For example, if the package name is

```
foo.bar
```

then the *name* of the file containing this package should be of the form:

```
.../foo/bar.go
```

i.e., the package source file must be located in a particular directory structure – to the extent that the package name requires it.

The *reason* for this is that the Go! engine has to be able to locate the file containing the compiled package when it is loaded.

8.1 Package contents

A package may contain class definitions, rule definitions, type definitions, package variable definitions, package constant definitions and initialization actions. It may also include directives to `import` other packages.

The order of definitions within a package is not important – the Go! compiler is able to handle mutually recursive programs without requiring forward declarations.

Many of the elements that may be found in a package are discussed elsewhere in this manual. In the following sections we focus on those elements that are not covered elsewhere.

8.1.1 Package constants

A package constant is declared at the top-level of a package, using a `=` statement:

```
packageName{
  ...
  V:type.
  V = initial.
  ...
}
```

The identifier `V` is constant in the sense that, once evaluated, it is not modifiable. It is evaluated as the enclosing package is loaded, in an order that is not guaranteed – although the compiler attempts to ensure that any dependent values are evaluated before the variable itself is evaluated.

The type declaration for the variable is required; however, it can be folded into the statement thus:

```
packageName{
  ...
  V:type = initial.
  ...
}
```

Unlike package variables – see Section 8.1.2 on the next page – package constants *may* be exported from a package. In fact, by default, all allowable definitions in a package are exported. Thus constants declared in a package are made available to any packages that `import` the package.

The type inference rule for a constant definition in a package is:

$$\frac{E_G \vdash_t Ex \rightsquigarrow T_{Ex}}{E_G \vdash_t extends_\theta(\texttt{V = Ex}, \mathrm{E}_G) = [(\mathrm{V}, \mathrm{T}_{Ex}), .. \mathrm{E}_G]} \tag{8.1}$$

where E_G is the environment derived from the package environment up to and including the definition of V itself.

Package constants are evaluated in the full context of the package; i.e., the expressions that define their value can involve functions defined in the package and can even involve –directly or indirectly – other package constants and variables.

However, if there is a circular dependency between package constants and variables; if the expression denoting the value of a constant refers to another constant, *and* that constant's value expression also refers to this constant, then this can lead to serious problems when loading the package – it is

possible for the Go! system to enter into a loop *during the loading* of the module.

To try to prevent this, the compiler prints a warning message if it encounters what appears to be a circular dependency in package constants and variables.

This does not apply to mutually recursive programs however; as they are not evaluated as part of the package loading process. Therefore, it is quite safe to have mutually recursive programs. Furthermore, those mutually recursive programs may reference package variables and constants without harm.

8.1.2 Package variables

A package variable is a re-assignable variable declared at the top-level of a package, using a `:=` statement:

```
packageName{
  ...
  V:type := initial.
  ...
}
```

Package variables have two major restrictions – compared to regular logical variables – their values must be ground at all times. In addition, variables are not exportable from packages. They also have a major freedom compared to logical variables – they can be re-assigned.

If you need to export a package variable, then build accessor and mutator programs to manipulate the variable. For example, in

```
mutator{
  V : list[integer] := [].

  getV:[]=>list[integer].
  getV() => V.
```

```
  add2V:[integer]*.
  add2V(N) -> V := [N,..V].
}
```

the variable `V` is not directly exported. The programs `getV` and `add2V` are exported; and can then be used by programs in other packages to modify `V`.

The merit of this approach is that the implementer of the `mutator` package is able to strictly control access to the variable – making it simpler to be certain of the integrity of the variable's state.

8.1.3 Package initialization

In addition to package variables and constants having an initial expression associated with them, it is possible to define an action that will be executed on loading the package. Such initialization actions use the notation:

```
packageName{
  ...
  $ {
    Action
  }
  ...
}
```

The initialization action is executed *after* any initializers associated with package constants and variables. Furthermore, if a package `imports` one or more other packages, then the initializers of those packages will also be run before the importing package's initializer – thus ensuring that the initializer executes in a well defined environment.

A package can have any number of initializers in its body, however the relative order of execution between these different initializers is not defined.

Package initializers can be useful for certain classes of *active* packages – such as file system packages that may need to open certain standard files.

8.1.4 Package exports

By default, *all* the exportable elements defined in a package are exported; except for re-assignable package variables. However, a definition may be prefixed by the `private` keyword, in which case the definition will not be exported.

The `private` keyword should appear before the first defining statement of the package element. For example, to define a `private` function `app`, the `private` keyword should be used as part of `app`'s type declaration:

```
private app:[list[t],list[t]]=>list[t].
app([],X)=>X.
app([E,..X],Y)=>[E,..app(X,Y)].
```

This is a recommendation rather than a language restriction. The Go! compiler will also pick up a `private` declaration if it precedes one of the rules that defines the program.

The `private` keyword may be attached to program elements, constant declarations, class definitions or even type definitions.

8.2 Importing packages

The `import` directive in a package body is used to indicate that a particular package is required for that package. The form of the `import` statement is:

`import` *`packagename`*`.`

where *packagename* is a dotted sequence of identifiers that matches the package name used in the package file. Note that *packagename* must match exactly the package name used in the package's defining source file.

The effect of an `import` directive is to make available to the importing package all the definitions of the imported package. This includes classes, rules of various kinds, any types defined within the imported package and any *constants* defined within the package.

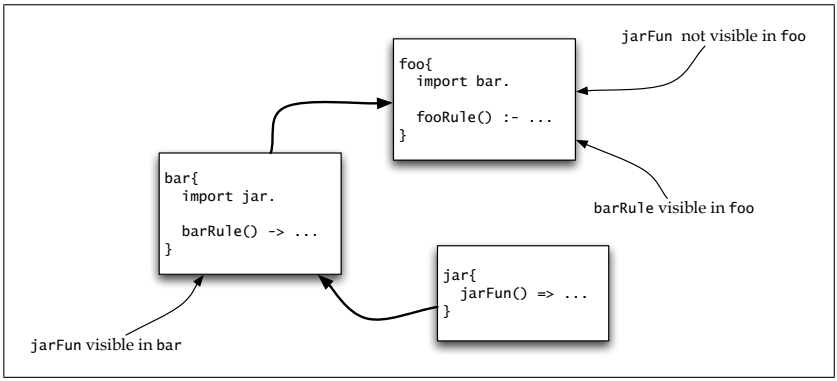


Figure 8.1: A three-way package `import`

The Go! engine ensures that any given package will only be loaded once, however many requests for its `import` are found. Furthermore, any initialization code associated with a package (see 8.1.3 on page 143) will also only be executed once.

Circular chains of `imports` The Go! compiler requires that a package be compiled before it can be imported; more specifically the compiler searches for the compiled package when compiling a package that `import`s a package. Thus, it may be important to ensure that dependent packages are compiled after the packages that they depend on. It is not permitted to have a circular chain of package `import`s – with one package importing another, which in turn causes the original to be imported.

It is possible for a package to `import` a package that `import`s other packages. These latter packages will be automatically loaded as needed. However, the definitions in these dependent packages are *not* automatically made available to the original `import`er. For example, figure 8.1 illustrates a case with three packages: `foo`, `bar` and `jar`. In this scenario `jarFun` is available within the `bar` package, but not in the `foo` package – even though loading the `foo` package will cause `jar` to be loaded. If `jarFun` is required directly within the `foo` then it will have to be explicitly `import`ed by the `foo` package. Of course, the `barRule` action procedure is

available within the `foo` package.

This can become an issue for type and other definitions that are shared over many packages. In that situation, the shared definitions will need to be `import`ed in each context that they are required.

8.3 Top-level main programs

Any package can also be treated as the top-level program – provided that the package has a definition for the single argument action procedure `main`. In fact, `main` is a reserved word in Go!: if a `main` program is defined in a package then it *must* be consistent with the type assertion:

```
main:[list[string]]*
```

If a package is executed at the top-level, then the `main` program in that package is executed and given as its single argument a list of the command-line arguments specified in the execution. For example, if a package `foo` were mentioned as the top-level package to execute in:

```
% go foo a b c
```

then the package `foo` must have an appropriate definition for `main` and that action procedure is entered – with argument the list

```
["a","b","c"]
```

Since the command line arguments are passed in as `string`s it is common for these argument strings to be parsed before they can be used in the application proper.

The `%%` parse expression and the `go.stdparse` package become handy in this situation. For example, to pass a integer value to a Go! fragment, where the number comes from the command line itself, then the classic way to do this is:

```
mainPackage{
  import go.stdparse.
```

```
    ...
    main([Arg,..More]) ->
      appProg(integerOf%%Arg);...
}
```

The `integerOf` grammar program parses a string into a `integer` value (see Section 12.2.3 on page 198).

See Chapter 20 on page 277 for further information on compiling and running Go! programs.

8.4 Standard Packages

Much of the functionality of the Go! system is encapsulated in special packages that are not automatically included in every program. By convention, all Go! system packages have package names of the form: `go.`*`name`*; for example, the system input/output package is called `go.io`. To access the standard I/O package, then, it is necessary to load the `go.io` package:

```
yourpackage{
  import go.io.

  ...
}
```

The reason that `go.io` is not automatically included in every package is that that permits non-standard I/O systems to be used - for example in embedded applications, or in systems which have to interact with file systems in special ways.

The standard set of packages will vary from time to time, the current set includes the packages

`go.cell` Implements a re-assignable resource entity.

`go.datelib` Implements a collection of date related functions.

`go.dynamic` Implements dynamic relations; relations that can be updated.

`go.hash` Implements a hash-table package.

`go.io` Implements the standard I/O package

`go.mbox` Implements an internal thread communication package.

`go.setlib` Implements a collection of set-like functions.

`go.sort` Implements a sort function

`go.stack` Implements a shareable updatable stack package.

`go.queue` Implements a shareable updatable queue package.

`go.stdlib` The standard Go! language support package.[1]

`go.stdparse` Implements a range of parsing functions, allowing the conversion of strings to numbers, for example.

`go.unit` Implements a unit-testing framework.

`go.xml` Implements an XML parser and displayer package. Also defines the Go! version of the DOM (Document Object Model).

`go.http` Implements many of the functions needed to build a Web server or an HTTP client.

[1]This package *is automatically loaded* as it is required for successful execution of any Go! program.

Part II

Standard library

The Standard library consists of the functions, predicates and actions that are part of the standard definition of Go!. Note, however, that in order to access certain of these built-in functions it may be necessary to use explicit `import` directives.

Arithmetic Primitives 9

Go! has two fundamental numeric types – `integer` and `float`. These types are not in a sub-type relationship to each other; however, they are both sub types of the `number` type. I.e., the type lattice is defined by:

```
integer<~number.
float<~number.
```

Most arithmetic functions are polymorphic where the arguments are a sub-type of `number`. For example, the `+` function is polymorphic, where both arguments must be either `integer` or `float` and returns a corresponding type.

Because there is no sub-type relationship between `float` and `integer` – i.e., neither is a sub-type of the other – it may be necessary to explicitly convert a value from one type to another. This conversion is not done automatically. The standard `n2float` function (see Section 9.1.14 on page 158) can be used to convert from an `integer` to a `float` and the `itrunc` function (see Section 9.1.11 on page 158) converts – with potential loss of precision – from `float`ing point to `integer`.

Any exceptions raised by arithmetic primitives take the form:

`error(`*`FunctionName`*`,`*`code`*`)`

where *`code`* gives some indication of the kind of exception being raised. See Appendix 21 on page 283 for a complete list of standard error codes.

9.1 Basic arithmetic primitives

9.1.1 + – Numeric addition

```
+:[T<~number,T]=>T
```

The + function expects two numeric arguments and returns their sum.

The precise type associated with + bears further inspection: + is actually a polymorphic function – defined over any kind of number value. However, the type of each argument should be the *same* and the type of the result is also the same. Thus, + might be thought of as being several functions rolled into one: an integer-only addition and a floating point addition function.

+ is a standard infix operator recognized by the Go! compiler.

Error exceptions [1]

'eINSUFARG' At least one of the arguments is uninstantiated.

9.1.2 - – Numeric subtraction

```
-:[T<~number,T]=>T
```

The - function expects two numeric arguments and returns the result of subtracting Y from X.

Note that unary arithmetic negation is equivalent to subtracting the negated expression from 0 (or 0.0 depending on the type of the negated expression).

Like +, - is actually a polymorphic function, separately handling integer and floating point arithmetic.

- is a standard operator recognized by the Go! compiler.

Error exceptions

'eINSUFARG' At least one of the arguments is uninstantiated.

[1] See Appendix 21 on page 283 for the definition of the standard error codes

9.1.3 * – Numeric product

```
*:[T<~number,T]=>T
```

The * function expects two numeric arguments and returns their product.

Like +, * is actually a polymorphic function, separately handling integer and floating point arithmetic.

* is a standard operator recognized by the Go! compiler.

Error exceptions

'eINSUFARG' At least one of the arguments is uninstantiated.

9.1.4 / – Numeric division

```
/:[T<~number,T]=>T
```

The / function expects two numeric arguments and returns the result of dividing *X* by *Y*.

Like other arithmetic functions, / is actually a polymorphic function, separately handling integer and floating point arithmetic.

/ is a standard operator recognized by the Go! compiler.

Error exceptions

'eINSUFARG' At least one of the arguments is uninstantiated.

'eIDIVZERO' *Y* is zero, or too close to zero

9.1.5 quot – Integer quotient

```
quot:[T<~number,T]=>integer
```

The quot function expects two numeric arguments and returns the integer quotient of dividing *X* by *Y*.

Like other arithmetic functions, quot is a polymorphic function, separately handling integer and floating point arithmetic. However, it always returns an integer result.

quot is a standard operator recognized by the Go! compiler.

Error exceptions

`'eINSUFARG'` At least one of the arguments is uninstantiated.

`'eIDIVZERO'` *Y* is zero, or too close to zero

9.1.6 `rem` – Remainder

```
rem:[T<~number,T]=>float
```

The `rem` function expects two numeric arguments and returns the remainder of the integer quotient of dividing *X* by *Y*. Note that while `quot` will always return an integral value, `rem` will always return a `float`.

`rem` is a standard operator recognized by the Go! compiler.

Error exceptions

`'eINSUFARG'` At least one of the arguments is uninstantiated.

`'eIDIVZERO'` *Y* is zero, or too close to zero

9.1.7 `abs` – Absolute value

```
abs:[T<~number]=>T
```

The `abs` function expects a numeric argument and returns the absolute value of *X*. The type of the returned value is the same as the argument: the absolute value of an `integer` is an `integer` and likewise for a `float`.

Error exceptions

`'eINSUFARG'` At least one of the arguments is uninstantiated.

9.1.8 `**` – Exponentiation

```
**:[T<~number,T]=>T
```

The `**` function expects two numeric arguments and returns the result of raising *X* to the power *Y*; i.e., X^Y.

`**` is a standard operator recognized by the Go! compiler.

Error exceptions

`'eINSUFARG'` At least one of the arguments is uninstantiated.

`'eINVAL'` Y is negative, and X is zero, or too close to zero

9.1.9 `integral` – Integer predicate

```
integral:[T<~number]{}
```

The `integral` predicate succeeds if its parameter is an integer; fails if it is a fractional value, or an integer that cannot be represented as a integer. This last point is important for large values; a number such as 1×10^{200} is integral from a mathematical point of view, but it cannot be represented as a 64 bit integer and so would *fail* the `integral` test.

This predicate will accept either `integer` or `float` values. However, it is clearly trivial for `integer` arguments.

Error exceptions

`'eINSUFARG'` The argument must be a number, variables are not permitted.

9.1.10 `trunc` – Extract integral part

```
trunc:[T<~number]=>T
```

The `trunc` returns the nearest integer value to its input. It works for all number values (including very large numbers) representable by Go!. Note that not all integral values are representable as `integer`s – especially very large values, with an absolute value larger than 2^{63}. Such large values must still be represented as `float`ing point numbers.

Error exceptions

`'eINSUFARG'` The argument must be a number, variables are not permitted.

9.1.11 itrunc – Extract integral part

```
itrunc:[T<~number]=>integer
```

The `trunc` returns the nearest integer value to its input. The value returned is an `integer`.

Error exceptions

`'eINSUFARG'` The argument must be a number, variables are not permitted.

9.1.12 floor – Largest integer that is smaller

```
floor:[T<~number]=>T
```

The `floor` returns the nearest integer value that is the same or smaller than its input.

Error exceptions

`'eINSUFARG'` The argument must be a number, variables are not permitted.

9.1.13 ceil – Smallest integer that is larger

```
ceil:[T<~number]=>T
```

The `ceil` returns the smallest integer value that is the same or larger than its input.

Error exceptions

`'eINSUFARG'` The argument must be a number, variables are not permitted.

9.1.14 n2float – Convert to float

```
n2float:[T<~number]=>float
```

The `n2float` 'converts' a number (either an `integer` or a `float`) into a `float`ing point equivalent.

Error exceptions

`'eINSUFARG'` The argument must be a number, variables are not permitted.

9.2 Modulo arithmetic

The modulo arithmetic primitives perform their arithmetic in a modulo range. This means that their arguments must be integer and the results are always constrained to be integers in the range $0..M-1$ where M is the modulus. If the modulus argument is 0 then the arithmetic is assumed to be at the precision of the machine (typically 64 bits)

9.2.1 `iplus` – modulo addition

```
iplus:[integer,integer,integer]=>integer
```

The `iplus` function expects three `integer` arguments and returns the sum of the first two modulo the third; i.e., `iplus(X,Y,M)` evaluates to $(X+Y)|_M$.

Error exceptions

`'eINSUFARG'` At least one of the arguments is uninstantiated.

`'eINTNEEDD'` At least one of the arguments is not an integer.

9.2.2 `iminus` – modulo subtraction

```
iminus:[integer,integer,integer]=>integer
```

The `iminus` function expects three `integer` arguments and returns the result of subtracting the second from the first, modulo the third; i.e., the value of `iminus(X,Y,M)` is $(X-Y)|_M$.

Error exceptions

`'eINSUFARG'` At least one of the arguments is uninstantiated.

`'eINTNEEDD'` At least one of the arguments is not an integer.

9.2.3 `itimes` – modulo multiplication

```
itimes:[integer,integer,integer]=>integer
```

The `itimes` function expects three `integer` arguments and returns the result of multiplying the first two arguments, modulo the third; i.e., the value of `itimes(X,Y,M)` is $(X * Y)|_M$.

Error exceptions

`'eINSUFARG'` At least one of the arguments is uninstantiated.

`'eINTNEEDD'` At least one of the arguments is not an integer.

9.2.4 `idiv` – modulo division

```
idiv:[integer,integer,integer]=>integer
```

The `idiv` function expects three `integer` arguments and returns the result of dividing the first two arguments, modulo the third; i.e., the value of `idiv(X,Y,M)` is $(X/Y)|_M$. The integer quotient of the division is returned.

Error exceptions

`'eINSUFARG'` At least one of the arguments is uninstantiated.

`'eINTNEEDD'` At least one of the arguments is not an integer.

9.2.5 `imod` – modulus

```
imod:[integer,integer]=>integer
```

The `imod` function expects two `integer` arguments and returns modulo result of the frist argument; i.e., `imod(X,M)` evaluates to $X|_M$.

Error exceptions

`'eINSUFARG'` At least one of the arguments is uninstantiated.

`'eINTNEEDD'` At least one of the arguments is not an integer.

9.3 Bit Oriented Arithmetic Primitives

All the bit oriented functions take `integer` arguments, and return `integer` results.

9.3.1 `band` – Bitwise and function

```
band:[integer,integer]=>integer
```

The `band` function expects two integer arguments and returns their binary bitwise intersection.

Error exceptions

`'eINSUFARG'` At least one of the arguments is uninstantiated.

`'eINTNEEDD'` At least one of the arguments is not an integer – i.e., is not representable as a 64 bit integer.

9.3.2 `bor` – Bitwise or function

```
bor:[integer,integer]=>integer
```

The `bor` function takes two integer arguments and returns their binary bitwise union.

Error exceptions

`'eINSUFARG'` At least one of the arguments is uninstantiated.

`'eINTNEEDD'` At least one of the arguments is not an integer – i.e., is not representable as a 64 bit integer.

9.3.3 `bnot` – Binary negation

```
bnot:[integer]=>integer
```

The `bnot` function expects an integer argument and returns its binary bitwise 1's complement.

Error exceptions

`'eINSUFARG'` The argument is uninstantiated.

`'eINTNEEDD'` The argument is not an integer – i.e., is not representable as a 64 bit integer.

9.3.4 `bxor` – Bitwise exclusive or function

```
bxor:[integer,integer]=>integer
```

The `band` function expects two integer arguments and returns their binary bitwise exclusive or.

Error exceptions

`'eINSUFARG'` At least one of the arguments is uninstantiated.

`'eINTNEEDD'` At least one of the arguments is not an integer – i.e., is not representable as a 64 bit integer.

9.3.5 `bleft` – Bitwise left shift function

```
bleft:[integer,integer]=>integer
```

The `bleft` function expects two integer arguments and returns result of leftshifting the first argument; i.e., `bleft(X,Y)` evaluates to $X * 2^Y$. The number is right-filled with zero.

Error exceptions

`'eINSUFARG'` At least one of the arguments is uninstantiated.

`'eINTNEEDD'` At least one of the arguments is not an integer – i.e., is not representable as a 64 bit integer.

9.3.6 `bright` – Bitwise right shift function

```
bright:[integer,integer]=>integer
```

The `bright` function expects two integer arguments and returns result of rightshifting the first argument by the second; i.e., `bright(X,Y)`

evaluates to $X/2^Y$. The result is sign-extended – if the original number is negative then the result is also negative.

Error exceptions

`'eINSUFARG'` At least one of the arguments is uninstantiated.

`'eINTNEEDD'` At least one of the arguments is not an integer – i.e., is not representable as a 64 bit integer.

9.4 Arithmetic inequalities

Basic inequality predicates such as `<`.

9.4.1 < – Less than predicate

```
<:[T<~number,T]{}
```

The `<` predicate expects two arguments and succeeds if the first is smaller than the second. Like many of the arithmetic functions, the arithmetic predicates are polymorphic, but require both arguments to be of the same type.

`<` is a standard operator, and `<` predicates are written in infix notation.

Error exceptions

`'eINVAL'` An attempt to compare incomparible values – such as variables.

9.4.2 =< – Less than or equal predicate

```
=<:[T<~number,T]{}
```

The `=<` predicate expects two arguments and succeeds if the first argument is smaller than or equal to the second.

`=<` is a standard operator, and `=<` predicates are written in infix notation.

Error exceptions

`'eINSUFARG'` At least one of the arguments is uninstantiated.

9.4.3 > – Greater than predicate

```
>:[T<~number,T]{}
```

The `>` predicate expects two arguments and succeeds if the first argument is greater than the second.

`>` is a standard operator, and `>` predicates are written in infix notation.

Error exceptions

`'eINSUFARG'` At least one of the arguments is uninstantiated.

9.4.4 >= – Greater than or equal predicate

```
>=:[T<~number,T]{}
```

The `>=` predicate expects two numeric arguments and succeeds if the first argument is greater, or equal to, the second.

`>=` is a standard operator, and `>=` predicates are written in infix notation.

Error exceptions

`'eINSUFARG'` At least one of the arguments is uninstantiated.

9.5 Trigonometric functions

The trigonometric functions generally accept either `integer` or `float` arguments. However, they will always *return* a `float` result.

9.5.1 `sin` – Sine function

```
sin:[number]=>float
```

The `sin` function returns the sin of its argument – interpreted in radians. The value of `sin` is only reliable if its argument in the range $[-2\pi, 2\pi]$.

Error exceptions

`'eINSUFARG'` The argument is uninstantiated.

9.5.2 `asin` – Arc Sine function

```
asin:[number]=>float
```

The `asin` function returns the arc sin of its argument. The returned value will be in the range $[-\pi/2, \pi/2]$.

Error exceptions

`'eINSUFARG'` The argument is uninstantiated.

`'eRANGE'` The parameter is out of range.

`'eINVAL'`

9.5.3 `cos` – Cosine function

```
cos:[number]=>float
```

The `cos` function returns the cosine of its argument – interpreted in radians. `cos(X)` is only reliable if `X` is in the range $[0, \pi]$.

Error exceptions

`'eINSUFARG'` The argument is uninstantiated.

9.5.4 acos – Arc Cosine function

```
acos:[number]=>float
```

The acos function returns the arc cosine of its argument in radians.

Error exceptions

'eINSUFARG' The argument is uninstantiated.

'eRANGE' The parameter is out of range.

'eINVAL'

9.5.5 tan – Tangent function

```
tan:[number]=>float
```

The tan function returns the tangent of its argument – interpreted in radians. atan requires its argument to be in the range $[-\pi/2, \pi/2]$.

Error exceptions

'eINSUFARG' The argument is uninstantiated.

9.5.6 atan – Arc Tangent function

```
atan:[number]=>float
```

The atan function returns the arc tangent of its argument.

Error exceptions

'eINSUFARG' The argument is uninstantiated.

'eRANGE' The parameter is out of range.

'eINVAL'

9.5.7 `pi` – return π

```
pi:()=>float
```

The `pi` function returns π, accurate to the resolution of the underlying IEEE floating point arithmetic.

9.6 Other math functions

9.6.1 `irand` – random integer generator

```
irand:[integer]=>integer
```

The `irand` function returns a random `integer` in the range $[0 \ldots X-1]$, where X is its argument. The value of X should be non-negative.

Error exceptions

`'eINSUFARG'` The argument is uninstantiated.

`'eINVAL'` A negative number is not permitted for `irand`

9.6.2 `rand` – random number generator

```
rand:[number]=>float
```

The `rand` function returns a random `float`ing point number in the range $[0 \ldots X)$, where X is its argument – which may be either `integer` or `float`. The value of X should be non-negative.

Error exceptions

`'eINSUFARG'` The argument is uninstantiated.

`'eINVAL'` A negative number is not permitted for `rand`

9.6.3 `srand` – seed random number generation

```
srand:[number]*
```

The `srand` action 'seeds' the random number generator with its `number` argumnent, which should be non-negative. `srand` can be used to either ensure a repeatable sequence (by seeding it with a fixed known value) or to ensure a more random, non-repeatable, sequence by seeding it with a value that is always different – such as the current time.

Error exceptions

`'eINSUFARG'` The argument is uninstantiated.

`'eINVAL'` A negative number is not permitted for `srand`

9.6.4 `sqrt` – square root function

```
sqrt:[number]=>float
```

The `sqrt` function returns the square root of its argument – which should be non-negative. The argument may be either an `integer` or a `float`; however, the result is always a `float`.

Error exceptions

`'eINSUFARG'` The argument is uninstantiated.

`'eINVAL'` A negative number is not permitted for `sqrt`

9.6.5 `exp` – exponentiation function

```
exp:[number]=>float
```

The `exp` function returns e^X, where X is its non-negative argument.

Error exceptions

`'eINSUFARG'` The argument is uninstantiated.

`'eRANGE'` The parameter is out of range; it causes an overflow to exponentiate it.

9.6.6 `log` – natural logarithm function

```
log:[number]=>float
```

The `log` function returns $\log_e(X)$. Its argument should be non-negative.

Error exceptions

`'eINSUFARG'` The argument is uninstantiated.

`'eRANGE'` The parameter is out of range.

9.6.7 `log10` – decimal logarithm function

```
log10:[number]=>float
```

The `log` function returns $\log_{10}(X)$. Its argument should be non-negative.

Error exceptions

`'eINSUFARG'` The argument is uninstantiated.

`'eRANGE'` The parameter is out of range.

9.7 Floating point manipulation

These functions give special manipulation of floating point numbers; they are commonly used in converting between floating point numbers and strings: either for parsing a string into a numeric value or in ddisplaying a number as a string.

9.7.1 `ldexp` – multiply by power of 2

```
ldexp:[float,float]=>float
```

The expression `ldexp(X,Y)` evaluates to $X \times 2^Y$. This is useful in converting between string representations of numbers and floating point numbers themselves.

Error exceptions

`'eINSUFARG'` The argument is uninstantiated.

9.7.2 `frexp` – split into fraction and mantissae

```
frexp:[float+,float-,integer-]{}
```

This predicate is satisfied if the fractional part of the first parameter – X – is equal to the second parameter – F – with the third parameter – E – equal to its exponent. More specifically, F should be a *normalized* number – i.e., its absolute value is in the range $[0.5, 1)$, or zero – and `frexp(X,F,E)` is satisfied if

$$X = F \times 2^E \wedge (F = 0 \vee |F| \epsilon [0.5, 1))$$

If X is zero, then both F and E should also be zero.

Although a predicate, the modes in this type definition indicate that X must be given and both F and E are output.

Error exceptions

`'eINSUFARG'` The argument X is uninstantiated.

9.7.3 `modf` – split into integer and fraction parts

```
modf:[flaot+,float-,float-]{}
```

A predication `modf(X,I,F)` is satisfied if the integer part of X is I and the fractional part is F. More specifically, $X = I + F$, and I is integral.

Although a predicate, the modes indicate that X must be given and both I and F are output.

This primitive is particularly useful when displaying floating point numbers.

Error exceptions

`'eINSUFARG'` The argument X is uninstantiated.

Standard Library 10

The standard libraries contain suites of programs that are useful in many different situations.

For many of the functions in this library, it is not necessary to explicitly `import` the corresponding packages. In fact, these programs are defined in the standard library `go.stdlib` – which is always automatically loaded for all programs.

However, some functions are in separate packages and will need to be explicitly `import`ed. All the packages in the standard library are part of the `go` package domain, and thus require an `import` statement of the form:

```
packageName{
  import go.library.
  ...
}
```

Where a function requires a specific package to be `import`ed, the description of the function explains which package to load.

10.1 List manipulation

Go! has a simple set of functions that can be used to manipulate lists. These are essentially culled from experience of what list functions are most used; of these list append (`<>`) is probably the most heavily used list function.

10.1.1 <> – List append

```
<>:[list[t],list[t]]=>list[t]
```

The <> polymorphic function appends its two arguments together: both arguments being lists of the same type. The definition of list append is quite short:

```
[] <> X => X.
[E,..X] <> Y => [E,..X<>Y].
```

As with many of the programs in the standard library, the *name* of the function is also an operator; allowing the use of infix notation in list append expressions.

10.1.2 append – List append predicate

```
append:[list[t],list[t],list[t]]{}
```

The <> list append function is convenient to use for the large majority of cases where it is used to concatenate two lists together. However, the traditional `Prologappend` predicate can be used for many additional things – such as splitting lists, searching lists and so on. It is defined as:

```
append([],X,X).
append([E,..X],Y,[E,..Z]):-append(X,Y,Z).
```

10.1.3 listlen – Length of a list

```
listlen:[list[t]]=>integer)
```

The `listlen` function counts the length of the list L. `listlen` is defined as though by:

```
listlen(L) => len(L,0).

private len:[list[t],integer]=>integer.
len([],C) => C.
len([_,..L],C) => len(L,C+1).
```

10.1.4 `in` – List membership

```
in:[T,list[T]]{}
```

The `in` standard predicate is true if the left hand argument term is on the list in the right hand argument.

The definition of `in` is:

```
X in [X,..._].
X in [_,..Y] :- X in Y.
```

Note that unification is used to determine element equality, and that an element may be on a list more than once: the `in` predicate will find subsequent entries on backtracking.

If you look for the definition of `in` in the `go.stdlib` package source you will not find it. This is because `in` has such a deep relationship to many of Go!'s features that the compiler generates specific versions of `in` whenever it is used.

10.1.5 `reverse` – List reverse

```
reverse:[list[t]]=>list[t]
```

The `reverse` polymorphic function reverses its argument. This is *not* 'naive reverse'; `rev` is a linear complexity function. The definition of `rev` uses an auxilliary function to accomplish list reversal:

```
reverse(L) => rv(L,[]).

private rv:[list[t],list[t]]=>list[t].
rv([],R) => R.
rv([E,..M],R) => rv(M,[E,..R]).
```

10.1.6 `nth` – Nth element of a list

```
nth:[list[T],integer]=>t
```

The `nth` function returns the n^{th} element of a list. The definition of `nth` is:

```
nth(L,_)::var(L)=>exception error("nth",'eINVAL').
nth([E,..._],1) => E.
nth([_,..L],N) => nth(L,N-1).
nth([],_) => raise error("nth",'eNOTFND').
```

10.1.7 front – the front portion of a list

```
front:[list[t],integer]=>list[t]
```

The front function returns the front portion of a list. It is defined as though by:

```
front([],_)=>[].
front([E,..L],C)::C>0 => [E,..front(L,C-1)].
front(_,_) => [].
```

10.1.8 tail – the tail portion of a list

```
tail:[list[t],integer]=>list[t]
```

The tail function returns the tail portion of a list, i.e., the last N elements of the list. It is defined as though by:

```
tail(L,C)::append(_,Tl,L), listlen(Tl,C) => Tl.
```

Note that its actual definition is significantly more efficient than this specification!

10.1.9 drop – drop n elements from a list

```
drop:[list[t],integer]=>list[t]
```

The drop function drops the first N elements from a list. It is defined as:

```
drop:[list[t],integer]=>list[t].

drop([],_)=>[].
drop(L,0) => L.
drop([_,..L],N) => drop(L,N-1).
```

10.1.10 iota – List construction

```
iota:[integer,integer]=>list[integer]
```

The iota function constructs a list of integers from X to Y inclusive. The definition of iota is:

```
iota(N,M)::N>M=>[].
iota(N,M)=>[N,..iota(N+1,M)].
```

10.2 Set manipulation

These programs support sets, represented as lists. Sets are not a fully first-class structure in Go!: there is no built in set data type. However, it is perhaps more useful to provide a suite of functions that implement set-like semantics over lists. In addition to these functions, Go! also has two specific operators for constructing set-like lists: the bag-of expression (see section 3.5.2 on page 67) and the bounded set expression (see section 3.5.3 on page 68).

These set functions are accessed via the setlib library:

```
import go.setlib.
```

10.2.1 \/ – Set union

```
\/:[list[t],list[t]]=>list[t]
```

The \/ (pronounced 'union') function unions two sets – represented as lists – together. Note that the \/ function assumes that both arguments are already sets – i.e., do not contain duplicate elements.

The definition of \/ is:

```
[]\/X=>X.
[E,..X]\/Y :: E in Y => X\/Y.
[E,..X]\/Y => [E,..X\/Y].
```

10.2.2 /\ – Set intersection

```
/\:[list[t],list[t]]=>list[t]
```

The /\ (pronounced 'intersection') function intersects two sets – represented as lists – together. Note that the /\ function assumes that both arguments are already sets – i.e., do not contain duplicate elements.

The definition of /\ is:

```
[]/\_=>[].
[E,..X]/\Y :: E in Y=> [E,..X/\Y].
[E,..X]/\Y => X/\Y.
```

10.2.3 \ – Set difference

```
\:[list[t],list[t]]=>list[t]
```

The \ (pronounced 'difference') function forms the set difference of two sets – represented as lists – together. Note that the \ function assumes that both arguments are already sets – i.e., do not contain duplicate elements.

The definition of \ is:

```
[]\_=>[].
[E,..X]\Y :: \+E in Y => [E,..X\Y].
[E,..X]\Y => X\Y.
```

10.2.4 subset – Subset predicate

```
subset:[list[t],list[t]]{}
```

The `subset` predicate is true if every element of the first list is also an element of the second.

For example:

```
subset([1,34],[1,2,34,10,34,21,-3])
```

is satisfied, whereas

```
subset([1,3,35],[1,2,34,10,34,21,-3])
```

is not, as `35` is not an element of the second argument.

The definition of `subset` is:

```
subset(X,Y) :-
  E in X *> E in Y
```

Character Primitives 11

This chapter provides a reference for the standard character primitives of the Go! language. The standard predicates define a number character classes as well as other attributes on characters.

The standard type for a character is `char`, and strings are of type `char[]`. Go!'s characters are based on the Unicode standard.

11.1 Basic character class primitives

11.1.1 `__isCcChar` – Other, control character

```
__isCcChar:[char+]{}
```

The `__isCcChar` predicate tests to see if its `char` argument represents a character in the Unicode 'Other, Control' general category. This includes the normal ASCII control characters.

Error exceptions

`'eINSUFARG'` The argument is uninstantiated.

11.1.2 `__isCfChar` – Other, format character

```
__isCfChar:[char+]{}
```

The `__isCfChar` predicate tests to see if its `char` argument represents a character in the Unicode 'Other, format' general category.

Error exceptions

`'eINSUFARG'` The argument is uninstantiated.

11.1.3 `__isCsChar` – Other, surrogate character

```
__isCsChar:[char+]{}
```

The `__isCsChar` predicate tests to see if its `char` argument represents a character in the Unicode 'Other, surrogate' general category. Surrogate characters are not complete characters in themselves.[1]

Error exceptions

`'eINSUFARG'` The argument is uninstantiated.

11.1.4 `__isCoChar` – Other, private character

```
__isCoChar:[char+]{}
```

The `__isCoChar` predicate tests to see if its `char` argument represents a character in the Unicode 'Other, private' general category. Private characters' interpretation is not determined by the Unicode standard.

Error exceptions

`'eINSUFARG'` The argument is uninstantiated.

11.1.5 `__isCnChar` – Other, unassigned character

```
__isCnChar:[char+]{}
```

The `__isCnChar` predicate tests to see if its `char` argument represents a character in the Unicode 'Other, unassigned' general category. Unassigned characters are reserved.

[1] Explicit use of surrogates is deprecated.

Error exceptions

`'eINSUFARG'` The argument is uninstantiated.

11.1.6 `__isLuChar` – Letter, uppercase character

```
__isLuChar:[char+]{}
```

The `__isLuChar` predicate tests to see if its `char` argument represents a character in the Unicode 'Letter, uppercase' general category. Uppercase letters may be used in identifiers.

Error exceptions

`'eINSUFARG'` The argument is uninstantiated.

11.1.7 `__isLlChar` – Letter, lowercase character

```
__isLlChar:[char+]{}
```

The `__isLlChar` predicate tests to see if its `char` argument represents a character in the Unicode 'Letter, lowercase' general category.

Error exceptions

`'eINSUFARG'` The argument is uninstantiated.

11.1.8 `__isLtChar` – Letter, titlecase character

```
__isLtChar:[char+]{}
```

The `__isLtChar` predicate tests to see if its `char` argument represents a character in the Unicode 'Letter, titlecase' general category. Titlecase letters include the German 'SS' character.

Error exceptions

`'eINSUFARG'` The argument is uninstantiated.

11.1.9 __isLmChar – Letter, modifier character

```
__isLmChar:[char+]{}
```

The __isLmChar predicate tests to see if its char argument represents a character in the Unicode 'Letter, modifier' general category.

Error exceptions

'eINSUFARG' The argument is uninstantiated.

11.1.10 __isLoChar – Letter, other character

```
__isLoChar:[char+]{}
```

The __isLoChar predicate tests to see if its char argument represents a character in the Unicode 'Letter, other' general category. This includes many of the CJK (Chinese Japanese Korean) ideographic characters.

Error exceptions

'eINSUFARG' The argument is uninstantiated.

11.1.11 __isMnChar – Mark nonspacing character

```
__isMnChar:[char+]{}
```

The __isMnChar predicate tests to see if its char argument represents a character in the Unicode 'Mark, nonspacing' general category.

Error exceptions

'eINSUFARG' The argument is uninstantiated.

11.1.12 __isMcChar – Mark, spacing combining character

```
__isMcChar:[char+]{}
```

The __isMcChar predicate tests to see if its char argument represents a character in the Unicode 'Mark, spacing combining' general category.

Error exceptions

'eINSUFARG' The argument is uninstantiated.

11.1.13 __isMeChar – Mark, enclosing character

```
__isMeChar:[char+]{}
```

The __isMeChar predicate tests to see if its char argument represents a character in the Unicode 'Mark, spacing combining' general category.

Error exceptions

'eINSUFARG' The argument is uninstantiated.

11.1.14 __isNdChar – Number, decimal digit character

```
__isNdChar:[char+]{}
```

The __isNdChar predicate tests to see if its char argument represents a character in the Unicode 'Number, decimal digit' general category.

Unicode allows for many different kinds of digit characters; from many different written languages. However, the Go! __isNdChar predicate is true only of those digit characters that the Unicode consortium denotes as denoting *decimal digits* (of which there are several hundred).

Note that even though a Go! language processor is required to correctly read all the potential digit characters as decimal digits, *generating* numeric values using other than the regular ASCII decimal digit characters is not required.

Error exceptions

`'eINSUFARG'` The argument is uninstantiated.

11.1.15 `__isNlChar` – Number, letter character

```
__isNlChar:[char+]{}
```

The `__isNlChar` predicate tests to see if its `char` argument represents a character in the Unicode 'Number, letter' general category. These characters are numeric, but are treated in the same way as letters.

Error exceptions

`'eINSUFARG'` The argument is uninstantiated.

11.1.16 `__isNoChar` – Number, other character

```
__isNoChar:[char+]{}
```

The `__isNoChar` predicate tests to see if its `char` argument represents a character in the Unicode 'Number, other' general category.

Error exceptions

`'eINSUFARG'` The argument is uninstantiated.

11.1.17 __isScChar – Symbol, currency character

```
__isScChar:[char+]{}
```

The __isScChar predicate tests to see if its char argument represents a character in the Unicode 'Symbol, currency' general category. This includes currency symbols that are not included in the native subset corresponding to the currency.

Error exceptions

'eINSUFARG' The argument is uninstantiated.

11.1.18 __isSkChar – Symbol, modifier character

```
__isSkChar:[char+]{}
```

The __isSkChar predicate tests to see if its char argument represents a character in the Unicode 'Symbol, modifier' general category.

Error exceptions

'eINSUFARG' The argument is uninstantiated.

11.1.19 __isSmChar – Symbol, math character

```
__isSmChar:[char+]{}
```

The __isSmChar predicate tests to see if its char argument represents a character in the Unicode 'Symbol, math' general category.

Error exceptions

'eINSUFARG' The argument is uninstantiated.

11.1.20 __isSoChar – Symbol, other character

```
__isSoChar:[char+]{}
```

The __isSoChar predicate tests to see if its char argument represents a character in the Unicode 'Symbol, other' general category.

Error exceptions

'eINSUFARG' The argument is uninstantiated.

11.1.21 __isPcChar – Punctuation, connector character

```
__isPcChar:[char+]{}
```

The __isPcChar predicate tests to see if its char argument represents a character in the Unicode 'Punctuation, connector' general category.

Error exceptions

'eINSUFARG' The argument is uninstantiated.

11.1.22 __isPdChar – Punctuation, dash character

```
__isPdChar:[char+]{}
```

The __isPdChar predicate tests to see if its char argument represents a character in the Unicode 'Punctuation, dash' general category.

Error exceptions

'eINSUFARG' The argument is uninstantiated.

11.1.23 __isPeChar – Punctuation, close character

```
__isPeChar:[char+]{}
```

The __isPeChar predicate tests to see if its char argument represents a character in the Unicode 'Punctuation, close' general category.

Error exceptions

'eINSUFARG' The argument is uninstantiated.

11.1.24 __isPfChar – Punctuation, final quote character

```
__isPfChar:[char+]{}
```

The __isPfChar predicate tests to see if its char argument represents a character in the Unicode 'Punctuation, final quote' general category.

Error exceptions

'eINSUFARG' The argument is uninstantiated.

11.1.25 __isPiChar – Punctuation, initial quote character

```
__isPiChar:[char+]{}
```

The __isPiChar predicate tests to see if its char argument represents a character in the Unicode 'Punctuation, initial quote' general category.

Error exceptions

'eINSUFARG' The argument is uninstantiated.

11.1.26 __isPoChar – Punctuation, other character

```
__isPoChar:[char+]{}
```

The __isPoChar predicate tests to see if its char argument represents a character in the Unicode 'Punctuation, other' general category.

Error exceptions

'eINSUFARG' The argument is uninstantiated.

11.1.27 __isPsChar – Punctuation, open character

```
__isPsChar:[char+]{}
```

The __isPsChar predicate tests to see if its char argument represents a character in the Unicode 'Punctuation, open' general category.

Error exceptions

'eINSUFARG' The argument is uninstantiated.

11.1.28 __isZlChar – Separator, line character

```
__isZlChar:[char+]{}
```

The __isZlChar predicate tests to see if its char argument represents a character in the Unicode 'Separator, line' general category; i.e., line separator characters.

Error exceptions

'eINSUFARG' The argument is uninstantiated.

11.1.29 __isZpChar – Separator, paragraph character

```
__isZpChar:[char+]{}
```

The __isZpChar predicate tests to see if its char argument represents a character in the Unicode 'Separator, paragraph' general category; i.e., paragraph separator characters.

Error exceptions

'eINSUFARG' The argument is uninstantiated.

11.1.30 __isZsChar – Separator, space character

```
__isZsChar:[char+]{}
```

The __isZsChar predicate tests to see if its char argument represents a character in the Unicode 'Separator, space' general category; i.e., space characters.

Error exceptions

'eINSUFARG' The argument is uninstantiated.

11.1.31 __isLetterChar – Letter character

```
__isLetterChar:[char+]{}
```

The __isLetterChar predicate tests to see if its char argument represents a Letter character. This represents the union of the Lu, Ll, Lt, Lm, Lo and Nl character categories.

Error exceptions

'eINSUFARG' The argument is uninstantiated.

11.1.32 __digitCode – Decimal value of a digit character

```
__digitCode:[char]=>integer
```

The __digitCode returns the decimal value associated with a particular digit character.

Error exceptions

'eINSUFARG' At least one of the arguments is uninstantiated.

11.1.33 __charOf – Unicode to character

```
__charOf:[integer]=>char
```

The __charOf returns the character corresponding to a Unicode value.

Error exceptions

'eINSUFARG' At least one of the arguments is uninstantiated.

'eINVAL' If the number does not represent a legal Unicode character.

11.1.34 __charCode – Character's Unicode value

```
__charCode:[char]=>number
```

The __charCode returns the Unicode code value of a character.

Error exceptions

'eINSUFARG' At least one of the arguments is uninstantiated.

'eINVAL' If the number does not represent a legal Unicode character.

11.1.35 `whiteSpace` – predicate for whitespace characters

```
whiteSpace:[char+]{}
```

The `whiteSpace` predicate is satisfied of a `char` if it is a standard white space character.

The `whiteSpace` predicate is part of the `go.stdparse` package.

String and symbol primitives 12

This chapter describes the various symbol and string processing primitives in the Go! standard library. Recall that Go! strings are lists of `char`; so many of the primitives described in Chapter 10 on page 173 and Chapter 11 on page 181 may also be relevant to processing strings.

12.1 Symbol and string processing

These functions manipulate symbols and strings.

12.1.1 `explode` – convert a symbol to a string

```
explode:[symbol]=>string
```

The `explode` function takes a `symbol` argument and returns a string – i.e., a list of `char` – consisting of the characters in the symbol's print name.

Error exceptions

`'eINSUFARG'` If the argument is an unbound variable.

12.1.2 `implode` – convert a string to a symbol

```
implode:[string]=>symbol
```

The `implode` function takes a string and returns a `symbol` whose print name is formed from the string argument.

Error exceptions

`'eINSUFARG'` If the argument is an unbound variable; or not a fully ground list of character.

12.1.3 `gensym` – generate a symbol

```
gensym:string=>symbol
```

The `gensym` function returns a *unique* `symbol` whose print name is formed from the string argument and a unique sequence of digits. The Go! engine attempts to ensure the uniqueness of the symbol by using a random number as the basis of the generated symbol.

The print name of the resulting `symbol` is patterned on:

`C`*NNN*

where `C` is the input prefix, and *NNN* is a random number.

Error exceptions

`'eINSUFARG'` If the argument is an unbound variable.

12.1.4 `int2str` – format an integer

```
int2str:[integer,integer,integer,char]=>string
```

The `int2str` function formats an integer into a string. Given a call of the form:

```
int2str(N,B,W,P)
```

The number to be formatted is *N*, the *base* of the representation is *B*, the number of characters to format the number into is *W*, the 'pad' character to use in the event that the number requires fewer characters is *P*.

If the width parameter *W* is zero, then the output of the string will be exactly the number required to format the number; otherwise exactly *abs(W)* characters will be returned. If *W* is less than

zero then the number will be left formatted, otherwise it will be right formatted.

If the base parameter B is less than zero then the number will be *signed*: i.e., either a `+` or a `-` character will be prefixed to the output string.

For example, the expression:

```
int2str(345,-16,5,‘ )
```

results in the string:

```
"  +159"
```

whereas,

```
int2str(345,10,0,‘ )
```

results in:

```
"345"
```

Error exceptions

`’eINTNEEDD’` If at least one of the arguments N, W, B is not an integer.

12.2 Parsing Strings

The `go.stdparse` standard library package includes a number of standard functions and grammars that are effective for interpreting strings.

12.2.1 `expand` – Simple tokenizer

```
expand:[list[t],list[t]]=>list[list[t]]
```

The `expand` function partitions a list into a list of sub-lists. Each element of the result is a fragment found between token ’markers’ (the second argument is the token marker).

For example, to split a string into words, with spaces between words, use `expand` :

```
expand("this is a list of words"," ")
```

which will return the result

```
["this", "is", "a", "list", "of", "words"]
```

12.2.2 `collapse` – List collapse

```
collapse:[list[list[t]],list[t]]=>list[t]
```

The `collapse` function is the converse of `expand` – it takes a list of lists of elements and strings them together into a single list. Between each element of the original list of 'words', it inserts a glue sub-sequence – which is the second argument.

For example, to construct a string from a list of words – putting a space between each word – use:

```
collapse(["this", "is", "a", "list", "of", "words"]," ")
```

The glue subsequence is insert *between* the elements:

```
"this is a list of words"
```

The `collapse` function can also be used as a kind of list flattener – converting a list of lists of things into simple list:

```
collapse([[1],[2,3],[4,5]],[])
```

to give:

```
[1,2,3,4,5]
```

12.2.3 `integerOf` – parse a string for an `integer`

```
integerOf:[integer-]-->string
```

The `integerOf` standard grammar will parse a string looking for an `integer` value.

If `integerOf` successfully parses a string as an `integer`, then the value represented in the string is unified with *N*.

Note that a classical way of using `integerOf` is in conjunction with the `%%` operator, as in:

```
X = integerOf%%"23"
```

which would result in `X` being unified with the number `23`.

Apart from the regular decimal notation, the `integerOf` grammar also recognizes Go!'s alternate integer notations – hexadecimal number (prefixed with a `0xfff`) – and character code (`0c`*`Char`*).

12.2.4 `naturalOf` – parse a string for a positive `integer`

```
naturalOf:[integer-]-->string
```

The `naturalOf` standard grammar will parse a string looking for a positive (i.e., unsigned) `integer` value.

If `naturalOf` successfully parses a string as an `integer`, then the value represented in the string is unified with *N*.

12.2.5 `hexNum` – parse a string for a hexadecimal `integer`

```
naturalOf:[integer-]-->string
```

The `hexNum` standard grammar will parse a string looking for a positive hexadecimal value.

If `naturalOf` successfully parses a string as an `integer`, then the value represented in the string is unified with *N*.

12.2.6 `floatOf` – parse a string for a `float`

```
floatOf:[float-]-->string
```

The `floatOf` standard grammar will parse a string looking for a `float` ing point value.

If `floatOf` successfully parses a string as a `float`, then the value represented in the string is unified with *N*.

The `floatOf` grammar accepts the same notation for floating point numbers as Go! itself; i.e., the normal floating point notation (see Section 1.2.5 on page 12).

12.2.7 `skipWhiteSpace` – skip white space in a string

```
skipWhiteSpace:[]-->string
```

The `skipWhiteSpace` standard grammar is true of a string if the input contains only white space characters. Use it to skip white space in text. The definition of white space is based on the Unicode standard – in particular it includes space characters, line characters, paragraph marks, and control characters.

12.2.8 `str2integer` – parse a string to get integer

```
str2integer:[string]=>integer
```

The `str2integer` function parses a string and decodes and returns an integer.

Error exceptions

`'eFAIL'` Not a numeric string

Dynamic knowledge bases 13

Go! supports several forms of dynamic storage: *dynamic* relations using the `dynamic` class, `hash` tables and individual shareable resources using the `cell` class.

The operators offered by the `dynamic` class to support dynamic knowledge bases permit the creation of 'dynamic' predicates; accessing their values and modification of the underlying knowledge base.

The `hash` class implements a more efficient form of access using hash tables – with the restriction that elements are identified by unique keys.

These facilities are similar to those found in `Prolog`; however there are some differences relating to the semantics and to the availability of these features:

- Dynamic knowledge bases must be specially declared in the sense that they are attached to particular variables, and those variables define the scope of the knowledge base.

- The second major difference is that the `dynamic` package only supports assertional facts – facts with no conditions. This reflects the overwhelming majority case for dynamic relations: they are populated with ground assertions and not general rules.

- While the `dynamic` and `cell` packages support tuples with variables embedded in them, the `hash` package does not support non-ground keys or values. (Nor do normal reassignable object and package variables.)

- Finally, modifications of dynamic knowledge bases are *actions* – which constrains the contexts in which they can be modified.

13.1 `cell` class – Shareable resource

The `cell` class is used to create a basic read/write resource or 'cell'. Using the `cell` class, it is possible to create resources that can be updated, shared and synchronized on.

The `cell` class is available in the `go.cell` package. To use it, you need to include the statement:

```
import go.cell.
```

in your program.

The methods available in a `cell` object are listed in Table 13.1. Since `cell` is a polymorphic class – polymorphic in the type of the element stored in the `cell` – we will refer to this type – $\mathtt{T}_V$ – when explaining the methods of a `cell` object. The `cell` is a

Method	Type	Description
`get`	`[]=>`$\mathtt{T}_V$	Access `cell` value
`set`	`[`$\mathtt{T}_V$`]*`	Reassign `cell` resource
`show`	`[]=>string`	Show contents as `string`

Table 13.1: Standard elements of a `cell[`$\mathtt{T}_V$`]` object

`sync`hronized entity – access to its contents are always serialized, making the `cell` itself thread-safe. However, if an action procedure requires access over several activities – a `get` followed by a `set` for example – then that transaction will require an overarching `sync` action.

13.1.1 Creating a new `cell` resource

```
cell:[T_V]@>cell[T_V]
```

A *cell* must be instantiated from the `cell` class using a declaration of the form:

Var `= cell(`*Exp*`)`

where *Exp* is the initial value of the `cell` variable.

For example, to create a new `cell` variable whose initial value is `0` we could use:

```
Counter = cell(0).
```

13.1.2 *cell*.get – The value of a `cell` resource

cell`.get:[]=>`T_V

where T_V is the type associated with the `cell` object when it is created.

The `cell` object has just two exported methods: `get` and `set`. We use `get` to access the value of the dynamic variable:

```
... CurrVal = Counter.get() ...
```

Any unbound variables embedded in the value of the read/write variable are *freshened* – i.e., replaced with new variables not occurring anywhere else. This makes it effectively impossible for read/write variables to share logical variables.

13.1.3 *cell*.set – Assign to a `cell` variable

cell`.set:[`T_V`]*`

where T_V is the type associated with the `cell` object when it is created.

The `set` action replaces a `cell` resource with a new value. It is written using the `set` attribute of the `cell`:

Var`.set(`*Ex*`)`

For example, to increment our counter we could use:

```
...;Counter.set(Counter.get()+1);...
```

Note that `set`ting a `cell` variable is an action – it can only be performed in a context where actions are expected. Furthermore, assignment to `cell`s is 'permanent' – i.e., it is not undone on backtracking.

13.1.4 *cell*.show – display contents of cell

```
cell.show:[]=>string
```

The show method function displays the contents of the cell – by invoking show on the bound value within the cell. The displayed string is prefixed by a $ character to highlight that a cell's value is being displayed.

13.2 Hash tables

The hash class provides a slightly more sophisticated form of shareable resource than the cell class. In particular it supports *hash table* lookup: i.e., primarily keyword-based search and updating of a table.

To access the hash table facilities it is necessary to incorporate the hash package:

```
import go.hash.
```

The methods available in a hash object are listed in Table 13.2. Since hash is a polymorphic type, in particular it polymorphic in the type of the key – T_K – and the type of the value – T_V.

However, for the actual keys associated with a hash table, and the values associated with those keys, must always be *ground*.

Method	Type	Description
count	[]=>integer	Count of elements in table
find	[T_K]=>T_V	Access hash value
insert	[T_K,T_V]*	Reassign hash entry
present	[T_K+,T_V]{}	Test for hash entry
delete	[T_K]*	Delete hash entry
ext	[]=>list[(T_K,T_V)]	Access hash table as list
keys	[]=>list[T_K]	Return list of keys in table

Table 13.2: Standard elements of a hash[T_K,T_V] object

13.2.1 `hash` – Create Hash table

`hash:[list[(`T_K`,`T_V`)],integer]@>hash[`T_K`,`T_V`]`

To create a new hash table we instantiate a new `hash` object giving it an *init*ial list of key/value pairs and its initial *size*:

```
... hash([('key1',Val1),...],10) ...
```

The `size` parameter is used as a guide to building the size of the hash table; the actual size of the table may vary in time. The types of the hash key and hash entry respectively and inferred from the initial value given in the `hash` table and/or from other uses of the table.

Multi-thread safe The `hash` class is a synchronized class, and each of the methods are synchronized. The result is that a hash table may be shared between threads without invalid results being returned. Of course, if a hash table *is* shared, then the internal synchronizations offered may not be enough to guarantee transactional integrity of applications – for example, when multiple operations on a `hash` table are required.

Error exceptions

`'eINVAL'` The type of the key associated with the hash table is not one of `symbol`, `number`, `char` or `string`.

13.2.2 *hash*`.find` – Access elements of table

hash`.find:[`T_K`]=>`T_V

The `find` function in the `hash` class is used to locate entries in the hash table. A function call of the form

H`.find(`*Key*`)`

returns the value associated with *Key*.

Error exceptions

`'eNOTFND'` There was no entry corresponding to the key in the table.

13.2.3 *hash*.present – Test presence of an element

hash.present:[T_K+,T_V]{}

The `present` predicate in the `hash` class is used to test for the presence of entries in the hash table. A goal of the form:

H.present(*Key*,*Val*)

succeeds if there is an entry in H that corresponds with *Key* that unifies with *Val*. Unlike the `find` method – which raises an exception – if the indicated Key is not present the predicate merely fails.

Note, though, that the mode of `present`'s type indicates that the key argument should be given. This reflects the restriction that `present` requires that the key is known.

13.2.4 *hash*.insert – Add element to table

hash.insert:[key:T_K,value:T_V]*

The `insert` method in the `hash` class is used to add new entries to the hash table. The action

H.insert(*Key*,*Value*)

inserts the *Value* term in association with *Key* – whether or not there already is an element corresponding to the `key` it is overwritten with the new `Value`.

Error exceptions

`'eINVAL'` The value or the key is not completely ground.

13.2.5 *hash*.delete – Remove element from table

hash.delete:[T_K]*

The delete action in the hash class is used to remove entries from the table. An action of the form:

H.delete(*Key*)

removes the entry corresponding to *Key* from the hash table *H*.

If there is no element corresponding to the key, this action has no effect.

Error exceptions

'eINVAL' The key used to access the element to delete is not ground.

13.2.6 *hash*.ext – Return all elements of table

hash.ext:[]=>list[(T_K,T_V)]

The ext function in the hash class is used to return all the entries in the hash table – as a list of 2-tuples, each consisting of the key and the value.

Note that the order of entries returned by ext does *not* necessarily reflect the order that they were inserted, nor does it reflect any kind of ordering relation between the entries. Hash tables do not, in general, preserve any kind of ordering between the elements.

13.2.7 *hash*.keys – Return all keys of table

hash.keys:[]=>list[T_K]

The keys function in the hash class is used to return all the distinct keys in the hash table – i.e., for each entry in the table there will be a corresponding entry in the list returned by keys. The main advantage of using keys instead of ext is that the values themselves are not extracted from the hash table.

Error exceptions

`'eINVAL'` There was an invalid entry in the table – should never happen!

13.2.8 *hash*.count – Count of elements in a hash table

```
hash.count:[]=>integer
```

The `count` function in the `hash` class is used to return the number of entries currently in the hash table.

13.3 Dynamic knowledge bases

The `dynamic` class provides facilities for implementing a simple form of *dynamic* relation. It provides a means for storing and manipulating 'atomic' facts – facts with no preconditions.

The `dynamic` class is accessed by `import`ing the `go.dynamic` package:

```
import go.dynamic.
```

The methods available in a `dynamic` object are listed in table 13.3 on the next page.

In addition to `dynamic` class itself, the `go.dynamic` package defines an interface `dynTest[]`. This interface is used by the client code to allow certain *callback*s when testing individual elements of the dynamic relation.

The `dynTest[T]` interface is defined as:

```
dynTest[T] <~ { check:[T]{} }.
```

This will be used as a test interface – for example with the `match` function only elements of the dynamic relation that satisfy the `check` predicate are returned.

Method	Type	Description
`mem`	`[T]{}`	Test if `T` is in the relation
`add`	`[T]*`	Add entry to relation
`del`	`[T]*`	Remove entry
`delc`	`[dynTest[T]]*`	Remove E s.t. `C.check(E)`
`delall`	`[T]*`	Remove all that match `T`
`delallc`	`[dynTest[T]]*`	Remove all `E::C.check(E)`
`ext`	`[]=>list[T]`	Return relation as a list
`match`	`[dynTest[T]]=>list[T]`	Return all `E::C.check(E)`

Table 13.3: Standard elements of a `dynamic[T]` object

13.3.1 `dynamic` – Creating a dynamic relation

```
dynamic:[list[T]]@>dynamic[T]
```

A `dynamic` relation is created by creating a new instance of the `dynamic` class where the argument of the constructor is a list of the initial tuples in the dynamic relation:

```
onTopOf = dynamic([('blockA','blockB')]).
```

This has the effect of declaring that `onTopOf` is a dynamic relation with an initial value approximately equivalent to the program:

```
onTopOf('blockA','blockB').
```

In general the dynamic relation can be 'seeded' with any number of initial facts, including no facts at all.

Note that all tuples in a dynamic relation must of the same type, and that uses of the dynamic relation must be type consistent with the elements of the relation. In the sections that follow, we will use `T` to refer to the type of entries in the `dynamic` relation.

13.3.2 *dynamic*.`mem` – Member of a dynamic relation

```
dynamic.mem:[T]{}
```

The `mem` predicate is satisfied for elements of the dynamic relation that unify with its argument:

```
...,onTopOf.mem((A,B)),...
```

If an entry in the dynamic relation has variables in it, then each time that entry is 'retrieved' via the `mem` method it is *refreshed*: i.e., any variables in the tuple are replaced with fresh variables.[1] This has the net effect of making dynamic relations very analogous to 'statically defined' relations – i.e., regular programs.

13.3.3 *dynamic*.add – Adding to a dynamic relation

```
dynamic.add:[T]*
```

We can add to a dynamic knowledge base by applying the `add` method of the `dynamic` object. It takes as an argument the element to be added – typically a tuple – `add` adds the new tuple to the end of the dynamic relation.

Note that the `add` method is an *action*; i.e., it is only legal to modify a dynamic program wherever actions are legal – typically within an action rule.

An action of the form:

```
...,onTopOf.add(('blockC','blockD')),...
```

will add the tuple

```
('blockC','blockD')
```

to the `onTopOf` dynamic program.

13.3.4 *dynamic*.ext – Dynamic relation as a list

```
dynamic.ext:[]=>list[T]
```

The `ext` method returns the extension of the dynamic relation as a list of entries:

```
...onTopOf.ext()...
```

will return, as a list, all the tuples in the `onTopOf` dynamic relation. Note that all variables in the dynamic relation are renamed to fresh variables in the returned list.

[1] The technical term here is 'standardizing apart'.

13.3.5 *dynamic*.del – Remove element

```
dynamic.del:[T]*
```

There are several methods for removing elements from a dynamic relation. The simplest is the `del` method. The `del` method takes the form:

`...,`*D*`.del(`*Term*`),...`

where *D* is the dynamic relation, removes the first element that unifies with *Term*.

It is legal to remove a non-existent entry – i.e., *Term* may not unify with any of the entries in the dynamic relation.

13.3.6 *dynamic*.delc – Remove element

```
dynamic.delc:[dynTest[T]]*
```

The `delc` method removes a tuple from a dynamic relation that satisfies a query. The query has to be encoded as a class label or object whose type is `dynTest[T]`.

The `delc` method takes the form:

`...,`*D*`.delc(Tst),...`

where *D* is the `dynamic` object and *Tst* is a `dynTest[T]` class. The `delc` method deletes the first entry in *D* for which `Tst.check(`*E*`)` succeed.

For example, given the `onTopOf` dynamic relation, a tuple representing a block being on top of itself – physically impossible but not logically. We can delete such an entry by defining the labeled theory in program 13.3.1 on the following page and using the action:

```
...;D.delc(selfTop(D));...
```

Note that there is no guarantee about the order of elements in a relation; therefore the `del` and `delc` methods should only be used when the programmer is certain that there is only one element that will match the test, or for which it doesn't matter.

Program 13.3.1 A theory about being on top of yourself

```
selfTop:[dynamic[T]]$=dynTest[T].
selfTop(D)..{
  check(E) :- D.mem((E,E))
}
```

13.3.7 *dynamic*.delall – Remove matching elements

```
dynamic.delall:[T]*
```

The `delall` method removes *all* tuples from a dynamic relation that match a given test vector. The `delall` method takes the form:

`...,`*D*`.delall(`*Term*`),...`

where *D* is the dynamic program, *Term* is a 'test' term that will be used to match potential elements of the dynamic relation. Essentially, `delall` removes all elements of *D* that match *Term*.

13.3.8 *dynamic*.delallc – Conditional delete elements

```
dynamic.delallc:[dynTest[T]]*
```

The `delallc` method removes all tuples from a dynamic relation that satisfies a query. The `delallc` primitive takes the form:

`...,`*D*`.delallc(Tst),...`

where *D* is the dynamic program, *Tst* is a 'test' labeled theory – in the same style as for `delc` (see section 13.3.6 on the previous page) – that will be used to match potential elements of the dynamic relation. Essentially, `delc` removes all elements *T* of *D* that satisfy `Tst.check(`*T*`)`.

13.3.9 *dynamic*.match – Return matching elements

```
dynamic.match:[dynTest[T]]=>list[T]
```

The `match` method returns a list of elements which satisfy the `check` predicate of the included `Tst` labeled theory:

```
D.match(Tst)
```

For example, given the `selfTop` class defined in program 13.3.1 on the facing page, the expression

```
...onTopOf.match(selfTop(onTopOf))...
```

will return, as a list, all the tuples in the `onTopOf` dynamic relation which are 'on top of themselves'.

13.3.10 Sharing a dynamic relation across threads

Dynamic relations represent resources that may be shared across threads. In order to prevent 'race conditions' where two threads compete for access to a dynamic relation, the programmer should use the `sync` action to synchronize access to the relation.

The internal methods of a `dynamic` *are* `sync`hronized; in effect guaranteeing that the defined actions in a `dynamic` relation are atomic. However, in order to support a larger grain transaction it will be necessary to use `sync` for a larger group of actions. For example, the following action removes from the `onTopOf` dynamic relation all entries that aslo satisfy the `green` predicate:

```
(green.mem(B) *> onTopOf.del((B,_)))
```

However, if the `green` and `onTopOf` relations are shared (or even just one of them is), then there can be undesirable race conditions between the evaluations of the `green.mem` and the `onTopOf.del` operations. In order to avoid this we can enclose the entire group in a `sync` action:

```
sync(onTopOf){green.mem(B) *> onTopOf.del((B,_))}
```

For this to work, of course, any other potential user of the `onTopOf` dynamic should also use the same object to `sync` on.

I/O and the File System 14

As with most modern languages I/O is not part of the language specification per se. The reason for this is that the different potential platforms for computation are so wildly different that a single model for I/O cannot fit all cases. However, Go! does come with an I/O package based on an object-oriented model of files and I/O.

In order to use the standard file system it is necessary to `import` the `go.io` package:

```
Program{
  import go.io.
  ...
}
```

This gives the `import`ing package access to the types, standard functions and standard input/output/error file channels.

14.1 Files

Go! file names are based on the URI file naming convention. A URI encodes not only the file name but also the *transport protocol* required to access the file. Common protocols include `file:` which refers to a file on the local file system, `http:` which is used to access files provided by a WWW server and `ftp:` which is used to access files using FTP service. Not all file transports support write access – in particular `http:` does not support writing .

The format of a URI as understood by Go! is:

proto`://`*host*`/`*file*

where *proto* is one of `file`, `sys`, `http` or `ftp`, *host* is either a hostname expressed using standard DNS notation or an IP address expressed using 'quads'.

If the *host* is empty then the current or `localhost` is assumed.

file The `file` protocol is used when accessing the local file system. Go! enforces its own access rights management; and it may be that a given process is not permitted to access a certain file even if the top-level Go! application is permitted. In addition, *relative* file names are interpreted relative to a *current working directory*. A relative file name is one that does not start with a leading `'/'` character.

For example, a `file` file identifier referring to a file `myFile` in the current working directory may be referred to using the file descriptor:

```
file://myFile
```

If no protocol token is given, then it is assumed that `file://` is prepended to the URI. Thus the `myFile` file may also be identified with:

```
myFile
```

http The `http` protocol uses the standard HTTP 0.9 protocol to request a WWW server to access a file. Although normally a WWW server is used to 'deliver' HTML files, other types of file may be handled by WWW servers.

ftp The `ftp` protocol uses the FTP protocol to access a file.[1] Unlike `http`, the `ftp` protocol may be used to write to files as well as read from files. However, the remote system may refuse a request to write a file.

In addition, a remote system may request a user identification and password. This is supported within the stream I/O system with an additional protocol although it is also possible to build the identification and password into the URL.

[1] Not currently implemented.

14.1.1 Character Encoding

Go! supports Unicode internally and in its access to the file system. Internally, the story for Unicode support is relatively simple: all characters are represented as 16 bit Unicode value. However, externally the Unicode story is somewhat more complex; as a Go! program is typically required to be able to read a mix of ASCII data, raw byte data, 16 bit Unicode (UTF16) and 8 bit Unicode (UTF8).

Each file channel is associated with an encoding type which indicates the mapping between the internal representations and the date represented in the file. The `ioEncoding` type is a standard built-in type and is defined:

```
ioEncoding ::= rawEncoding |
   utf16Encoding | utf16SwapEncoding |
   utf8Encoding |
   unknownEncoding.
```

The meaning associated with these encoding types is:

`rawEncoding` This form of encoding is one byte per character, regardless of the actual data. For an input channel, this is equivalent to 'normal' reading for Unicode un-aware systems. If your data is pure binary or ASCII, for example, you should use `rawEncoding` for your encoding type.

It is an error to write character data whose encoding is outside the range 0..255 to an output channel that is flagged as using `rawEncoding`.

`utf16Encoding` This form of encoding corresponds to the standard UTF16 character encoding. Each character is represented as 16 bits, or two bytes with the most significant byte first.

`utf16SwapEncoding` This form of encoding corresponds to swapped UTF16 character encoding: each character is represented as two bytes with the least significant byte first.

`utf8Encoding` This form of encoding corresponds to standard UTF8 character encoding. UTF8 encodes ASCII characters (as well as certain others) in a single byte, others are encoding as multi-byte sequences.

In many ways this encoding is the safest for reading and writing text files. However, because the number of bytes per character depends on the data being processed, it is not very suitable for random access to files. Nor is it suitable for processing binary data.

`unknownEncoding` This encoding style is most useful for reading data. The Unicode standard encourages the use of sentinel character data at the beginning of a text file. If present, the sentinel information is used to automatically determine if the encoding is actually one of `utf16Encoding` or `utf16SwapEncoding`. If no sentinel is present, then the encoding defaults to `utf8Encoding`. For an output channel, this is equivalent to `rawEncoding`.

Normally the encoding is defined at the time a file channel is opened; however, it is possible to *change* the file encoding for certain kinds of file objects. Such a change affects any remaining input/output operations on the file.

14.1.2 Standard file types

The two types `inChannel` and `outChannel` define the operations over input file objects and output file objects respectively. There are specific functions for different kinds of I/O channels: files, sockets and so on.

The `inChannel` type interface summarized in Table 14.1 on the next page.

The `outChannel` interface is summarized in Table 14.2 on the facing page.

Method	Type	Description
`inCh`	`[]=>char`	Next character
`inBytes`	`[integer]=>list[integer]`	Next N bytes
`inB`	`[]=>integer`	Next byte
`inLine`	`[string]=>string`	Read a line
`inText`	`[string]=>string`	Read text block
`pos`	`[]=>integer`	current file position
`seek`	`[integer]*`	reset file position
`eof`	`[]{}`	End of file test
`close`	`[]*`	Close file channel
`setEncoding`	`[ioEncoding]*`	Change encoding

Table 14.1: The `inChannel` interface

Method	Type	Description
`outCh`	`[char]*`	Write character to file
`outB`	`[integer]*`	Write a byte to file
`outBytes`	`[list[integer]]*`	Write list of bytes
`outStr`	`[string]*`	Write a string to file
`outLine`	`[string]*`	Write a line
`close`	`[]*`	Close the file channel
`setEncoding`	`[ioEncoding]*`	Change encoding

Table 14.2: The `outChannel` interface

14.2 Accessing Files

14.2.1 `ffile` – determine the existence of a file

```
ffile:[string]{}
```

The `ffile` predicate is satisfied if the file exists in the file system.

14.2.2 `ftype` – determine the type of a file

```
ftype:[string]=>fileType
```

The `ftype` function determines what kind of file, if any, a file is.

The `fileType` type is defined as an algebraic type:

```
fileType ::= fifoSpecial | directory | charSpecial | blockSpecia
| plainFile | symlink | socket | unknownFileType.
```

Error exceptions

`'eSTRNEEDD'` The argument should be a ground string.

`'eNOFILE'` A component of the file path does not exist.

`'eNOTFND'` The named file does not exist

`'eNOPERM'` Insufficient privileges to access file, typically a component of the file path.

`'eINVAL'` An error, possibly too many symbolic links, encountered in trying to access the file.

`'eIOERROR'` An I/O error encountered while trying to access the file.

14.2.3 `fmodes` – determine permissions of a file

```
fmodes:[string]=>list[filePerm]
```

Determine the permission modes associated with a file. The returned value is a list of `filePerm` symbols, where `filePerm` is given by the type definition:

```
filePerm ::= setUid | setGid | stIcky | rUsr | wUsr | xUsr |
rGrp | wGrp | xGrp | rOth | wOth | xOth.
```

Error exceptions

`'eSTRNEEDD'` The argument should be a ground string.

`'eNOFILE'` A component of the file path does not exist.

`'eNOTFND'` The named file does not exist

`'eNOPERM'` Insufficient privileges to access file, typically a component of the file path.

'eINVAL' An error, possibly too many symbolic links, encountered in trying to access the file.

'eIOERROR' An I/O error encountered while trying to access the file.

14.2.4 fsize – determine size of a file

```
fsize:[string]=>integer
```

Determine the size of a file. The returned integer is the number of bytes in the file.

Error exceptions

'eSTRNEEDD' The argument should be a ground string.

'eNOFILE' A component of the file path does not exist.

'eNOTFND' The named file does not exist

'eNOPERM' Insufficient privileges to access file, typically a component of the file path.

'eINVAL' An error, possibly too many symbolic links, encountered in trying to access the file.

'eIOERROR' An I/O error encountered while trying to access the file.

14.2.5 frm – remove a file

```
frm:[string]*
```

Delete the named file from the file system.

Error exceptions

'eSTRNEEDD' The argument should be a ground string.

'eNOFILE' A component of the file path does not exist.

'eNOTFND' The named file does not exist

`'eNOPERM'` Insufficient privileges to access file, typically a component of the file path.

`'eINVAL'` An error, possibly too many symbolic links, encountered in trying to access the file.

`'eIOERROR'` An I/O error encountered while trying to access the file.

14.2.6 `fmv` – rename a file

```
fmv:[string,string]*
```

Rename a file from the first argument to the second argument.

Error exceptions

`'eSTRNEEDD'` The argument should be a ground string.

`'eNOFILE'` A component of the file path does not exist.

`'eNOTFND'` The named file does not exist

`'eNOPERM'` Insufficient privileges to access file, typically a component of the file path.

`'eINVAL'` An error, possibly too many symbolic links, encountered in trying to access the file.

`'eIOERROR'` An I/O error encountered while trying to access the file.

14.2.7 `fcwd` – return current working directory

```
fcwd:[]=>string
```

The `fcwd()` function returns the current working directory.

This function should be used with care – especially if the `fcd` action is also being used. The reason is that different threads may have different views of the current working directory.

14.2.8 `fcd` – set current working directory

```
fcd:[string]*
```

The `fcd()` action procedure sets the current working directory.

14.2.9 `fls` – list of file names

```
fls:[string]=>list[string]
```

The `fls` function returns a list of the names of the files – based on the patterns of its argument. A function call of the form

```
fls(".")
```

will return a list of the files in the current working directory.

14.2.10 Open a file for reading

```
openInFile:[string,ioEncoding]=>inChannel
```

The `openInFile` function returns an `inChannel` object that represents an input channel to the file specified in the argument. Note that this name is a file name, not a URL.

The encoding argument is used to determine how to interpret incoming data: as raw (or ASCII) data, as UTF. If you don't know the encoding, use `unknownEncoding` and Go! will attempt to guess the encoding of the data based on a possible sentinel character.

Error exceptions

`'eSTRNEEDD'` The *uri* argument should be a ground string.

`'eNOPERM'` The client is not permitted to access the file system.

`'eNOFILE'` The requested file does not exist.

`'eCFGERR'` A problem arose when attempting to open the file in a mode required by Go!.

14.2.11 Access a URL for reading

```
openURL:[string,ioEncoding]=>inChannel
```

The `openURL` function opens a URI – as opposed to a file – and returns an `inChannel` object. `openURL` supports the Go! file name convention, including the possibility of using Internet based access protocols such as `ftp:` and `http:`.[1]

The encoding argument is used to determine how to interpret incoming data: as raw (or ASCII) data, as UTF.

Error exceptions

`'eSTRNEEDD'` The *uri* argument should be a ground string.

`'eNOPERM'` The client is not permitted to access the file system.

`'eNOFILE'` The requested file does not exist.

`'eCFGERR'` A problem arose when attempting to open the file in a mode required by Go!.

14.2.12 Create a file

```
openOutFile:[string,ioEncoding]=>outChannel
```

The `openOutFile` function generates a file channel that can be used for write access – in file 'create' mode – of a file. `openOutFile` supports the Go! file name convention, including the possibility of using Internet based access protocols such as `ftp:` and `http:`.[2]

The encoding argument is used to determine how to write the outgoing data: as raw (or ASCII) data, as UTF. If you require compatiblity with non-Unicode aware systems use `rawEncoding`; otherwise for character data `utf8Encoding` or `utf16Encoding` are suitable choices.

Note that it is an error to write data in `rawEncoding` when writing character data with a non-zero leading byte. The Go! system will strip off such extra data – discarding it.

[1] Currently, only `file:` and `http:` are supported.
[2] Currently, only `file:` and `http:` are supported.

If you specify `utf16Encoding` the Go! system automatically inserts a Unicode sentinel character (`\+fffe`) at the beginning of the file. This permits other UTF16-aware systems to read the character data.

Error exceptions

`'eSTRNEEDD'` The *uri* argument should be a ground string.

`'eNOPERM'` The client is not permitted to access the file system.

`'eNOFILE'` The requested file does not exist; normally this means that some intermediate directory in the *uri* does not exist.

`'eCFGERR'` A problem arose when attempting to open the file in a mode required by Go!.

14.2.13 Open file in append mode

```
openAppendFile:[string,ioEncoding]=>outChannel
```

The `openAppendFile` function generates a file channel that can be used for write access – in file 'append' mode – of a file. I.e., once opened, new data is appended to the end of the file. `openAppendFile` only supports the `file:` file name convention.

The encoding argument is used to determine how to read and write the data: as raw (or ASCII) data, as UTF. If you require compatiblity with non-Unicode aware systems use `rawEncoding`; otherwise for character data `utf8Encoding` or `utf16Encoding` are suitable choices.

Error exceptions

`'eSTRNEEDD'` The *uri* argument should be a ground string.

`'eNOPERM'` The client is not permitted to access the file system.

`'eNOFILE'` The requested file does not exist.

`'eCFGERR'` A problem arose when attempting to open the file in a mode required by Go!.

14.2.14 Connect to TCP host

```
tcpConnect:[string,integer,inChannel-,outChannel-,ioEncoding]*
```

The `tcpConnect` action procedure connects to a TCP/IP server as identified by the first parameter – which is the *host* `string` – and the second parameter – which is the *port*. If the attempt is successful then a pair of file channels are returned: the `inChannel` channel is used to read data coming from the remote connection and the `outChannel` channel is used to write data to the remote connection.

The encoding argument is used to determine how to read and write data to the remote connection: as raw (or ASCII) data, as UTF. If you require compatiblity with non-Unicode aware systems use `rawEncoding`; otherwise for character data `utf8Encoding` or `utf16Encoding` are suitable choices.

Error exceptions

`'eSTRNEEDD'` The *host* argument should be a ground string.

`'eINTNEEDD'` The *port* argument should be a ground integer value.

`'eNOPERM'` The client is not permitted to access the remote system.

`'eINVAL'` The *host* is not a legal host names or *port* is not a legal port identifier.

`'eIOERROR'` Unable to establish a connection to the remote host.

14.2.15 Spawn a new process and connect to it

```
pipeConnect:[string,list[string],list[(symbol,string)],
outChannel-,inChannel-,inChannel-,ioEncoding]*
```

The `pipeConnect` action procedure spawns a separate process – at the operating system level. An action such as

`pipeConnect(`*`Cmd`*`,`*`Args`*`,`*`Env`*`,`*`sIn`*`,`*`sOut`*`,`*`sErr`*`,`*`encoding`*`)`

results in executing the command:

Cmd Args

with environment variables set from *Env*. The command argument *Cmd* should be a fully qualified program name – it may be a script file – giving the absolute path of the program to execute.

Each element of the environment is a pair: a `symbol` denoting the environment variable to set and a `string` giving its value. This form of environment is consistent with `getenv` (see Section 17.4.1 on page 252) and `envir` (see Section 17.4.3 on page 252).

If the attempt to spawn the new command is successful then three file channels are returned: *sIn* is the new process's 'input' channel – i.e., data written to *sIn* will be read as input by the child process – output generated by the child process will be available from the *sOut* file channel object and any error output of the child process (i.e., its `stderr`) will be available via the *sErr* channel.

The `ioEncoding` argument is used to specify the encoding of the various channels to and from the sub-process.

Error exceptions

`'eSTRNEEDD'` The *Cmd* argument should be a ground string.

`'eNOPERM'` The client is not permitted to access the remote system.

`'eINVAL'` The environment was not set up correctly, or one of the string arguments was not properly formed.

`'eIOERROR'` Unable to establish a connection to the spawned process.

14.3 Reading from files

All of the following functions and actions reference an `inFile` object value as returned by a file opening operation.

14.3.1 inCh – read a character

```
inChannel.inCh:[]=>char
```

The `inCh` method of the `inChannel` class returns the next character from the input channel. If the channel is already at end of file, then an exception will be raised.

Error exceptions

`'eNOPERM'` The client is not permitted to access this input channel.

`'eEOF'` Attempted to read past end of file.

`'eIOERROR'` I/O error arose when reading from the channel

14.3.2 inLine – read a line

```
inChannel.inLine:[string]=>string
```

The `inLine` function returns the next line of text – as a string – from the `inFile` channel. A line of text is defined as the sequence of characters up to either end-of-file or until a terminating string is found – the terminating string is the `string` argument passed in to `inLine`.

To read a line that is terminated by the new-line character, use:

```
inChannel.inLine("\n")
```

Hint:

> Setting the *term* to `"\r\n"` will read a line in the 'cr-lf' form; which is useful for certain Internet protocols.

Error exceptions

`'eNOPERM'` The client is not permitted to access this input channel.

`'eEOF'` Attempted to read past end of file.

`'eINSUFARG'` The *term* string should be a ground string.

`'eSTRNEEDD'` The *term* string should be a ground string.

`'eIOERROR'` I/O error arose when reading from the channel

14.3.3 inB – read a byte

```
inChannel.inB:[]=>integer
```

The `inB` method returns the next byte from the input channel – *as an* `integer` *value.*

The actual amount of data read by this function will depend on the encoding set for the file. If the file channel is set to read UTF8 or UTF16 encoded characters, then this will return the next unicode character – a 16 bit value. Otherwise, *inChannel*`.inB` will return a value corresponding to a single byte.

Error exceptions

`'eNOPERM'` The client is not permitted to access this input channel.

`'eEOF'` Attempted to read past end of file.

`'eIOERROR'` I/O error arose when reading from the channel

14.3.4 inBytes – read a sequence of bytes

```
inChannel.inBytes:[integer]=>list[integer]
```

The `inBytes` method returns the next N characters from the input channel – *as a list of* `integer` *byte values.*

Error exceptions

`'eNOPERM'` The client is not permitted to access this input channel.

`'eEOF'` Attempted to read past end of file.

`'eIOERROR'` I/O error arose when reading from the channel

14.3.5 `inText` – read a segment of text

```
inChannel.inText:[string]=>string
```

The `inText` function returns the next block of text – as a string – from the opened input channel. The block of text is defined as the sequence of characters up to one of the characters in the argument string.

To read a line that is terminated by the new-line character, use:

```
file.inText("\n")
```

Hint:

> Where the *term* string consists of just a single character, the `inText` function will have the same effect as `inLine`. Setting the *term* string to the empty string will have the effect of reading the entire contents of the file in a single string.

Error exceptions

`'eNOPERM'` The client is not permitted to access this input channel.

`'eEOF'` Attempted to read past end of file.

`'eINSUFARG'` The *term* string should be a ground string.

`'eSTRNEEDD'` The *term* string should be a ground string.

`'eIOERROR'` I/O error arose when reading from the channel

14.3.6 `eof` – test for end of file

```
inChannel.eof:[]{}
```

The `eof` predicate is true if the input channel is at the end of file. Attempting to read past the end of file – i.e., when *inChannel*`.eof()` is true – will result in an `error` exception.

14.3.7 `close` – close input channel

```
inChannel.close:[]*
```

The `close` action closes the connection to the input channel. Subsequent attempts to read from this channel will result in a `'noPERM'` error.

14.4 Writing to files

14.4.1 `outCh` – write a character

```
outChannel.outCh:[char]*
```

The `outCh` action writes a character to the output channel.

Error exceptions

`'eNOPERM'` The client is not permitted to access this channel.

`'ECHRNEEDD'` The *C* argument should be a character.

`'eIOERROR'` I/O error arose when writing to the channel

14.4.2 `outStr` – write a string

```
outChannel.outStr:[string]*
```

The `outStr` action writes a string to the output channel.

Error exceptions

`'eNOPERM'` The client is not permitted to access this channel.

`'ECHRNEEDD'` The *C* argument should be a character.

`'eIOERROR'` I/O error arose when writing to the channel

14.4.3 `outLine` – write a line

```
outChannel.outLine:[string]*
```

The `outLine` action writes a string to the output channel, and follows it with a new-line character.

Error exceptions

`'eNOPERM'` The client is not permitted to access this channel.

`'ECHRNEEDD'` The *C* argument should be a character.

`'eIOERROR'` I/O error arose when writing to the channel

14.4.4 `outB` – write a byte

```
outChannel.outB:[integer]*
```

The `outB` action writes a single byte to the output channel. The argument to `outB` should be in the range 0..255.

Error exceptions

`'eNOPERM'` The client is not permitted to access this input channel.

`'eINVAL` A non-valid byte value was given.

`'eIOERROR'` I/O error arose when writing to the channel

14.4.5 `close` – close output channel

```
outChannel.close:[]*
```

The `close` action closes the connection to the output channel. Any subsequent attempt to write to this channel will result in a `'noPERM'` error.

14.5 Standard I/O channels

The standard version of the I/O package includes three already opened files: `stdout` and `stderr` which are output files and `stdin` which is an input file.

14.5.1 `stdin` – standard input

```
stdin:inChannel
```

The `stdin` value is an `inChannel` object that represents the standard input to the Go! application. Reading from `stdin` has the same effect as reading from the standard input.

14.5.2 `stdout` – standard output

```
stdout:outChannel
```

The `stdout` value is an `outChannel` object that represents the standard output from the Go! application. Writing to `stdout` has the effect of writing to the system's standard output.

14.5.3 `stderr` – standard error output

```
stderr:outChannel
```

The `stderr` value is an `outChannel` object that represents the standard error output channel from the Go! application.

14.6 Establishing a TCP server

Go! has a simple, yet powerful, primitive that allows an application to easily establish a TCP *server* on a particular port.

14.6.1 tcpServer

```
tcpServer:[integer,serverProc,ioEncoding]*
```

This action establishes a new TCP server that is listening on a specified port:

```
tcpServer(9999,SrP,utf8Encoding)
```

The `tcpServer` action waits – i.e., blocks the Go! thread that is executing this action. When a remote client establishes a connection with a Go! server then a subsidiary Go! thread is `spawn`ed off to handle the new connection; the `spawn`ed thread executes the action

```
SrP.exec(Host,hostIP,port,inP,outP)
```

where *host* is the hostname of the computer making the connection, *hostIP* is its IP address (both of which are `string`s), the local *port* assigned to this connection, and the input channel (where the handler thread reads data from the connection) object `inP` and the output channel object – where the handler writes to the connection.

The IP address is a string in so-called quad form. Note that the host name is not necessarily reliable: it is possible to spoof hostnames; therefore you should only use this field for informational purposes.

The object `SrP` passed in to the `tcpServer` action should be of type `serverProc`:

```
serverProc <~ { exec:[string,string,integer,inChannel,outChannel]*
```

When the `SrP.exec()` action procedure terminates the connection to the remote client will be closed – unless it already been closed through an error in the I/O handling.

The `ioEncoding` argument is used to control the encoding of character data passing between the server and the remote connections.

The `tcpServer` action itself does not normally terminate – unless it was not possible to establish the TCP listening port or unless a permission predicate raises an exception during a test of an incoming connection. If it is desired, it is possible to spawn a separate thread whose purpose is to establish and execute the `tcpServer`, while other threads continue with application activities.

Error exceptions

`'eINTNEEDD'` The *port* argument should be an integer value.

`'eNOPERM'` The client is not permitted to establish a TCP listener.

`'eCFGERR'` A problem arose when attempting to establish the listener.

Communicating processes 15

The synchronized action permits processes to share resources, but it is a poor tool for enabling inter-process coordination – because sync does not offer a direct way of allowing one thread to communicate with other threads. In Go! we use a *message passing* paradigm to enable inter-process coordination. The two key actions involved in message communication are message send and message receive. The former involves a *message dropbox* and the latter a *message mailbox*.

The facilities discussed in this are part of the go.mbox package, accessing them requires an import statement:

```
import go.mbox.
```

15.1 Mailboxes and dropboxes

The dropbox[T]/mailbox[T] metaphor corresponds – loosely – to the public dropbox in which mail is dropped and the private mailbox from which messages are retrieved. The idea is that if two (or more) processes wish to communicate, they send messages by using dropboxes and read them from their mailboxes.

Mailboxes and dropboxes support a *conversational* model of communication: each mailbox/dropbox pair denotes a conversation between processes. There is no particular implied mapping of mailboxes and dropboxes to particular Go! threads; thus it supports inter-application cooperation as easily as intra-application cooperation between threads.

The mailbox type is defined as:

```
mailbox[T] <~ { next:[]=>T.
    nextW:[number]=>T.
    msg:[T]*.
    msgW:[T,float]*.
    pending:[]{}.
    dropbox:[]=>dropbox[T] }.
```

The `next` function is used to read the next message from the mailbox, the `nextW` function combines reading with a timeout; the `pending` predicate is true if there is at least one message in the `mailbox` and the `dropbox` function returns a `dropbox` object that can be used to deliver messages to this `mailbox`.

The `dropbox` type is simpler:

```
dropbox[T] <~ { post:[T]* }.
```

The single method in the `dropbox` interface is the `post` action. `post` is used to deliver a message to the associated mailbox.

Note that `mailbox`es and `dropbox`es are *polymorphic*. In fact, they are polymorphic in the type of the messages associated with the mailbox. In effect, mailboxes and dropboxes form a kind of *typed channel* of communication. The most paradigmatic instantiation of a message type is a type defined using a type definition of enumerated symbols and constructor functions – see Sections 2.4.2 on page 51 and 2.4.3 on page 52.

Note that the `mailbox` and `dropbox` interfaces may be realized in multiple ways. The `go.mox` package permits threads to communicate with each within a single Go! invocation, not between invocations. Note, furthermore, that a given `mailbox` may have a number of different `dropbox` implementations – with differing communication capabilities – all targeted at delivering messages to a given mailbox.

15.2 Using a dropbox

15.2.1 Message send

dropbox[T].post:[T]*

The `post` method is used to deliver a message to the `mailbox` associated with the `dropbox`. In principle, a `mailbox` may have more than one way of delivering messages to it; with a different `dropbox` for each technique.

Note that the sending process is given no immediate response to the message send. If required – which is often – the sender should wait for a response using a message receive action.

15.3 Using a mailbox

15.3.1 Creating a local mailbox

```
mailbox:[]@>mailbox[T]
```

The `mailbox` constructor is used to construct a `mailbox` entity that can be used to receive messages.

Note that a `mailbox` should itself be used in a message: you should not send a mailbox in a message to another thread (say). Thus `mailbox`es are inherently owned by a single-thread. Of course, it is quite possible to send the `dropbox` associated with the `mailbox` in a message to another thread – or to store it in a well known location.

15.3.2 `nullhandle` – an empty dropbox

```
nullhandle:[]@=mailbox[T]
```

The `nullhandle` constructor is an empty dropbox – all messages posted to it will be discarded.

15.3.3 Retrieving the `next` message

```
mailbox[T].next:[]=>T
```

A message can be retrieved from a `mailbox` using a combination of the `next` function (or the `nextW` function) of the mailbox

and the `case` action. Both functions get the first message and returns it; their behavior differs in the case that there is no first message: the `next` function suspends until there is a message, and the `nextW` suspends for a limited time – its argument in seconds. If there is no message in that time then `nextW` raises a `timedout` exception.

The raw `next` function returns a T value that encapsulates the message. Note that there is no immediate way to confirm the sender of the message – that information should be encoded in the message value itself if it is required.

15.3.4 Next matching message

```
mailbox[T].msg:[T]*
```

The `msg` action can be executed to retrieve a *matching* message. Invoking the `msg` action on a `mailbox` will cause its queue of messages to be searched looking for a message that matches the argument.

If there is no message in the `mailbox`, then the `msg` action *suspends* until a matching message is received in the `mailbox`.

When a matching message is found, it is *unified* with the argument of `msg`; thus `msg` also *returns* the found message.

The `msg` action deletes the message once one is found.

As with the raw `next` function, there is no immediate way to confirm the sender of the message – that information should be encoded in the message itself if it is required.

15.3.5 `nextW` – a time limited message receive

It is possible to 'time out' a message receive – by using the `nextW` function instead of the `next` function:

```
mailbox[T].nextW:[number]=>T
```

This function returns the same value as the `next` function, but it differs in behavior should there be no message in the `mailbox`.

If there is no message in the `mailbox` for the given period of time, then `nextW` `raises` a `timedout` exception. `timedout` is an error value defined within the `go.mbox` package to signal that a timeout has occurred.

It is the responsibility of the receiver to ensure that the `timedout` exception is caught appropriately:

```
...;case Mbx.nextW(0.5) in (
  Ptn1 -> Act1
| ...
  Ptn1 -> Act1
) onerror (
  timedout -> TimeoutAction
  );...
```

This fragment looks for a message, and if there is none for 0.5 seconds after the start of the call to `Mbx.nextW(0.5)` then the `timedout` exception is raised and the *TimeoutAction* is entered.

The numeric argument of the `nextW` function refers to a number of seconds (it may be fractional) of real elapsed time; it does not refer to processor time.

The start time of time-out is calculated from just before there is any attempt to read any messages; and the timeout is invoked only if there are no messages in the message queue.

Using timed `nextW` to access messages from the mailbox without care in the choice of timeout values is likely to result in programs that have bugs that are hard to detect – since there is always some non-determinism in the order of execution in multi-threaded applications. Furthermore, especially for networked applications, computing the appropriate values to assign for timeouts is likely to be problematic given the enormous variation in network latency delays.

15.3.6 Wait for matching message

mailbox[T]`.msgW:[T,number]*`

The `msgW` action is similar to the `msg` action except that a timeout is given. The `msgW` action will either find a matching message in the `mailbox` or will wait for the given timeout – expressed as a number of seconds starting from the beginning of the `msgW` action.

If no matching message is delivered to the `mailbox` within the *timeout*, then a `timedout` exception is raised.

If a matching message is found, it is *unified* with the argument of `msg`; thus `msg` also *returns* the found message.

The `msgW` action deletes the message once one is found.

15.3.7 `pending` – predicate to check for messages

```
mailbox[_].pending:[]{}
```

The `pending` predicate is satisfied if there are any messages currently in the `mailbox` queue.

Note that in order for this predicate to have a well defined *fluent* the call to `pending` – together with any consequent actions – should occur inside a `synchronized` region. Otherwise messages may added or even removed – if the mailbox is also shared across threads – invalidating the truth value of the `pending` predicate.

15.3.8 `dropbox` – return the dropbox associated with a mailbox

```
mailbox[T].dropbox:[]=>dropbox[T]
```

The `dropbox` function returns a `dropbox` for the associated `mailbox`. This dropbox can be used locally to send messages to the mailbox.

Time and dates 16

Go! has a number of primitives for handling dates and times. There are two key structures; a time value – represented as a `float` – and a date value – represented as a `date` type.

16.1 Time

16.1.1 `now` – return current time

```
now:[]=>float
```

The `now` function returns the current time – expressed as a `float` – which represents the number of seconds since Jan 1, 1970 GMT. To convert this value into a `date` value, use the `time2date` function (see Section 16.2.2 on page 247). The number returned is potentially fractional; the resolution of the clock is implementation dependent but is at least in milliseconds.

16.1.2 `today` – return time at 12:00am

```
today:[]=>integer
```

The `today` function returns the time at midnight this morning – expressed as the number of seconds since 12:00am, Jan 1^{st} 1970. The effect is to return a number that represents today's date.

16.1.3 `ticks` – return CPU time used

```
ticks:[]=>float
```

The `ticks` function returns the number of seconds of CPU time – expressed as the number of seconds – consumed by the Go! invocation since it started. The number returned is potentially fractional; the resolution of the clock is implementation dependent but is at least in milliseconds.

The `ticks` function is best used differentially – take the `ticks()` measure before some important timing and again after the timed event. The difference will tell you how long the event took.

16.1.4 `delay` – delay for a period of time

```
delay:[number]*
```

The `delay` *action* causes the currently executing thread to delay for a given number of seconds. Other threads may continue executing while this thread suspends. `delay` can accept either an `integer` number of seconds or a fractional number of seconds – expressed as a `float`.

16.1.5 `sleep` – sleep until a given time

```
sleep:[number]*
```

The `sleep` *action* causes the currently executing thread to suspend until a given time. Other threads may continue executing while this thread suspends. An action call of the form:

```
sleep(now()+10)
```

is equivalent to:

```
delay(10)
```

16.2 Dates

The `go.datelib` package provides a number of higher-level functions that support manipulating dates. These functions center on

the `date` type – a type interface for representing dates and times related to the *wall clock*.

To use these functions, you will need to import the `datelib` package:

```
import go.datelib.
```

16.2.1 The `date` type

The `date` type interface represents a date and time value, it is summarized in:

```
date <~ {
  time:[]=>float.                          -- raw time
  clock:[]=>(integer,integer,float). -- The time in ordinary time
  date:[]=>(integer,integer,integer,integer). -- Broken out date
  tzone:[]=>float.    -- Time zone
  zone:[]=>string.    -- Name of the time zone
  dow:[]=>dow.        -- Day of the week
}.
```

Where `dow` is an algebraic type capturing the days of the week:

```
  dow ::= monday |
  tuesday |
  wednesday |
  thursday |
  friday |
  saturday |
  sunday.
```

clock The `clock` function returns the time associated with the `date` value – as a triple:

`(`*`Hours`*`,`*`Minutes`*`,`*`Secs`*`)`

where

`Hours` is an integral `number` from 0 (midnight) to 23

Minutes is an integral `number` from 0 to 59

Seconds is a `float`ing point value in the range $[0, 60)$

date The `date` function returns the date associated with the `date` value – as a 4-tuple:

`(`*Dow*`,`*Day*`,`*Month*`,`*Year*`)`

where

Dow is an `integer` indicating the day of the week: 0 = sunday, 6 = saturday.

Day is an `integer` in the range 1..31, indicating the day of the month

Month is an `integer` in the range 1..12, indicating the month, with 1=January

Year is an `integer` indicating the year. Dates with a *Year* less than 1900 are not guaranteed to be valid.

time This function returns the `float`-style time value associated with the `date` – i.e., a number representing the number of seconds since Jan 1, 1970.

tzone The `tzone` function returns the `number` of *hours* that this `date` value is offset from UTC.

zone The `zone` function returns the *name* of the time zone associated with this `date` value.

dow The `dow` function returns a `dow` (day of the week) value. The `dow` type is an enumeration of the possible days of the week:

```
dow ::= monday | tuesday | wednesday | thursday
      | friday | saturday | sunday.
```

less The `less` predicate allows the comparison of two `date` values. A query of the form:

```
Dte.less(Rte)
```

is true if the date and time associated with `Dte` is *before* the date/time associated with `Rte`.

16.2.2 `time2date` – convert a time value to a `date`

```
time2date:[float]=>date
```

The `time2date` function converts a time value – expressed as a number of seconds since Jan 1, 1970 – into an instance of the `date` class. As a result, the date in calendrial terms is computed.

The date is computed relative to the default location that the Go! system is executing in; i.e., the time is returned in local terms.

16.2.3 `time2utc` – convert a time value to a `date`

```
time2utc:[float]=>date
```

The `time2utc` function converts a time value – expressed as a number of seconds since Jan 1, 1970 – into an instance of the `date` class. As a result, the date in calendrial terms is computed.

The date is computed as a UTC time (i.e., as used to be known GMT). The merit of this is that the value returned by `time2utc` is globally consistent; the problem is, of course, that for those who do not live in the same time zone as Greenwich, the time will be somewhat different to the local time – possibly including the date also.

16.2.4 `dateOf` – return a `date`

```
dateOf:[integer,integer,integer,integer,integer,
number,number]=>date
```

The `dateOf` function returns a `date` value, computed from the arguments; which are the Day, Month, Year, Hours, Minutes and seconds and the hour offset from UTC.

We provide the `dateOf` function, rather than simply allowing a direct use of a `date`-valued constructor, because of the calculations needed to ensure a valid `date` structure.

16.2.5 `rfc822_date` – parse a date in RFC822 format

```
rfc822_date:[date-]-->string
```

The `rfc822_date` grammar parses a date in the format specified by RFC822. A typical RFC822 date looks like:

```
Wed 23 Nov, 2005 10:14:34 UT
```

The returned value in a `rfc822_date` is a `date` object.

Miscelleneous 17

These miscellaneous library primitives access and manipulate the environment in which the Go! program executes.

17.1 Equality, matching and identicality

There are many notions of equality in Go!: two terms may be unifyable, 'matchable' and may be identical.

17.1.1 = – unifiability test

```
=:[t,t]{}
```

The = predicate is true if its two arguments are unifiable – i.e., can be made to be identical after potential substitution of values for unbound variables. = is a standard operator, and so = predicates are written in infix notation.

The = predicate is so fundamental to Go! that it should be considered part of the definition of the language.

17.1.2 .= – match test

```
.=:[t,t]{}
```

The .= predicate is satisfied if it is possible to make its arguments identical without substituting any variables in the second. I.e., `X .= Y` is satisfied if X and Y can be made to be identical without binding a variable in Y; otherwise the .= test will *fail*. It is permitted to bind unbound variables in X, however.

The `.=` predicate implements the same matching semantics as for the heads of equations, action rules and message receive clauses.

17.1.3 == – identicality test

```
==:[t,t]{}
```

The == predicate is satisfied if its arguments are identical without substituting any variables in either argument; otherwise the == test will *fail*.

17.2 Variables, terms and frozen terms

17.2.1 var – test for variable

```
var:[t]{}
```

The `var` predicate is satisfied if its argument is a variable, and fails otherwise.

17.2.2 nonvar – test for variable

```
nonvar:[t]{}
```

The `nonvar` predicate is satisfied if its argument is *not* a variable, and fails otherwise.

17.2.3 ground – test for groundedness

```
ground:[t]{}
```

The `ground` predicate is satisfied if its argument is *ground*, and fails otherwise.

A term is ground if it does not contain any variables. This test applies to all types of values – including program values. A program is considered to be ground iff its *free* variables are ground

– a program's bound variables are universally quantified and we consider a term of the form: $\forall X.X$ to be ground.

17.3 Internet hosts

These two functions allow programs to compute IP addresses and hostnames of computers on the Internet. Their use requires an on-line connection to the Internet.

17.3.1 `hosttoip` – determine IP address

```
hosttotip:[string]=>list[string]
```

The `hosttoip` function returns the IP addresses associated with the *host* computer. The returned list is a list of strings, each of which is an IP address in normal 'quartet' form. Note that it isn't possible for this to be a guaranteed complete list – it depends on the local DNS server.

Error exceptions

`'eSTRNEEDD'` A ground string is required.

17.3.2 `iptohost` – determine host name

```
iptohost:[string]=>string
```

The `iptohost` function returns the host name associated with a given IP address. The returned list is a string giving the hostname.

Note that the form of the hostname is not guaranteed to be 'canonical' – i.e., the full hostname including domain. The Go! system attempts to determine the full hostname but improperly configured local systems will be able to mystify Go!.

Error exceptions

`'eSTRNEEDD'` A ground string is required.

17.4 Environment variables

These access and manipulate the environment variables that are often available for program to customize their operation.

17.4.1 `getenv` – access environment variable

```
getenv:[symbol,string]=>string
```

The `getenv` function expects a ground `symbol` argument – the environment variable name – and a string (`char[]`) *default* argument. If the environment variable is set in the program's current environment then the value of the environment variable is returned as a string. Otherwise, the `getenv` function returns the *default* value.

17.4.2 `setenv` – set environment variable

```
setenv:[symbol,string]*
```

The `setenv` action expects a `symbol` argument – K – and a string argument V; both parameters must be ground. This action sets the value of the K environment variable to be V.

17.4.3 `envir` – return all environment variables

```
envir:[]=>list[(symbol,string)]
```

The `envir` function returns all the environment variables available to the program. Each environment variable is 'presented' as a 2-tuple: the first element of the tuple is a symbol denoting the name of the environment variable and the second is a string denoting the variable's value.

17.4.4 `getlogin` – access login name

```
getlogin:[]=>string
```

The `getlogin` function returns the user name (login ID) of the 'owner' of the process executing this Go! application.

17.5 Program and thread management

These allow threads to monitor the state of other threads and wait for the termination of a thread.

17.5.1 `__command_line` – command line arguments

```
__command_line:[]=>list[string]
```

The `__command_line` function returns a list of strings corresponding to the program name and command line arguments passed to the Go! run-time engine.

17.5.2 `exit` – terminate Go! execution

```
exit:[integer]*
```

The `exit` action terminates the current execution of the Go! system and returns its argument as the process's return code to the operating system.

Error exceptions

`'eNOPERM'` This is a privileged action, and the calling thread has insufficient permissions to close down the Go! engine.

`'eINVAL'` A non-integer was passed as the return code to return from the Go! invocation.

17.5.3 `kill` – terminate a Go! thread

```
kill:[thread]*
```

The `kill` action terminates a thread identified by its argument. Only the creator thread of a thread may kill it.

Error exceptions

`'eNOPERM'` This is a privileged action, and the calling thread has insufficient permissions to kill threads.

`'eINVAL'` The to-be-killed thread is not a valid local thread.

17.5.4 `process_state` – access process state

```
process_state:[thread]=>process_state
```

The `process_state` function expects a `handle` argument and returns a symbol that identifies the current state of that thread:

`quiescent` The thread has not yet executed any instructions.

`runnable` The thread is currently one of the actively executing threads.

`wait_io` The thread is currently suspended waiting for an I/O event.

`wait_term` The thread is waiting for another thread to terminate.

`wait_timer` The thread is waiting for an alarm clock.

`wait_child` The thread is waiting for a child process – as opposed to a thread – to terminate.

`dead` The thread has died.

`process_state` is a standard type, whose definition is:

```
process_state ::=
  quiescent | runnable | wait_io | wait_term |
  wait_timer | wait_lock | wait_child | dead.
```

Error exceptions

`'eINSUFARG'` The *H* argument should be a handle, not a variable.

`'eINVAL'` The *H* argument is not a valid local handle.

17.5.5 `waitfor` – wait for a thread to terminate

```
waitfor:[thread]*
```

The `waitfor` standard action suspends the current process until the *H* thread has terminated. If *H* has already terminated then the `waitfor` action simply continues; otherwise the calling thread is suspended until *H* terminates.

Error exceptions

`'eINSUFARG'` The *H* argument should be a handle, not a variable.

`'eINVAL'` The *H* argument is not a valid local handle.

17.5.6 `__shell` – execute shell command

```
__shell:[string,list[string],list[(symbol,string)],number]*
```

The `__shell` action invokes a *Cmd* program or shell script with the arguments constructed from the argument list in the environment constructed from the environment argument.

Each element of the environment is a symbol/string pair of the form:

`(`*envVar*`,`*value*`)`

If environment is empty, then sub-process'es environment has the default minimum number of environment variables set.

The return code resulting from the execution of the program is returned as the value of *Ret*.

Note: In a multi-threaded Go! application, if one thread issues a `__shell` command, other threads emphdo not suspend waiting

for the shell command to terminate. This allows an application to spawn off more than one shell command. However, the thread issuing the `__shell` action is suspended until the spawned process terminates.

XML processing 18

Go! provides a simple library for processing XML documents. This library provides programs for 'grabbing' an XML document, parsing it into a DOM-like structure and displaying it. Processing XML documents is the foundation for many other forms of data processing.[1]

Go!'s XML processing is based on string processing. I.e., a typical mode of operation is to first of all 'grab' an XML document in a string – `list[char]` – and then parse it using `xmlParse`. Conversely, to display an XML document, we first of all convert it to a string, and then write it out on the appropriate channel.

To use Go!'s XML processing facilities, it is necessary to access the `go.xml` package:

```
import go.xml.
```

18.1 Go!'s XML document type

The `xmlDOM` type provides the foundation for Go!'s XML processing. It is defined as:

```
xmlDOM <~ showable.
xmlDOM <~ {
  pickElement:[symbol]=>xmlDOM.
  elementPresent:[symbol].
  hasAtt:[symbol,string]{}.
```

[1] This version of the XML parser is not a complete XML 1.0 compliant parser: it does not understand external DTDs, nor does it understand all of XML 1.0's features. This may be extended in a future release.

```
  pickAtt:[symbol]=>string.
  pickText:[symbol,string]=>string.
  xml:[]=>string.
}.
```

The `xmlAttr` type encapsulates attributes associated with the `xmlDOM` entity (especially the `xmlElement` constructor):

```
xmlAttr ::= xmlAtt(symbol,string).
```

An `xmlDOM` term corresponds to the XML infoset view of an XML document. A node has one of three forms: an `xmlElement`, an `xmlText` element or an `xmlPI` processing instruction. Of these, the most complex is the `xmlElement` structure.

Note that `xmlDOM` is a sub-type of the `showable` type, which itself is in the `go.showable` package. However, it is not required for any package that is simply *using* the `xmlDOM` type to also import the `go.showable` package.

pickElement The `pickElement` function returns the first child element whose name is that given.

`pickElement` raises a `'eNOTFND'` exception if the element is not present.

elementPresent The `elementPresent` predicate is satisfied if the named element is present as a child.

hasAtt The `hasAtt` predicate is satisfied if the named attribute is present.

pickAtt The `pickAtt` function returns the value of the named attribute if present. Note that the XML specification permits multiple values for an attribute of an element; however, the `pickAtt` function returns an arbitrary member if there is more than one.

If the attribute is not present, then an `'eNOTFND'` exception will be raised.

xml The `xml` function returns an XML string representation of the `xmlDOM`.

Note that while most of the methods are not meaningful for all `xmlDOM` constructores, the `xml` function is.

18.1.1 xmlText

```
xmlText:[string]@=xmlDOM
```

Text appearing in an XML document is collected in an `xmlText` term. Note that the parser removes empty text elements during the parse, and strips off leading and trailing white space from other text elements.

The parser supports standard entities (see Section 18.1.5 on the following page); which are substituted for in the body of the `xmlText` term.

Note that `xmlText` does *not* support many of the methods in the `xmlDOM` type – for example, invoking the `pickElement` function will result in an `'eINVAL'` exception being raised. The `xmlText` constructor is a state-free constructor, and the expectation is that normally it will be matched against rather than used.

18.1.2 xmlPI

```
xmlPI:[string]@=xmlDOM
```

Processing instructions appearing in the XML document are returned in an `xmlPI` term.

18.1.3 xmlElement

```
xmlElement:[symbol,list[xmlAttr],list[xmlDOM]]@=xmlDOM
```

A tagged element is represented as an `xmlElement` term. This has three arguments, corresponding to the tag, the list of attributes and the list of child elements.

The parser automatically converts local tag names into their fully expanded form when there is a name-space declaration; including the default namespace declaration.

For example, the XML fragment:

```
<foo xmlns="http:myNameSpace#" id="bar">
  <subfoo/>
</foo>
```

is represented, using `xmlDOM` terms, as:

```
xmlElement('http:myNameSpace#foo',
  [xmlAtt('http:myNameSpace#id',"bar")],
  [xmlElement('http:myNameSpace#subfoo',[],[])])
```

18.1.4 xmlAtt – attributes

```
xmlAtt:[symbol,string]@=xmlAttr
```

Atributes in an `xmlElement` are represented using the `xmlAtt` term. Each attribute `xmlAtt` term has two arguments: the attribute name and the value of the attribute – as a string.

The Go! xml parser expands standard entities in attribute values.

18.1.5 Standard entities

The Go! parse (and display) functions are aware of a restricted set of standard entities. It is not possible to declare new entity definitions. The standard entities that the parser is aware of are shown in Table 18.1:

Entity	Definition
&amp;	&
&lt;	<
&gt;	>
&apos;	'
&excl;	!
&quot;	"

Table 18.1: Go! standard entities

18.2 Parsing XML documents

There are two programs in the standard xml library for parsing documents: `grabURL` which is a convenience function for accessing the contents of a URI and `xmlParse` which is a grammar program for parsing strings.

18.2.1 grabXML

```
grabXML:[string,string]=>(string,xmlDOM)
```

The `grabXML` function takes two `string` arguments: a 'base' url (B), a 'request' url (U) and returns a pair – the fully resolved url and the contents of the document found at that url, parsed as an `xmlDOM` document. The request url may be relative, in which case it is interpreted relative to the base url B.

Error exceptions [1]

`'eNOTFND'` It was not possible to access the document at the resolved location.

18.2.2 xmlParse

```
xmlParse:[xmlDOM-]-->string
```

The `xmlParse` grammar parses a string which contains an XML document and returns the `xmlDOM` structure corresponding to the parse.

The `xmlParse` parser does not validate XML documents, nor does it process DTDs. However, it is 'namespace' aware: documents can be processed with namespace declarations and recognized properly.

The parser is not currently a full XML parser; Table 18.2 on the following page is an enumeration of the features of XML and the extent to which they are supported. Although the parser records

[1] See Chapter 21 on page 283 for the definition of the standard error codes

XML 1.0 Feature	Supported
Attributes	yes
Character references	yes
Comments	yes
Conditional sections	no
DTD	no
Document declaration	no
Empty Tags	yes
Entities	predefined only
Namespaces	yes
PCDATA	yes
Processing Instructions	yes
Tags	yes
Validation	no
Well-formedness	partial

Table 18.2: XML 1.0 features

processing instructions, there are no PIs that it recognizes directly.

18.3 Displaying XML documents

18.3.1 xmlDisplay - show XML structure

```
xmlDisplay:[xmlDOM]=>string
```

The `xmlDisplay` function returns a string XML representation of an `xmlDOM` term. It uses the local tag and attribute names in constructing the displayed string.

`xmlDisplay` is 'entity' aware: characters in the values of attributes and in `xmlText` elements that require representing as elements in order to conform to XML are handled correctly.

`xmlDisplay` is an extremely simple generator, optimized for performance rather than elegance of output. In particular, it does not 'pretty print' the XML structure into an easily readable form.

18.4 Miscelleneous functions

18.4.1 hasNameSpace – Look for a name space

```
hasNameSpace:[xmlDOM,string]{}
```

The `hasNameSpace` predicate is true of an `xmlElement` if has an `xmlns` attribute – i.e., if its default namespace is known. The second argument is the name space.

This predicate is equivalent to:

```
hasNameSpace(X,N) :- X.hasAtt('xmlns',N).
```

Lexical parser generator 19

The `Golex` tool is part of the standard Go! system, but it is not a normal package. It is a tool that takes in the specification of a lexical parser (strictly a parser based on regular expressions) and generates a Go! program that can parse `strings` into a list of tokens.

The `Golex` tool, and the `Golex` language, is closely modeled on the `lex` languae tool. Of course, it is slightly modified to be convenient for Go! rather than C, and its implementation is not based on `lex`'s implementation.

19.0.2 The parts of a `Golex` file

A typical `Golex` program looks like:

```
-- The preamble ...
import go.stdparse.
yyTokType ::= FLT(float) | ID(string) | EOF.
%%
-- The rules ...
 [ \n\t\b\r]+ => skip      -- ignore white space
 "--"[^\n]*"\n"          -- line comment
 "-"?[0-9]+("."[0-9]+([eE][-+]?[0-9]+)?) => (FLT(floatOf%%yyTok))
 [a-zA-Z_][a-zA-Z_0-9]* => (ID(yyTok))
-- Block comment uses a state
 "/*" => <comment>         -- long comment
  <comment> "*/" => <initial>
  <comment> .            -- implies a skip
%%
```

```
-- The postamble
  private parseAll:[yyLexer]*.
  parseAll(Lx) ->
    Tok = Lx.nextToken();
    stdout.outLine("Token is "<>Tok.show());
    ( Tok.token()!=EOF?
        parseAll(Lx)).
  main([])::stdin.eof() -> .
  main([]) ->
      parseAll(yyTest(stdin.inLine("\n"),0,0));
    main([]).
```

There are three parts to the `Golex` program: the preamble, the rules section and the postamble.

preamble The preamble section of a `Golex` file has to satisfy two key obectives: define the type of tokens found by the lexer and to ensure that any required `import` declarations are made.

If there are any `import` statements, they should be at the beginning of the section.

The `yyTokType` type declaration is mandatory and declares the types of tokens generated by the lexer.

There is an additional requirement that may be removed in a future version: the constructor `EOF` must be declared for the `yyTokTpe`.

rules The rules section is typically the largest section and contains all the rules that make up the lexer.

postamble The postamble section contains any definitions – with their declarations – that are referenced by the rules.

19.1 `Golex` rules

19.1.1 Anatomy of a rule

The general form of a `Golex` rule is:

`[<State>] Matcher => Body`

where the *<State>* is optional (as is the body). If the body is omitted, then so is the arrow.

We discuss states in more detail in Section 19.3 on page 271.

The *Matcher* is a regular expression (see Section 19.2 on the following page) that defines the strings that the rule 'fires' over.

The *Body* decides how to interpret a successful match of the regular expression. There are three possibilities: to ignore the string, to produce a token expression, or to change state.

skip If the body of the rule consists of the word `skip`, or is omitted, then the effect is to ignore the matched string. No output is generated as a result of matching the matcher.

A classic use of this is, of course, to implement a comment facility in a lexer.

expression If the *Body* is text enclosed in parentheses, then it is interpreted as a Go! expression. Some conditions apply to this expression:

- The type of this expression must be `yyTokType`, or a sub-type of it.
- If the expression refers to the pseudo-variable `yyToken`, then that is replaced by a `string` that denotes the entire string that was matched by the rule.
- If the expression refers to the pseudo-variable `yyLine`, then that is replaced by an `integer` that represents the line number of the beginning of the matched string.
- If the expression refers to the pseudo-variable `yyLLine`, then that is replaced by an `integer` that represents the line number of the end of the matched string.
- If the expression refers to the pseudo-variable `yyPos`, then that is replaced by an `integer` that represents the number of characters in the complete string that are before the matched string.

- If the expression refers to the pseudo-variable `yyLPos`, then that is replaced by an `integer` that represents the number of characters in the complete string that are up to the end of the matched string.

The expression returned by the body of the rule is returned as the value of lexer itself, in particular, it is the returned value of the `nextToken` method applied to the lexer object.

state If the *body* is a state-name enclosed in `<>` characters then the matched string is ignored, but the state of the lexer is switched to the named state.

19.2 Regular expressions

The matcher part of a rule takes the form of a regular expression. The language of regular expressions supported by `Golex` is very similar to the regular expression language of `lex` itself. This should aid in using `Golex`.

A regular expression consists of a sequence of elements of the form:

- A string of characters, enclosed in `"` characters or `'` characters and following Go!'s standard escape conventions, matches exactly that string and no other.

- A period matches any character except the new-line character.

- A character set pattern of the form:

  `[`*charSeq*`...`*charSeq*`]`

  where *charSeq* is either a regular Go! character sequence or a range triplet of the form:

  charSeq`-`*charSeq*

matches any of the explicitly identified characters, or in the case of a range triplet any of the characters in the range.

For example, the pattern:

```
[a-z_]
```

matches any lowercase ASCII letter and the underscore character.

There are some special considerations in the character set pattern: in order to include either of the `[]` characters themselves in the set, they should be escaped with a `\` character. In order to include a `-` character it should be the first or the last character in the set, or escaped with a `\`. In order to include the `^` character it should not be the first character in the set.

- A negative character set pattern of the form:

  `[^`*charSeq*`...`*charSeq*`]`

  will match any character except those quoted.

- A pair of regular expressions separated by the `|` character indicates disjunction. I.e., a pattern of the form:

  P_1 `|` P_2

  will match any string that matches either of P_1 or P_2.

- A pair of regular expressions with no separating characters is considered to match strings consisting of consequtive substrings that match the two component patterns.

- A regular expression followed by a `*` will match strings which match the left pattern any number of times (including zero).

  For example, the pattern:

  ```
  [a-z]*
  ```

matches any sequence of lowercase letters (including the empty set).

- A regular expression followed by a `+` will match strings which match the left pattern at least once. For example, the pattern:

  ```
  [a-z]+
  ```

  matches any sequence of lowercase letters (not including the empty string).

- A regular expression followed by a `?` will match strings which optionally match the left pattern. For example, the pattern:

  ```
  [-+]?
  ```

  optionally matches a character which is either `-` or `+`.

  Note that this is optional only in the sense that the string may have a `-` or `+` character in it. If one of those characters were present the matches must match it!

- A regular expression enclosed in parentheses matches a string if the embedded regular expression matche the string.

  The pattern language does not have any relative precedences for the regular expression operators. Hence, if there is any potential for ambiguity, then the regular expression pattern must fully parenthesized.

- The regular expresion `eof` only matches the empty string at the end of the original input string.

- Any other character appearing in a regular expression pattern is considered an error, and will result in an error message generated by the `Golex` tool.

19.3 Lexer states

The set of rules in a `Golex` file may be partitioned into named *states*. Each state defines a set of rules that only apply to that state. In fact, if a rule does not have an explicit identifying state, its state is the default state – also identified by the name `<initial>`.

Partitioning rules in this way is a powerful tool for implementing certain kinds of situation. For example, one might write a block comment rule:

```
"/*" => <comment>          -- long comment

  <comment> "*/" => <initial>
  <comment> .            -- implies a skip
```

The leading `/*` characters cause a switch into the `<comment>` state. In this state, all characters are ignored (this follows from the single period matcher with an empty body) until the `*/` characters are matched. The rule for `*/` causes another switch, this time to the `<initial>` state, which is the default state.

If we did not have this concept of states, we would have to write the comment rule in a somewhat more elaborate form:

```
"/*"([^*]|\*[^/])*"*/"
```

The somewhat simpler rule:

```
"/*".*"*/"
```

is not sufficient because the period `.` does not match a newline.

19.4 Using a `Golex` lexer

The `Golex` tool expects a rule file to have the extension `.glx`. Applying the `Golex` tool to this file results in a Go! program of the same name and defining a package of the same name. This program must be compiled by the normal Go! compiler before it can be used.

```
% golex lexer.glx
% goc lexer.go
```

The Golex tool defines in *lexer*.go a constructor whose name is yy*lexer* that can be used to parse strings.

The type of yy*lexer* is defined by the declaration:

```
yylexer:[string,integer,integer]@>yyLexer.
```

where the first argument is the string to be analysed, the second is a notional start line and the third is a notional start position; and where yyLexer is defined using the type interface:

```
yyLexer <~ {
  nextToken:[]=>yyToken.
  currentToken:[]=>yyToken.
}.
```

and yyToken itself is defined via the interface:

```
yyToken <~ {
  token:[]=>yyTokType.
  isToken:[yyTokType]{}.
  line:[]=>integer.
  pos:[]=>integer.
}.
```

The yyTokType type is the type defined within the .glx file itself.

The methods in the yyToken type are:

token This returns the token expression that was returned by the tokenizer. It ultimately refers to the body of a rule in the Golex source file.

isToken This is essentially a predicate form of the token function:

```
Tk.isToken(T) <=> Tl.token()=T
```

line This is the number of line numbers encountered in the original string before this token was recognized.

pos This is the number of characters that are before the recognized sub-string in the original string.

The `yy`*`lexer`* constructor constructs an object that can parse a string. It should be invoked with the string to parse, a number indicating the number of the first line and a number indicating the first character position:

```
Lexer = yylexer(Input,0,0)
```

The case for non-zero values for the second and third arguments is when the lexer itself is only invoked on a fraction of the original input.

To actually recognize a token, the `nextToken` function should be invoked from the tokenizer object itself:

```
Tok = Lexer.nextToken()
```

The resulting object, which is of type `yyToken`, can be inspected for the returned token, where its line number and character position is.

Part III

Using Go!

Running Go! programs 20

20.1 Compiling a Go! program

Before an Go! program can be executed, the source code must first be compiled into abstract machine instructions – this is done using a separate compiler program.[1] The Go! compiler is invoked at the command-line using the `goc` command as follows:

`goc [-g] [-gb` *progName*`]* [-b` *progName*`]* [-p] [-P` *dir*`]*` *filename*`*`

where *file*s specify one or more Go! source files. The options have the following meanings:

`-g` – **enable debugging** Results in extra debugging information included with the compiled code. This extra code is activated When the compiled program is run with the '`-g`' option – see section **??** on page ?? for one example of how to trace the execution of a Go! program.

`-b` *progName* – **set break point** If debugging is enabled, the break option requests that a break point is set on entry to a particular program. When execution reaches that program then the debugger will be notified of the break point.

`-gb` *program* – **enable debugging for** *program* **only** The plain `-g` option enables debugging for all programs in a package. The `-gb` option enables debugging for specific programs only; other programs will not have debugging information compiled for them.

[1] For the Technology Preview Release the Go! compiler requires the `April`language system to be installed.

The compiler will generate debugging information for *any* program of the given name – whether it is defined inside a class body or at the top-level.

`-p` – **enable profiling** Compiles extra information that can be used to help profile performance of the executing program. When a program is compiled with the `-p` option, then, when the program is run, a file `goProfile.out` is generated that contains a trace of the execution of the program. This file can be processed to extra performance statistics – such as the number of times each line of source is visited.

The standard Go! program `go.profiler` is a simple program that analyses the results of a `goProfile.out` file, generating a listing of the number of times each line of the program is executed.

`-P` ***dir*** – **Add to compiler path** Adds *dir* to the end of the 'class path'. The class path is a list of directories which is used to locate source files *and* pre-compiled modules that are `import`ed.

The default compiler path is:

```
.:/opt/go
```

assuming that `/opt/go` is the installation point of your Go! system.

The compiler is aware of the `GO_DIR` environment variable. If set then this environment variable overrides the default compiler path.

The *file*s are assumed to contain Go! source code. The standard extension for Go! source programs is *.go*. Note that the compiler enforces the convention that the 'package name' of the program reflects the file name. For example, if a program file contained the package:

```
foo.bar.jar{
  ...
}
```

then this must be contained in a file whose name takes the form:

```
.../foo/bar/jar.go
```

I.e., the tail of the complete file name must match the declared package name in the file.

The reason for this is that is a program file `import`s a package, as in:

```
...
  import foo.bar.
...
```

then the compiler will search for a file of the form `.../foo/bar.goc` occurring on the class path.

20.2 Running a Go! program

A Go! program is run using the `go` runtime engine command:

`% go` *options prog* Arg_1 ... Arg_n

where *prog* is a package contained in a file `prog.goc` occurring on the class path; and Arg_i are the program arguments to be passed to the Go! program.

By default, the standard entry point of a program is an action procedure called `main` – this takes as its argument a list of `string`s – the strings forming the arguments to the application. The entry point can be changed by using the `-m` option.

The command line arguments may be parsed – into `integer`s for example – using the facilities provided by the `go.stdparse` package. For example, to parse the first argument into a `integer` use:

```
...
  main([F,..R]) ->
    doSomething(integerOf%%F);
    ...
```

Command line options are always consumed by the engine. If there are arguments that follow the package name they are *not* processed by the engine and are passed as normal arguments to the Go! application. For example, to pass application specific command-line options use a command such as:

`go` *`package`* `-a1 -a2 ...`

to start the Go! application. The arguments `-a1`, `-a2` etc., are passed as is to the Go! program; and can be accessed using the standard `__command_line` function (see Section 17.5.1 on page 253).

`-v` – **display version information** This option causes the engine to display a line indicating the version of the engine before executing the program.

`-h` *N* – **set initial size of global heap** This option sets the initial size of the global heap to $N \times 1024$ cells. The global heap will grow automatically as required; however, this may require a number of expensive garbage collections before the system decides the heap should be larger.

The default value for the initial global heap size is 200K cells (on a 32bit system, slightly less than 1MB) setting a larger value for some programs may improve performance.

`-s` *N* – **set default initial thread heap size** Each thread has its own local heap, on which most terms are constructed during unification. The default initial size of the thread heap is 1024 cells; the `-s` option will change that to $N \times 1024$ cells.

Again, the system will automatically grow a thread's heap if required. However, performance may be improved by setting the default to a larger value. Note that this sets the initial heap size for *all* threads created in the application.

`-m` *`entry`* – **non-standard entry point** By default, the Go! runtime executes the `main` action procedure occurring in the loaded program. However, the `-m` flag can be used to invoke a different program. Which ever entry point is specified, it must be a single argument action procedure that takes a list of `string`s as its argument.

`-P` ***dir*** **– add *dir* to the class path** The `-P` option adds the named *dir* to the front of the class path. This list of directories is used to locate the compiled code of packages that are loaded – either directly as the main program package – or indirectly when that package (or other packages) `import` additional packages.

Like the compiler, the Go! engine is aware of the `GO_DIR` environment variable. If set then this environment variable overrides the default execution path:

`/opt/go/:`*`cwd`*`/`

is used – where `/opt/go/` is the *installation directory* that Go! is installed in and *cwd* is the current directory.

`-d` ***dir*** **– set non-default initial working directory** Normally the initial working directory for the Go! engine is based the value of the `PWD` shell environment variable.[1] Using the `-d` option allows you to override this with a different directory.

`-g` **– invoke debugger** This option turns on any debugging code that was compiled into the application by the `-g` option of the compiler. More precisely, the Go! loads the `go.debug` package and initializes the package with the command line option.

`-L` **URL – set logfile** Occasionally the Go! engine will report messages in a logfile. By default the logfile is equivalent to `stderr`. Using the `-L` option allows log messages to be recorded in a file.

If it is used, the `-L` option should be the first command line option.

`-R` ***seed*** **– Set randomization seed** By default the random number generator uses the same seed for every execution. However, setting the initial *seed* – which should be a large integer for maximum randomness – will help to ensure more random random numbers.

[1] Strictly, the value returned by the `getcwd()` system call.

Standard error codes 21

21.1 Error handling

Many of the built-in primitives may raise an exception as an alternative to failing or succeeding. This is primarily for those situations where a failure would not represent an appropriate response. For example, an attempt to divide by zero will result in a error exception.

Error exceptions may be trapped by user-level Go! programs (see errorhanding); or in the final analysis an uncaught exception will cause the Go! application to be terminated.

The description for each built-in primitive gives a listing of the different error `code`s generated by that primitive. In this section we give a complete listing of the standard error codes and some explanation of the meaning of the error.

21.1.1 The standard `exception` type

Exceptions raised, either internally by a library function or explicitly in a user-level program, are represented by `exception[]` values. This interface is defined as:

```
exception <~ { cause:[]=>string. code:[]=>symbol }.
```

The `cause` function returns a string representing a reason for the exception; however, for most programs the `code` function represents a more language-neutral of determining the kind of exception.

In addition to the `exception` type, Go! also defines in its standard library the `error` constructor. This class is used by most

of the built-in functions when they wish to report an error to the caller of the function:

```
error:[string,symbol]$=exception.
error(R,C)..{
  cause()=>R.
  code()=>C.
  show()=>"error: "<>R<>": ("<>__errorcode(C)<>")".
}
```

21.2 Standard exception code symbols

`'eABORT'` The `eEOF` error exception is raised when the user has requested that a thread be aborted. This is typically only possible if the program is being debugged.

`'eASSIGN'` The `eASSIGN` is raised when a program attempts to assign to a variable when it is not permitted. Assignment is only permitted within action rules that are executed 'at the top level' of a thread; in particular, assignment is not permitted within `action` sequences or `valof` sequences.

`'eCFGERR'` The `eCFGERR` is raised when a configuration request was no possible. A standard place where this occurs in the file I/O system; Go! attempts to ensure that it can access external files in a non-blocking interrupt driven mode. If that is not possible then a `eCFGERR` error exception will be raised.

THe `eCFGERR` error is also raised when there is a conflict between the configuration of a channel and the attempted operation on it; for example, attempting to read or write encoded terms to channels that are not configured to `rawEncoding` mode will result in this error.

`'eCHRNEEDD'` The `eCHRNEEDD` is raised when an argument to a primitive is not a `char` value – this normally means that a `char` was expected but an unbound variable was passed in.

`'eCODE'` The `eCODE` is raised when attempting to execute something which is not executable – most commonly when trying to call an unbound variable.

The nature of the Go! language means that it is not always possible to determine at compile time that all 'program variables' are bound to executable code. As a result, on occasion, one may find an `'eCODE'` error being raised.

`'eCONNECT'` The `eCONNECT` is raised when attempting to perform a connection to a remote server and it is denied for some reason. The actual problem will vary, from the server not actually listening to the port to the server not permitting you to make the connection.

`'eDEAD'` The `eDEAD` exception is raised when a deadlock is detected. This occurs principally when using locks and the `sync` action. It is especially easy to get into a 'deadlock' situation when attempting to `sync` on more than one lock; in that situation make sure that the nesting order of `sync` actions is the same for all occurrences.

For example, if one action sequence is:

```
sync(L1){
  sync(L2){
    Action1
  }
}
```

and another action sequence – executing in another thread – using the same locks is:

```
sync(L2){
  sync(L1){
    Action2
  }
}
```

then a deadlock is possible because both sequences may execute their first `sync` action; but then they will both be deadlocked on their second `sync`.

Deadlock detection is not infallible, as it relies on the fact that there are no other executable processes. It is quite possible for two threads to deadlock, but to have other threads active at the same time.

`'eDIVZERO'` The `eDIVZERO` error is raised by builtin primitives when the application program attempts to divide a number by 0.

For example, a call of the form:

```
...,foo(1/0),...
```

would result in an `eDIVZERO` error. An active error handler would have to have a clause that matched:

```
error("/",'eDIVZERO') :- ...
```

to catch this error.

`'eEOF'` The `eEOF` error exception is raised when an attempt is made to read past the end of file of some input file.

`'eFAIL'` The `eFAIL` error is raised when a function or a procedure fails. Functions are assumed to be 'total' on their arguments and if none of the equations in a function definition apply to the arguments then an `'eFAIL'` error exception will be raised. Similarly, if none of the action rules in a procedure match the arguments then, rather than backtracking, an `'eFAIL'` error is also raised.

Recall that the heads of both equations and action rules are *matched* against the arguments of the calls rather than being unified (see Section 3.6 on page 77). This has the effect of increasing the probability of a call to a function or procedure failing – since matching is not permitted to bind variables in the call. However, such a 'success' in the case of functions

and procedures would likely be erroneous since long tradition has it that functions do not 'side-effect' their arguments.

For example, in the function `fact` below, there are rules for the case of zero and for positive numbers, but no case for negative numbers:

```
fact(0)=>1.
fact(N)::N>0 => N*fact(N-1).
```

A call to the `fact` function with a negative argument would raise the error:

```
error("fact",'eFAIL')
```

`'eINSUFARG'` The `eINSUFARG` is raised when an argument to a primitive is insufficiently instantiated. Typically this is when an argument is a variable when it should not be.

`'eINSUFTPE'` The `eINSUFTPE` is raised when the type associated with an `??` expression is non-ground. In general, non-ground types are not safe when wrapped in an `??` expression.

`'eINTNEEDD'` The `eINTNEEDD` is raised when an argument to a primitive is not a whole number. Typically this is raised by primitives such as `band` which implement a bitmap interpretation of numeric values (see 9.3.1 on page 161).

`'eINVAL'` `'eINVAL'`

The `eINVAL` is raised when an argument to a primitive is not valid for some reason.

`'eINVCODE'` The `eINVCODE` is raised when a particular type of program is expected and the actual type is incorrect.

`'eIOERROR'` The `eIOERROR` is raised when there is an operating system error to do with I/O. Typically, this is caused when the device being written to is full, or when a socket connection 'breaks'.

`'eLSTNEEDD'` The `eLSTNEEDD` is raised when an argument to a primitive is not a list.

`'eNOFILE'` The `eNOFILE` is raised when attempting to access a resource (typically a file) which does not exists.

`'eNOPERM'` The `eNOPERM` is raised when attempting to access a resource for which the task (or the user running the Go! application) does not have permission.

`'eNOTFND'` The `eNOTFND` is raised when a file or other system resource is not found.

`'eNUMNEEDD'` The `eNUMNEEDD` is raised when an argument to a primitive is not a number.

`'eOCCUR'` The `eOCCUR` exception is raised when a unification attempts to construct a 'circular' term. For example, unifying `X` with `f(X)` would result in an infinite structure if permitted. This is the so-called 'occurs check' violation.

Most logic programming systems do not even check for occurs check because of the potential run-time overhead. However, it is not safe, and the Go! system attempts to avoid to perform the check when it is known to be safe (such as the first time a variable is unified).

`'eRANGE'` The `eRANGE` is raised when an argument to a primitive is out of the range of permissable values for that particular operation.

`'eSPACE'` The `eSPACE` exception is raised if the system cannot function in the amount of memory it has. This is generally a fatal error.

`'eSTRNEEDD'` The `eSTRNEEDD` is raised when an argument to a primitive is not a list of symbols – i.e., when it is not a string.

`'eSYMNEEDD'` The `eSYMNEEDD` is raised when an argument to a primitive is not a symbol.

`'eSYSTEM'` The `eSYSTEM` error exception is raised when a system limit is exceeded. A typical case of this is when unifying terms that are so complex that internal buffers are exceeded.

Note that not all system overflows result in a 'recoverable' exception. If total system memory is exhausted, for example, then an unrecoverable error is raised and the Go! engine will be terminated.

`'eVARNEEDD'` The `eVARNEEDD` is raised when an argument to a primitive is not an unbound variable – and the primitive is not able to handle unification.

`'eUNIFY'` The `eUNIFY` is raised when attempting to compare two 'incomparible' values – such as two code values. Although Go! is a higher-order language, it is not a complete higher-order language – in the sense that higher-order unification is not part of the language.[1] Whenever two code values – such as functions or relations – are compared then an `eUNIFY` error exception will be raised.

[1] Higher order unification is not decidable; which makes incorporating it into a programming language problematical.

Installing Go! A

This appendix gives instructions on installing Go! and `April`on a Unix-based system. It is possible to install and use both on a Windows system, under Cygwin; however this is beyond the scope of this chapter to explain how.

Warning:

> The information in this appendix is subject to change and may be different in your situation.

A.1 Getting Go!

The installation of Go! requires the installation of three packages: `ooio`, `April`and Go! proper. For a number of reasons, Go! is currently only distributed as source.

The `ooio` library is a basic library that supports many functions of the other two packages.

The `April`language system is a full programming language. It is used only to compile Go! programs, however.

The Go! language system is the complete engine, compiler and documentation.

A good place to get the source tarballs for these packages is

`http://homepage.mac.com/frankmccabe`

Assuming that you have the files `ooio-`*`date.tgz`*, `april-`*`date.tgz`* and `go-`*`date.tgz`* then the process involves first of all compiling and installing `ooio`, then `April`, and then doing a similar job for Go!.

A.2 Installation directory

The systems `ooio`, `April`and Go! are designed to be installed by default in the directories `/opt/ooio`, `/opt/april` and `/opt/go` respectively. It is possible to set up both to install in different locations; for example in your home directory. However, we *do not* recommend installing either in the 'standard' installation directories `/usr/bin` and/or `/usr/local/bin`. However, it is certainly possible to install *links* to the `April`/Go! compilers and run-time systems in those locations.

The installation directory is important as it includes a number of files that are important for the smooth running of both `April`and Go!.

You can make the appropriate directories in the default locations using:

```
% mkdir /opt/nar /opt/april /opt/go
% chown <yourUserName> /opt/nar /opt/april /opt/go
```

A.3 Building the `ooio` library

Unpack the `ooio-`*`date.tgz`* file and enter the `ooio` top-level directory. The `ooio` library is set up to use the `configure` script – which was automatically generated using a combination of `automake` and `autoconf`.

To configure `ooio` run the `autogen.sh` script:

```
ooio % ./configure [--prefix=dir]
...
```

It is only necessary to supply the `--prefix` argument if `/opt/ooio` is not the preferred installation directory.

Once configured, compile the library using the `make` command:

```
ooio % make all
```

If you own the target directory, then you can simply:

```
ooio % make install
```

to install it; otherwise, try:

```
ooio % sudo make install
```

remembering, of course, that the password the system will ask for is your's, not that of root.

A.4 Building April

The process to build and install Aprilis similar to that for the ooio library; with the additional wrinkle of dealing with a non-standard location for the ooio library.

Unpack the april.tgz tarball and enter the april5 directory.

To configure Aprilrun the configure script:

```
april5 % ./configure [--prefix=dir] [--with-ooio=ooioDir]
...
```

It is only necessary to supply the --prefix argument if /opt/april is not the preferred installation directory, and the --with-ooio argument is only needed if you installed ooio in a non-standard place.

Once configured, compile Aprilland install it using the make command:

```
april5 % make all install
```

A.4.1 Paths

Since April(and Go!) are designed to install in their own directories, it is useful to extend the path environment to include them. You can do this in your .bash_login file in your home directory[1]:

```
export PATH=$PATH:/opt/april/bin:/opt/go/bin
```

[1]The process for doing this if you use a different shell will be similar.

A.5 Building Go!

The process for setting up Go! is exactly analogous to that for `April`; excepting that if you have installed `April`in a non-standard directory, you need to inform Go!'s configuration script:

```
go % ./configure[--with-april=dir] [--with-ooio=ooioDir]
```

Making Go! is also similar:

```
go % make
```

There is one additional step in building Go!: checking that everything is running Ok. The command:

```
go % make check
```

will check out quite a few of the features of Go!. If this test does not terminate normally then is there is likely a problem that needs to be sorted out.

Assuming that the test works out, install Go!:

```
go % make install
```

A.6 Setting up EMACS

The Go! (and `April`) systems come with special modes for the GNU EMACS editor.[1]

To access the EMACS modes for Go! and `April`, you will need to modify your `.emacs` file. Just edit that file (in EMACS of course), and make sure that the following lines:

```
(add-to-list 'load-path "/opt/go/share/emacs/site-lisp")
(add-to-list 'load-path "/opt/april/share/emacs/site-lisp")

;;; April mode
(autoload 'april-mode "april")
(setq auto-mode-alist
```

[1]Editor does not do justice to EMACS, it is more like a language/editor/operating system; if a little old-fashioned.

```
  (cons '("\\.ap\\|.ah$" . april-mode) auto-mode-alist))
(add-hook 'april-mode-hook 'turn-on-font-lock)

;;; Go! mode
(autoload 'go-mode "go")
(setq auto-mode-alist
  (cons '("\\.\\(go\\|gof\\|gh\\)$" . go-mode)
    auto-mode-alist))
(add-hook 'go-mode-hook 'turn-on-font-lock)
```

alltt in there somewhere.

With Go! mode turned on, EMACS knows how to indent Go! programs according to the informal style formatting rules (very similar to those used in this book).

A.7 Go! Reference

The Go! reference manual will be located in the file:

`/opt/go/Doc/go-ref.pdf`

Go! Emacs mode B

Included in the Go! system is a simple Emacs mode that makes editing and debugging Go! programs simpler. See Section A.6 on page 294 for details of how to modify your Emacs environment to access the Go! mode.


This mode is very much a work in progress, especially the debugging aspect of the mode.

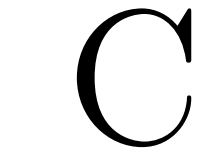
B.1 Editing Go! programs

B.2 Debugging Go! programs in Emacs

The `C-c C-d` command initiates a Go! session under the debugger. It takes two arguments: the name of the package to debug and a list of arguments to pass as command line arguments to the debugged process.

See Section C.1.1 on page 299 for instructions on how to use the default debugger from within Emacs.

Debugging Go! programs



The Go! system supports debugging using a combination of two core techniques: the compiler insert additional code when compiling with the `-g` option, and the Go! engine supports the monitoring of such programs.

C.1 The default debugger

The default `go.debug` package is a simple debugger that may be used to debug Go! programs. However, it is very simple in its capabilities; its primary purpose is to facilitate the use of a system such as Emacs to handle the actual debug display and interaction.

In this section we summarize the Emacs interface to using the default debugger.[1]

C.1.1 Debugging a package

In order to invoke the debugger on a package, the Emacs command `C-C C-D` (control-C control-D) command is used.

There are two arguments to this command: the name of an Emacs buffer containing the program to be debugged and the sequence of arguments to the program.

By default, the currently visited Go! buffer is the one that will be debugged. It must have a definition of `main` in it.

Arguments are passed to the debugged program in an analogous way to the command line: run-time switches and arguments

[1] This assumes that the Emacs environment has been set up so that `Go-mode` is active when you are editing a Go! program.

are typed in to the Emacs mini-buffer and passed to the Go! system. The `-g` switch itself is not needed: Emacs assumes that you are debugging a program, not just running it! By default, no additional arguments are passed to the debugged program.

You can use `C-P` (control-P) to recall a previously entered set of arguments to pass to the debugged program.

Once the program is initialized, the Emacs window is split into two panes: a trace pane and a source pane. The trace pane shows the commands and results going to the debugged program and the source pane shows where in the source of the program you are.

When the debugger is waiting for input, it shows a standard prompt:

`[`*`thread`*`] (go.Debug) ?`

Normally, this prompt is shown just before entering some program or evaluating some expression within a program that has had debugging code enabled. The prompt will not show for programs that are not specially compiled.

The following commands are available:

`n` The `n` command (typing the `n` key) executes the call that the debugger was waiting on and reports the result in the trace window.

Note that if a call has sub-expressions then each of the sub-expressions will be paused on separately.

The source pane shows where in the program the debugged program is currently located.

Note that the trace window only shows an abbreviated form of the call:

```
call foo/2
```

To show the whole call, use the `x` command. Or use one of `1 - 9` to show one of the first 9 arguments.

`s` The `s` command steps into the call, and continues the tracing/debugging cycle within the defined program.

`0` The `0` command displays the name of the program being called.

`1 ...9` The `1` command shows the first argument. The digits `1` through `9` result in the corresponding argument of the call being displayed.

`x` The `x` command displays the whole call.

`V` The `V` command displays all the variables that are known in the current rule. Not that this may differ slightly from the variables in the source rule as the compiler performs significant amount of transformation of complex Go! rules.

`v` The `v` command takes a single argument – the name of a variable – and displays that variable in the trace pane.

`c` The `c` command continues the execution of the program with no further tracing – until a break point is hit; at which point debugging may resume.

No output other than normal output from the program is displayed during this mode.

`t` The `t` command continue execution in the same way as `c`, except that the trace pane displays all the programs as they are entered and left.

`q` The `q` command terminates the Go! program.

C.2 A debugging strategy

The Go! compiler uses an auxiliary debug package – `go.debug` – to implement the run-time aspect of debugging a program. The strategy is very similar to the approach that many programmers have to debugging: they add write statements to the code to figure out what is happening. The only real difference is that the 'write statements' becomes calls to the `go.debug` package.

The standard debug package may be replaced with a user-defined one; in particular the `-G` *`pkg`* command line option may be used to override the debugger for a particular execution of a Go! program.

Any alternate implementation of a debugger should `import` the standard `go.debug` package in order to access the correct `debugger` interface as described below.

As with other packages, there is a type interface associated with the debugger. The `debugger` type defines the contract between the program being debugged and the debugger:

```
debugger <~ {
  line:[string,integer,integer,integer]*.
  break:[symbol]*.

  evaluate:[symbol,list[thing],symbol,
            list[(symbol,thing)]]*.
  value:[symbol,thing,symbol,list[(symbol,thing)]]*.
  prove:[symbol+,list[thing]+,symbol+,
         list[(symbol,thing)]+]{}.
  succ:[symbol+,list[thing]+,symbol+,
        list[(symbol,thing)]+]{}.
  call:[symbol,list[thing],symbol,
        list[(symbol,thing)]]*.
  return:[symbol,symbol,list[(symbol,thing)]]*.
  parse:[symbol+,list[thing],list[thing],symbol,
         list[(symbol,thing)]]{}.
  parsed:[symbol+,list[thing],symbol,
          list[(symbol,thing)]]{}.

  asgn:[symbol,thing]*.
  vlis:[thing]*.
  trigger:[symbol,thing]*.
  rule:[symbol,integer,list[(symbol,thing)]]*.
  xrule:[symbol,integer]*.
}.
```

line The `line` method is invoked quite liberally by a debugged program. It is intended to be used to inform the user where in the source the program is currently at.

The form of a `line` call to the debugger object is:

```
Dbg.line(file,line,sPos,ePos)
```

where *Dbg* is the currently active debugger object.

The *file* is the name of the source file, as it was identified by the compiler. The integer *line* identifies the line number where the feature occurs in the source file and the integers *sPos* and *ePos* bracket the *character offsets* of the feature from the beginning of the file.

The *sPos*/*ePos* pair gives a more precise indication of where the feature is, but is more difficult to interpret manually.

It is the responsibility of the debugger to ensure that the user is made aware of where the program is currently. How that is done will vary of course. The default debugger package (see Section C.1 on page 299) simply prints a message giving the line number.

Note that the debugger should not pause at this point. In particular, it should not wait for user input. The individual program entry methods are used by the debugged program when user input is expected.

break The `break` method is invoked if the user requested that a particular function be interrupted with an entry to the debugger. This is useful when trying to focus on a problem in a particular sub-part of the program.

The argument to `break` is the name of the stopped program:

```
Dbg.break(Program)
```

The debugger should respond to this method by enabling debugging if it is not enabled, and by ensuring that the debugger will pause at the next suitable moment.

evaluate The `evaluate` method is invoked prior to calling a function. The first argument is the name of the function being evaluated and the second is the list of arguments:

$$\texttt{Dbg.evaluate(}Program\texttt{,[}A_1\texttt{,...,}A_n\texttt{],}Key\texttt{,}$$
$$\texttt{[(}V_1\texttt{,}Val_1\texttt{),...,(}V_n\texttt{,}Val_n\texttt{)])}$$

The first argument is the name of the function being called, the second is the list of arguments to the function call. The third argument is a *Key* that is unique in the package to this call and the final argument is a list of `symbol`/value pairs that gives the current values of the variables that are in scope.

Some debuggers may choose to display it as a function call, others may choose to only display the function name – with perhaps the arity.

The reason for displaying the entire call may be obvious. However, if the arguments to the call are large, then the user may be unnecessarily slowed down since a large argument to a function call is quie likely to be followed by large arguments to subsequent calls.

The *Key* argument is a unique symbol that identifies this call to the function. It is used to help match up calls with returns (see the `value` method below).

In normal circumstances the debugger should accept some command input from the user and determine the intentions of the user from that input.

The standard debugger only displays the program name and the arity of the call. The reason for this is that a call's arguments can be very large, and to display them every time can quickly become tedious.

The `X` command displays the call in its entirety.

value The `value` method is called after a function has returned. The first argument is the name of the function, and the second is the returned value:

```
Dbg.value(Program,Value,Key,
          [(V_1,Val_1),...,(V_n,Val_n)])
```

The *Key* argument can be used to resume debugging appropriately if debugging/tracing was suspended for the call to the function. As with the `evaluate` method call, the list of variables with their known values reflects the current state, in this case after the call to the function.

prove The `prove` method is called when trying to prove a predicate condition. Note that this may invoked in failure mode also, in which case the debugger should report that the predicate condition could not be proved. Otherwise, the `prove` is analogous to the `evaluate` method.

The first argument is the name of the relation being queried, the second is a list of the arguments to the query:

```
Dbg.prove(Program,[A_1,...,A_n],Key,
          [(V_1,Val_1),...,(V_n,Val_n)])
```

The *Key* argument is a unique symbol that identifies this call to a function. It is used to help match up calls with returns (see the `value` method below). The (V_i, Val_i) list contains all the local variables and their current values.

In normal circumstances the debugger should accept input from the user, determine the intentions of the user from that input.

Note that the failure mode of `prove` arises when the predicate condition fails. Since the `prove` call is before the actual call to the predicate condition, the failure of the predicate condition will be propagated back into the `prove` condition. The standard debugger has two clauses for `prove`: the first one handles the case where the predicate is entered into for

the first time, and the second one handles the case where the predicate failed.

In the latter case, the debugger prints a message and then fails; propagating the failure back further still: perhaps to a real choice point in the program.

succ The `succ` method is called when a predicate condition was successfully proved. Note that this may invoked in failure mode also, in which case the debugger should report that the predicate condition is going to be re-attempted, and *then the debugger should also fail.*

The argument to `succ` is the name of the relation:

Dbg.succ(*Program*,*Key*,[(*V*$_1$,*Val*$_1$),...,(*V*$_n$,*Val*$_n$)])

The *Key* argument can be used to resume debugging appropriately if debugging/tracing was suspended for the call to the relation. As with the `prove` method call, the list of variables with their known values reflects the current state, in this case after a successful proof.

call The `call` method is called when entering an action procedure call. The first argument is the name of the action procedure, and the second is the list of arguments to the call:

Dbg.call(*Program*,[*A*$_1$,...,*A*$_n$],*Key*,
[(*V*$_1$,*Val*$_1$),...,(*V*$_n$,*Val*$_n$)])

return The `return` method is called when a completing an action procedure call. The argument is the name of the returning action procedure:

Dbg.return(*Program*,*Key*,[(*V*$_1$,*Val*$_1$),...,(*V*$_n$,*Val*$_n$)])

parse The `parse` method is called when trying to parse a string. Note that this may invoked in failure mode also, in which case the debugger should report that the grammar nonterminal could not be recognized, and then fail.

The first argument is the name of the grammar non-terminal. The third is the list of arguments to the non-terminal and the second is the stream that is to be parsed. Often, this is a `string` but it may be any `list`:

```
Dbg.parse(Program,Stream,[A1,...,An],Key,
          [(V1,Val1),...,(Vn,Valn)])
```

parsed The `parsed` method is called when a grammar non-terminal was successfully recognized. Note that this may invoked in failure mode also, in which case the debugger should report that the non-terminal is going to be re-attempted, and *then the debugger should also fail.*

The first argument is the name of the parsed non-terminal. The second is the remaining stream:

```
Dbg.parsed(Program,Stream,Key,
           [(V1,Val1),...,(Vn,Valn)])
```

rule The `rule` method is called on successful entry to a rule of some kind. Its arguments include a list of all the variables in that rule. The rule may be any of the kinds of rule that Go! supports:

```
Dbg.rule(Program,No,[(V1,Val1),...,(Vn,Valn)])
```

The *Program* is the name of the program being entered, and *No* is the number of the rule being entered – the first rule is rule 1.

xrule The `xrule` method is called when a rule is left normally (i.e., not on failure). The arguments to `xrule` are the program name and the rule number:

```
Dbg.xrule(Program,No)
```

asgn The `asgn` method is called when an object variable, or a package variable, is given a value. The first argument is the name of the variable, and the second its value:

Dbg.asgn(*VarName*,*Value*)

vlis The `vlis` method is called when a `valis` action is executed, given the value return for that `valof` expression. The argument is the value being returned:

Dbg.vlis(*Value*)

trigger The `trigger` method is called when a delayed variable is triggered. The two arguments to `trigger` are the name of the variable and its value when triggered:

Dbg.trigger(*VarName*,*Value*)

GNU Common Public Licence

D

Version 2, June 1991

51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the

rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

GNU General Public License

Terms and Conditions For Copying, Distribution and Modification

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or

translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   (a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

   (b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

(c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

(a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

(b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

(c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Pro-

gram except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy

both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

   Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of

following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

No Warranty

11. Because the program is licensed free of charge, there is no warranty for the program, to the extent permitted by applicable law. Except when otherwise stated in writing the copyright holders and/or other parties provide the program "as is" without warranty of any kind, either expressed or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. The entire risk as to the quality and performance of the program is with you. Should the program prove defective, you assume the cost of all necessary servicing, repair or correction.

12. In no event unless required by applicable law or agreed to in writing will any copyright holder, or any other party who may modify and/or redistribute the program as permitted above, be liable to you for damages, including any general, special, incidental or consequential damages arising

OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

> *one line to give the program's name and a brief idea of what it does.*
> Copyright (C) *year name of author*
>
> This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.
>
> This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

> You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

> Gnomovision version 69, Copyright (C) *year name of author*
> Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type 'show w'.
> This is free software, and you are welcome to redistribute it under certain conditions; type 'show c' for details.

The hypothetical commands `show w` and `show c` should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than `show w` and `show c`; they could even be mouse-clicks or menu items—whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

> Yoyodyne, Inc., hereby disclaims all copyright interest in the program
> 'Gnomovision' (which makes passes at compilers) written by James Hacker.
>
> *signature of Ty Coon*, 1 April 1989
> Ty Coon, President of Vice

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking

proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

Glossary

action procedure A set of rules that defines a behavior in a program. Contrast, for example, with a *predicate* program which defines a relation.

bag of A list-valued expression whose value is determined by finding all solutions to a predicate condition.

basket A basket type lattice is one where every chain is exactly three elements long; effectively meaning that there is no significant subtype relationship between non-trivial type elements.

bounded set A list valued expression whose value is determined by applying a predicate test to all elements of a base list.

callback A *callback* is an interface that is implemented by the client code of an interface. Typically, callbacks are used to allow a library package to invoke specific functionality within the client to act as a kind of test. Callbacks are the object ordered analogue of lambda functions being passed in to higher order processing functions such as list map.

cell A primitive resource that supports a re-assignable memory model.

chain A chain is a sequence of types where for each pair of types in the link T_i and T_{i+1} it known that T_i is a subtype of T_{i+1}. All types are in a chain of at least three elements: `void`, the type itself and `top`.

character catagories A classification of characters introduced by the Unicode consortium to abstract common roles of character glyphs.

class label A class label is a term that is associated with a theory – a set of axioms that collectively describes a concept.

conditional A form of condition that applies a test to select one of two branches to apply. There are conditional forms of expression, action, predicate and grammar condition.

conditional expression An expression whose value is one of two possible values – depending on the success or otherwise of a predicate test.

exception recovery expression An expression which incorporates a exception handler – if an exception is `raised` during the evaluation of an expression then its evaluation is terminated and an error handler executed instead.

fluent A *fluent* is a predicate with a particular extent in time: fluents have a starting time and an ending time (although it may not always be clear what those end points are in all situations.)

forall A condition that is satisfied if a dependent goal is satisfied whenever the governing goal is satisfied. For example, all the children of someone are male, is a typical forall condition.

guarded pattern A pattern whose semantics is governed by a predicate: a unification or pattern match of the pattern with a term is deemed to succeed only if the guard is satisfied.

negation A predication which is satisfied iff its embedded predication is *not* satisfied. Go! uses negation-as-failure semantics negation.

non-terminals A part of a grammar that is itself defined by other grammar rules.

one-of A predicate condition that may only be satisfied once – the evaluation will not seek alternate solutions to a one-of condition once a successful solution has been found.

package A package is a group of related definitions and programs that is separately compiled and loaded. Packages may contain classes, rules and types. Packages are accessed with the `import` statement.

polymorphic A type that is not completely ground – it may refer to many kinds of values.

predication A syntax form that expresses a condition that must be satisfied.

raise exception expression Not a normal expression – evaluation of a `raise` exception expression results in the exception being raised – and the current evaluation aborted up to an enclosing exception handler. The value of the `raise` exception expression is used by the exception handler.

recursive A recursive type is one which may have components of the same type as the overall type.

spawn expression An expression denoting an action that has been spawned to executed as an independent thread of execution.

strong clause A form of clause that has an if-and-only-if semantics. All clauses in a predicate must be either regular or strong. In the latter case, the predicate is assumed to be formed of mutually exclusive cases – each determined by a single strong clause.

terminals An atomic element of a stream that a grammar is processing.

theta environment A theta environment is a set of mutually recursive definitions that make up the definitions of a class or package.

transport protocol A mechanism used to access a resource or to deliver messages between applications.

type annotated expression An expression whose type is explicitly marked by the programmer.

type assignment A type assignment of an expression is a mapping from the expression to a type term.

type constraints a type constraint is a predicate that must be satisfied if the program is to be *type safe*.

type expression A type expression is a term that *denotes* a type. A simple type expression would be something like the symbol `char` – which denotes the type of character expressions. A more complex example would be `list[char]` which denotes lists of characters – i.e., strings.

type inference A process for determining whether a program is type safe or not. A type-unsafe program *might* be able to compute a value, but is likely to not be able to compute a valid value.

type inference Type inference is the process by which a type expression can be *automatically* assigned to an identifier or expression *without* requiring that the type of the identifier be explicitly declared.

type interface An interface is associated with a type that defines the legal operations on values of that type. More specifically, the interface defines the expressions possible on the right hand side of a 'dot' expression.

type lattice a partial ordering with an identified `top` element that is bigger than all others, and a bottom element `void` that is smaller than all other elements.

valof expression An expression whose value is determined as a result of executing an action.

Index