Language Report

Kalkyl

0.1/24.12.07

Reference Compiler “Zuse”

Base Library

Contents

1 Introduction

Kalkyl arose out of the need to make innovations of strongly academic languages such as Haskell accessible for system and industrial application programming through a practical descendant. In order to retain the advantages of the functional approach, above all the referential transparency, the stateful imperative programming is typified and encapsulated from the remaining functions designated as pure. In this way, program effects remain understandable and reflected at the type level.

Functions without any exchange with the outside world, such as user interactions, are considered pure. Consequently, the function results depend solely on the parameters of their interface. This, admittedly quite trivial, requirement increases the traceability and reuse of the programming enormously, but also facilitates the automatic parallelization in many cases; properties that are in demand today more than ever and which Kalkyl intentionally promotes through simplifications and an ergonomic syntax.

Efficient

As a powerful tool, Kalkyl allows developers to write programs at a high abstract level, without sacrificing fast execution speed. This goal is achieved by outsourcing all technical issues into the implementation of Kalkyl, which uses optimized C as a cross-platform intermediate language to generate compilations with native speed and to greatly simplify interoperability with existing software. Since, unlike the model Haskell, there is no garbage collection, mostly unboxed types used and laziness only when explicitly requested, Kalkyl achieves the stated goals much more easily. In addition, imperative language resources are offered directly, especially for manual memory management, which translate into the corresponding C instructions straightforward, so that Kalkyl is equally suitable for machine-level programming.

Type Safe

Despite strict type treatment, Kalkyl offers the highest possible genericity by restricting the data type of a parameter only as much as appears necessary with regard to the operations used. This facilitates functions to be formulated extremely generally in a natural way. Furthermore, Kalkyl has a system to generalize interfaces about certain properties, which can be implemented for any other type. Since such type abstractions are already resolved at compilation time, together with all the necessary checks, there is no negative impact on program execution. Rather, the all-encompassing static type system guarantees the greatest possible faultlessness or comes closest to this ideal state as soon as a program can be compiled.

Contemporary

In view of today's multi-core processors, a modern programming language must provide practicable means for parallel program execution. As a purely functional programming language, Kalkyl is naturally predestined for easy-to-use parallelization of individual sections. In addition, an actor model embedded in the language allows any number of concurrent processes that communicate securely with each other through messages.

Well Thought Out

The focus is also on programming in an accommodating syntax, which combines the comprehensibility of classic Algol languages such as Pascal with the power and elegance of Haskell. This is based on the fact that source texts have to be read more often than written. On top of that, Kalkyl's vocabulary is largely based on internationalisms, which occur in the same meaning in most European languages. Common mathematical notations are also preferred in order to be as language-neutral as possible making it easier for beginners to get started.

Expressive

The extremely regular and totally expression-oriented language structure leads to an unusually high degree of orthogonality: Every construct, including functions, can be understood as an object[1] and linked with other objects to form higher structures. This opens up a wealth of abstractions for the programmer, which conventional language only offers to a limited extent or in an extremely cumbersome syntax.

1 In Kalkyl, the term “object” function primarily as a synonym for value and has no reference to object-oriented imperative languages.

Summary

Logo	
slogan	“Salvation for Programmers”
official website	kalkyl.dev
language structure	declarative and totally expression-oriented
basic paradiga	functional
subparadigms	generic, modular, reactive, concurrent
role models	Ada, Clean, Fortran, Haskell, Idris, Lua, OCaml, Pascal, Python
syntax	Haskell-like with influence from Algol languages and Python
type system	total, static, partially nominal, inferring
file extensions	k • plan
implementation	standard compiler “Zuse”
official repository	repo.kalkyl.dev
translation model	compiling with C as an intermediate language
	Interpreter and interactive environment

Kalkyl is a purely functional programming language, which, in contrast to its predecessor Haskell, evaluates strictly by default and lazy only explicitly at the programmer's request. In addition, Kalkyl offers direct access to imperative means, but encapsulated from the rest of the purely functional programming. Furthermore, Kalkyl is characterized by an extremely high linkability of language constructs. For practical use, Kalkyl interoperates seamlessly with C.

About the Name

The name "Kalkyl" derives from the German word for "calculus" and refers to the first high-level programming language, the "Plankalkül" by Konrad Zuse. Furthermore, the "y" in the name not only represents an international transcription for the German letter "ü", but can be likewise interpreted as a reference to Python. Beside those honoring allusions, emphasis is also placed on the mathematical nature of the language.

The designation "Plan" for the exchange format of the package system is taken as well from "Plankalkül"; but can be alternatively understood as a shortening of "Rechenplan" (literally: computation plan; or derived from the same Old High German word: "reckon plan") introduced by Zuse as a term for "program".

Declarative Role Models

Haskell	• syntax and referential transparency • static type system with type inference • partially applicable higher order functions and operators • generalized algebraic data types and pattern matching • smart constructors (as an idea) • multiparametric type classes with associated types and fundeps • deriving instances • idea behind monads for sequential computation
Clean	• referential transparency through uniqueness types • IO handling • implementation using graph rewriting
Erlang / Elixir	concurrent programming using messages
Lisp, Rebol, Tcl	prefix notation as basic structure
Agda, Idris	value-dependent types and program verifications
F#, OCaml, Opal	suggestions for syntax and module system

Along with the ideas, Kalkyl inherited Haskell's syntax, albeit strongly revised to be more coherent. In addition, there is an effort to make syntax sugars such as Haskell's do notation superfluous, in that Kalkyl also offers classically imperative constructs from the outset, embedded in a purely functional language nonetheless.

At this point, it should be emphasized that Clean's uniqueness types are considered to be a much cleaner, simpler and the superior solution for IO as well as other program effects; especially to avoid the lack of composability and resulting complex stacking of monads.

Imperative Role Models

C	simplicity, 100% control over memory, compilation model
Fortran, Pascal	syntax, pointers
Ada	syntax, loops, types
Occam	syntax, processes and parallelism
Lua	syntax, table as general purpose data structure
Python	syntax
Nim	comment syntax, translation to C
Go	tooling, concurrency
Rust	affine types / memory security, error reports, developer tools

One design goal is that you can do everything in Kalkyl that you can do in C, but under certain rules guaranteeing referential transparency and memory safety. However, these rules must be easy to understand so that they quickly become intuitive when used.

Feature Scope

One of the main goals of Kalkyl is the exclusion of features that do not completely differ from the existing constructs or are not general enough to cover all special cases. In order to systematically avoid such redundancies, Kalkyl is designed on the basis of its own model of programming paradigms:

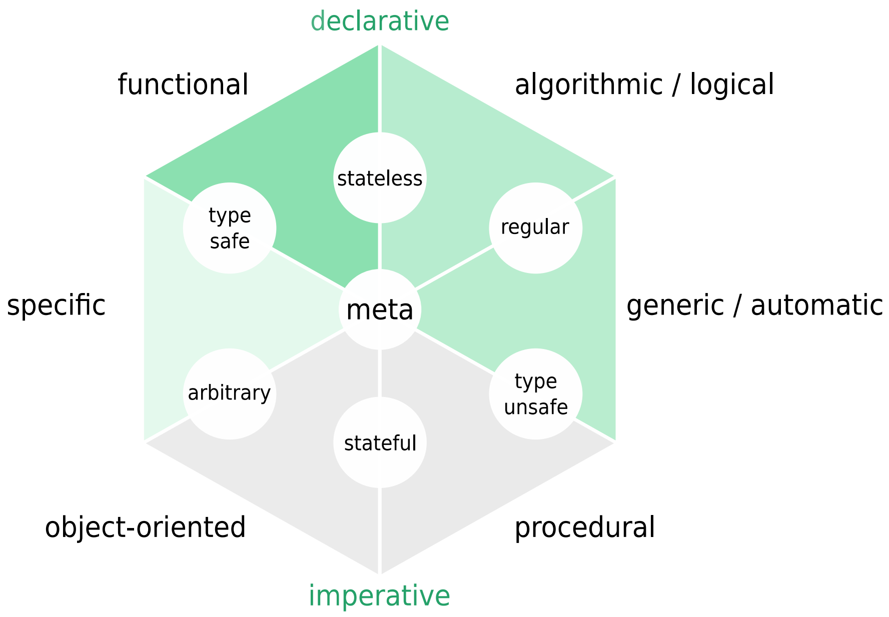

Fig 1: Feature Scope According to a Cube Model of Programming Paradigms; every programming language can be classified using three axes, which stand for opposing approaches: functional / procedural, logical / object-oriented and generic / specific.

A clear assignment to a certain programming paradigm is not always possible, as they overlap numerously while most concepts are just renamed or syntactically handled differently, but ultimately do the same. For this reason, it is also avoided to develop and advertise Kalkyl as a multi-paradigm language. For example, the modular programming is also solved statelessly and therefore just an extension of the functional approach.

Intended Features

- pure, total, partially applicable and variadic higher-order functions
- named and optional arguments
- generalized algebraic data types with definable constructors
- local type variables / existential type statements
- refinable value-dependent types
- compound data with its own namespace
- measuring units
- multiparametric type classes with functional dependencies and associated types
- globally unique and coherent instances of type classes

- several instances of the same type next to each other through explicit naming
- generation of instances
- maintaining referential transparency through uniqueness types
- imperative programming style in an expression-oriented way
- real imperative language constructs and manual memory management
- native support for concurrent and parallel programming
- a real module system (not just namespace), but no separate modular language

Further concepts are not planned, so that Kalkyl is considered completed or fully implemented when version 1.0 is reached. There is no need for additional features as in imperative languages, since Kalkyl is developed with the view that a good programming language does not have to support innumerable programming paradigms in order to be powerful enough for general applicability. Last but not least, the functional nature and uniform syntax should be preserved.

2 Fundamentals

This chapter anticipates all aspects that seem too basic to continue into detail without them. The lexical structure of Kalkyl is dealt with first, followed by other essential topics of the language. All further chapters are based on the rules discussed here.

Structural Order

The development of software using Kalkyl, from project management to the finished program, can be described as a hierarchical order, according to which this specification is organized in reverse:

0. No programming language can ever come close to being useful if it is not easy to set up and obtain libraries from software repositories. The tooling is at least as important as the language itself. Above all, a package manager to share code in a systematic way is considered indispensable today, so that Kalkyl provides one integrated into the reference compiler.
1. Kalkyl programs are developed as so-called **plans** meaning nothing else than project folders of a certain structure with additional information for the package management and deployment.
2. At the highest organizational level, a Kalkyl program is made up of a number of compilation units called components, with non-project units being referred to as program libraries. Each compilation units has a public interface and can used by other components.
3. Each component expose a set of declarations, which specify entities such as values, data types, concepts, but also preinstalled modules. The program to be executed is ultimately created from declarations.
4. At the next lower level are expressions evaluating to bindable values, each with a static type. All expressions of a Kalkyl program are somehow embedded in declarations, mostly to make them callable.
5. The lexical structure predetermines the concrete representation of Kalkyl programs in text files. In particular, the variety of literal representations are mentioned here at that lowest language level.

2.1 Lexical Structure

Kalkyl's lexical structure relies on significant indentation to define code blocks. The language is case-sensitive and enforce capitalization rules, leading to simplifications in many places. The vocabulary represents its own composition; and therefore differs greatly from the main model Haskell.

BNF Notation and Symbols

A modified Backus-Naur form is used to formally describe Kalkyl as far as possible:

```
nonterminal = expression # comment
```

code / terminal symbol	**`code`**	(any colorful text)
Unicode characters	`U+…`	`U+[start, stop, step]`
single value or range of possible characters		
choice	`|`	
choice with highest operator priority	`/`	= (… | …)
subexpression	`(expr)`	
optional expression	`[expr]`	
relative complement	`\`	
smart spacing		(just whitespace)
concatenative, omitted when required		
smart unbreakable spacing	`&`	
mandatory line break	`↲`	
one indent	`→`	
any integer	`%int`	`%int:[min, max]`
mere namespace	`Namespace`	
listing	`N.{a, b, …}`	`= (N.a | N.b | …)`
recursive self-concatenation	$\mathtt{expr}_n$	`= (expr | expr expr | …)`
— until	$\mathtt{expr}_{[1,\ 2]}$	`= (expr | expr expr)`
recursive listing	`{expr}`	`= [expr [, expr […]]]`
any digit	0-9	`= (0 | 1 | …)`
any allowed letter	A-Z | a-z | a-A	`= (A | B | …)`
number of Unicode characters	`:[min, max]`	`:[min, _] :[_, max]`

Kalkyl's BNF was developed with the aim of being more concise but also easier to understand than any EBNF:

- Definitions are separated from each other by indentation; and a concatenation operator is not required to avoid unnecessary syntactic noise, which only distracts from the language to be described.
- The smart spacing can be thought of as a structure of any length to describe gaps of arbitrary sizes:

  ```
  ( ) = ' ' | ' ' ~
  ```

- In the case of an adjacent empty expression (optional), the smart spacing does not apply.
- Valid identifiers may only be composed of Latin, Greek or Cyrillic letters and may contain hyphens to separate word components. In addition, the KBNF differentiates between uppercase and lowercase letters.
- Variables must start with a lowercase letter:

  ```
  a = EXPR
  ```

- In contrast to variables, namespaces always begin with a capital letter:

  ```
  Namespace.var = expr
  ```

  ```
  Namespaces offer the advantage of binding each case separately:
  ```

  ```
  N.a = expr

  N.b = expr

  N.c = expr

  a = N        # = N.a | N.b | N.c

  a = N.{a, b} # = N.a | N.b
  ```

Symbols Used

A number of vague symbols are used to describe the structure of Kalkyl without resorting to concrete examples:

`expr`	any expression
`lit`	any literal
□	any name / identifier
$□_A$	any capitalized name
$□_a$	any uncapitalized name
.□	any name / identifier qualified by one or more module namespaces
○ , △ , □	any operator symbol / operation
a … z / A … Z	exemplary identifier
T	any type
C	any concept
N	any namespace
…	further programming
$\texttt{echo}_{in}$	any input as an exemplary IO action
$\texttt{echo}_{out}$	any output

Specific expressions are named according to their data type, but in uppercase like $\texttt{int}_n$ for any integer.

Comments and Documenting

Line comments begin with a hash #, documenting comments with two ##:

```
## Calculates the area of a rectangle given its width and height.
## - Width of the rectangle in cm.
## - Also given in cm.
## ! Use the generic (overloadable) function #area instead.
rectangle-area : -width Float64 -height Float64 -> Float64
rectangle-area w h = NaN if w < 0 or h < 0 else w * h
```

A documenting comment must be placed immediately after the indentation and before the respective statement to be considered as an associated short explanation for display in IDEs.

Notes on parameters are given in order, with labels optionally mentioned – if available:

```
## …
## -width Width of the rectangle in cm.
## -height Also given in cm.
## …
```

Within documenting comments, names prefixed with a hash are considered redirects to Kalkyl identifiers.

There are no block comments.

Furthermore, comments should not be misused to “comment out” code. Instead, Kalkyl provides the keyword ex for this purpose.

MD Files for In-depth Documentation

In Kalkyl, extensive doc comments on individual functions are generally superfluous due to the strict typing and self-explanatory specifications with labels, so that short explanations of 1 to 3 sentences are completely sufficient in most cases. In addition, mixing extensive documentation with examples and program code is considered bad practice and instead separation by different files is preferred in order to keep the Kalkyl source file clear. For this reason, the convention is hereby stated that a same-named MD file of a component is considered to be its additional documentation, which is intended for much more detailed explanations with example code.

Keywords

Kalkyl inherits most of its keywords from Haskell, but deviates in certain places to appear simpler. Furthermore, abbreviated keywords are avoided, and instead Haskell's declarative vocabulary gets carefully supplemented or modified by borrowings from ALGOL languages, without becoming cumbersome.

All of the words listed are blocked and not available as identifiers:

declarations	`component` / `provides`
	`use` / `under` / `unqualified` / `as`
	`section`
	`subtype` / `type` / `supertype` / `deriving`
	`metric` / `nonmetric`
	`concept` / `given` / `install` / `preinstall`
	`alias`

general additions	`ex`
	`has` / `is`
	`with`
	`forall` / `where`

binding expressions	`let` / `in` / `using`
	`module`

functional controls	`if` / `unless` / `then` / `else`
	`case` / `of`

imperative controls	`do`
	`for` / `while` / `loop`

jump commands	`break` / `continue` / `return`

verbal operators	`not` / `and` / `nand` / `or` / `nor` / `xor` / `xnor`
	`in?` / `is?`

> The keywords are somewhat sorted according to the syntax and their semantic meaning. In fact, the vast majority of them function as delimiters.

Jump commands are actually just ordinary functions that always return a generic action. Due to their fundamental role, the identifiers of jump commands are still considered keywords and should be highlighted as such in text editors. The same applies to all verbal operators.

> The keyword `as` is used not only in declarations for renamings, but also in expressions for explicit type conversions.

Memory management functions such as `malloc`, `init`, `remalloc`, `free`, `new` and `dispose` should not be marked as keywords, but still be distinguished from ordinary functions because of their importance.

Punctuation and Operators

Character sequences which, like keywords, have syntactic significance, but do not consist of letters, are referred to as punctuation:

Declarer	**indentation**		marking blocks
	line break		marking the end of expressions or subexpressions
	`:`	*	specifier / specifying expressions and declared identifiers
	`=`		definer / defining identifiers
	`~`	*	same name
Separators	`,`		listing of imported or exported identifiers
			listing of constraints
			listing of functional dependencies
			separating elements in set literal
		*	combining individual expressions into a tuple / tuple builder
	`;`		separating declarations on the same line
		*	linking expressions into a sequence
Grouping	`( )`		grouping of expressions / tuples
	`[ ]`		accessing an element by index
			getting a slice
			defining an interval
	`{ }`		enclosing elements of a data structure
Qualifiers	`$`		value caller
	`&`		method caller
	`.`		qualifier / qualifying through namespaces
	`::`		typing qualifier / qualifying through types (return type hint)
	`^`		dereferencer (distinguished from pointer by postfix position)
Pattern Syntax	`->`		separating parameters from the return
			separating patterns from their associated expressions
	`..`		omission
	`\`		introducing function expression
	`_`		placeholder
	`|`		separating cases and patterns on the same line
			separating data constructor declarations on the same line
			separating functional dependencies from the parameters
Marks	`'`		enclosing number literals with spaces
			enclosing string literal
	`"`		enclosing string literal

Dashes of labels and slashes of paths are distinguished from binary operators with the same symbol by missing spaces.

The symbols marked with an asterisk can also be viewed as restricted prefix or infix operators, as they are not syntactically restricted, but also appear independently in expressions.

Difference Between Punctuation and Operators

Some punctuations can also be viewed as operators, since they not only appear in a very specific context together with other punctuation marks and thereby form syntax, but also occur quite generally, like the specifier and method caller. On the other hand, operators are, in a sense, syntax symbols too, as they are predetermined as part of the language definition and hence not user-definable like names. For this reason, all operators could be listed here likewise. In order to somehow distinguish between mere syntax and actual operator characters from a practical point of view, the following criteria are established:

- Operators always appear as independent and not only in the context of a specific syntax; hence any valid expression of an appropriate type can be an operand:

  ```
  expr + expr
  ```

- Furthermore, operators behave like functions, just with a more symbolic name in infix (or postfix) notation, but are equally partially applicable:

  ```
  (x +) = add x
  ```

This fundamental behavior does not exclude the possibility that an operator can also be syntactically interwoven with punctuation or other operators:

```
ptr^.field := expr
```

> In this example, the dereference `^` operator appears in the middle of the identifier as the only exception besides the separating period `.` between the namespace and the name. However, the dereference operator can also appear in ordinary expressions, like any other operator.

In order to be a full-fledged operator, the operands would have to be expressions to the left and right of the period, which would only be possible if names themselves were first class values. It is conceivable to use another symbol like `$` to express that the specified name itself is just a variable for the actual name:

```
name x = Namespace.$x
```

However, there is currently no need for such a possibility, which is why the qualifier `.` is currently viewed as a mere punctuation mark as part of the naming syntax.

The address operator `@` represents a special case because it can only be used with certain types of identifiers; literals and other expressions are not permitted as operands. Therefore, the address operator, and for the same reasons the dereferencer, could be viewed as special syntax like lambda expressions and not as operators in the actual sense. Because of their operational appearance and practical importance, they are still considered as operators, although not as first-class.

In summary, this consideration shows that operators represent by no means just symbolic names for functions, but are more syntactically embedded in the language, which is why in Kalkyl operators are not user-definable and instead viewed as a special kind of punctuation.

Positional Differentiation of Operators

To mitigate the limitations of the ASCII character set, the position of operands is also significant in identifying the actual operator. Therefore, a distinction is made not only in terms of the symbol, but also whether the operator is applied prefix, infix or postfix:

```
75 * 2     # multiplication
x : *Int   # uniqueness type operator
```

Pre- and postfix operators can be accumulated, depending on the type and its implementation:

```
x : ***Int # but has no additional effect on types that are already unique
```

However, this potential is only exploited very sparingly in order not to unnecessarily complicate the language, so that Kalkyl has just a few pre- / postfix operators overall.

There is also a rule that spaces can be omitted between identifiers or bracketed expressions and pre- / postfix operators, whereas spacing is mandatory for infix operators, with the exception of a few idioms.

Identifiers

An Identifier refers to a bound expression. Although the full Unicode character set is available, identifiers may only contain letters of the Latin, Greek, or Cyrillic alphabet, since Kalkyl uses mandatory capitalization rules to distinguish constructors from values including ordinary functions. Consequently, types and their data constructors must always start with an uppercase letter; whereas values, parameters, and type variables have to begin with a small letter. Affixes and context rules provide further independent name domains:

Group	Identification – by Context	Examples
value, parameter	lowercase at value level	`echo`
predicate	lowercase + suffix `?`	`odd? 91`
data method	operator `&` + lowercase	`"Hey Bro!" &length`
tuple field	value + dot notation + lowercase	`person.name`
label	prefix `-` + lowercase	`map -f (\x -> x + x)`
operator	keyword or punctuation	`45 / 8`
type variable	lowercase at type level	`type T x …`
constructor	capitalized	`True`, `Math.add`
namespace	capitalized + dot notation	`Math.add`
compilation	capitalized at file level	`component Example`
repository	capitalized	[only internally used]
measuring unit	postfix notation (in any case)	`17.8 cm / x cm`

After the first letter, any other letters, but also numbers, may follow, between which a hyphen can optionally separate components of word compositions for better readability.

> The mandatory capitalization rules are intended to increase readability and free the programmer from convention issues. Furthermore, this regulation enables elegant syntactic simplifications, especially with pattern matching.

The term “value” includes variables, functions, parameters as well as type variables, just everything that is not a namespace, constructor or label.

Identifiers with the suffix `?` are only allowed to bind expressions evaluating to a boolean value which may also be a function; hence the exclamation mark is to be understood as a letter that can only appear at the end of a lowercase word.

> For the sake of simplicity, identifiers are also referred to as names in the specification, whereas the compiler distinguishes between internal identifiers and the context-dependent names given by the programmer.

Since records must be assigned to names in lower case, there is no risk of confusion with namespaces.

A distinction is made between closed and open namespaces: The former are introduced with **`use`** for the respective libraries, unless they are explicitly imported under an open namespace with **`under`**.

Units of measurement are special entities for they represent constructors in postfix notation, which can be both lowercase and uppercase, hence clashing with values and other constructors of the same name.

> Since metric prefixes and SI units only consist of individual letters and do not form meaningful words, name collisions should be easy to avoid. The advantage of being able to write quantified numbers as usual [**`width = 36 cm`**] outweighs emergency solutions [**`36<cm>`** or **`36[cm]`**].

No Direct Overloading of Names

One of Kalkyl's ideals is that, unlike in many other languages, functions cannot be overloaded arbitrarily, so that the IDE's intelligent code completion does not show countless entries under the same name, but instead only one generic function with some constraints. To realize this ideal, there are type classes, as first appeared in Haskell and referred to as concepts in Kalkyl, which systematically provide ad hoc polymorphism, but without the disadvantage mentioned above. In addition, Kalkyl offers optional parameters and anonymous sum types, which together with concepts eliminate the need for direct overloading 99% of the time. For these reasons, the name of a value may only be assigned exactly once in a respective namespace.

Identifiers with Special Meaning

Some claimed identifiers have special meanings:

repositories	**`GitHub`**, **`Local`**, **`Public`**
values	**`program`**, **`undefined`**

The function **`program`** is assumed to be the entry point of a compilation unit when executed.

With **`undefined`**, a value can be implemented, which, however, leads to an error as soon as it gets evaluated.

Measuring Units

Identifiers for units can consist of permitted letters and the following characters:

`%` **`°`**

Allowed Unicode Characters for Identifiers

```
Letter.Capital.Latin.basic
    = U+[41, 5A] # 'A' … 'Z'

Letter.Small.Latin.basic
    = U+[61, 7A] # 'a' … 'z'

Letter.Capital.Latin.supplement
    = U+[C0, D6] | U+[D8, DE]

Letter.Small.Latin.supplement
    = U+[DF, F6] | U+[F8, FF]

Letter.Capital.Latin.Extended-A
    = U+[100, 136, 2] | U+[139, 147, 2] | U+[14A, 17D, 2]

Letter.Small.Latin.Extended-A
    = U+[101, 137, 2] | U+138 | U+[13A, 148, 2]
    | U+[14B, 17E, 2] | U+17F

Letter.Capital.Latin.Additional
    = U+[1E00, 1E94, 2] | U+[1E9E, 1EF8, 2]

Letter.Small.Latin.Additional
    = U+[1E01, 1E93, 2] | U+[1E95, 1E9D] | U+[1E9F, 1EF9, 2]

Letter.Capital.Greek.basic
    = U+[391, 3A1] | U+[3A3, 3A9]

Letter.Small.Greek.basic
    = U+[3B1, 3C9]

Letter.Capital.Cyrillic.basic
    = U+[400, 42F] # 'А' … 'Я'

Letter.Small.Cyrillic.basic
    = U+[430, 44F] # 'а' … 'я'

Letter.Capital.Cyrillic.extended
    = U+[48A, 4F8, 2]

Letter.Small.Cyrillic.extended
    = U+[48B, 4F9, 2]
```

Literals

Literals are values written directly into the source text and thus forming matchable patterns. There are a whole range of different literals, for example absolute and relative paths like in shell languages:

empty expression	`()`

whole number	`926729`
	`0b10010`
	`0xFFFFAA`

fractional number	`2.71828`
	`1.1(36)`

special numerals	`NaN` / `-Inf` / `Inf`
negative number	`-123`
positive number	`+123`

binary data	`b'10011011001010010011110011 10'`
	`x'9B293CE'`

string
+ escape sequences / formats
	`"Hallo, Welt!"`
	`"Moin,\n$first $last!"`

raw string	`'C:\Windows\System32\system.ini'`
tuple	`("Leipzig", 1234)`
named tuple	`(-name "Olaf", -age 26)`
plain array	`{0, 2834, 43, 59, 732, 29}`
associative array	`{-id "header", -class "new"}`
variant array	`{76, “Hallo Welt!”, True, None}`
interval	`[a, b]`

The empty expression can act as either a value or type depending on the context

Beside the exponential notation for particularly large or small numbers, periods can also be expressed using parentheses.

Text, also known as character string in other languages, is separated from the rest of the programming through single or double quotes. The only difference is that for text literals formed with single quotation marks, double quotation marks do not need to be marked. The same principle applies to the reverse case:

```
"His name is "Olaf"."
```

If the string is marked as formatted using a preceding "f", curly brackets are interpreted as an area into which printable values can be inserted directly:

```
let name = "Olaf" in f"His name is {name}."
```

Intervals are just special set builders:

```
[a, b] ≡ {x | x >= a, x =< b}
```

Constructors

Constructors are calls that behave like literals forming matchable patterns (→ pseudoliterals). Kalkyl differentiates between constructors that return either a type or a value. In contrast to identifiers of ordinary name bindings, constructors must begin with an uppercase letter. This allows data constructors to be distinguished from variables as placeholders during pattern matching:

```
type Either t has Left t | Right t

right? : Either a -> Bool
right? Left  x = False             # or just: Left _
right? Right x = True
```

For the sake of simplicity, data constructors get mostly abbreviated as “data”. A Nullary or fully applied type constructor, on the other hand, correspond to the term “type”.

2.2 Program Structure

In Kalkyl, a distinction is made between declarations and expressions. Declarations denote constructs at the top file level, which bind expressions. On the other hand, expressions include literals, calls, structures like controls or combinations of these with one another. Bound values formed by evaluated expressions can in turn be called by other declarations:

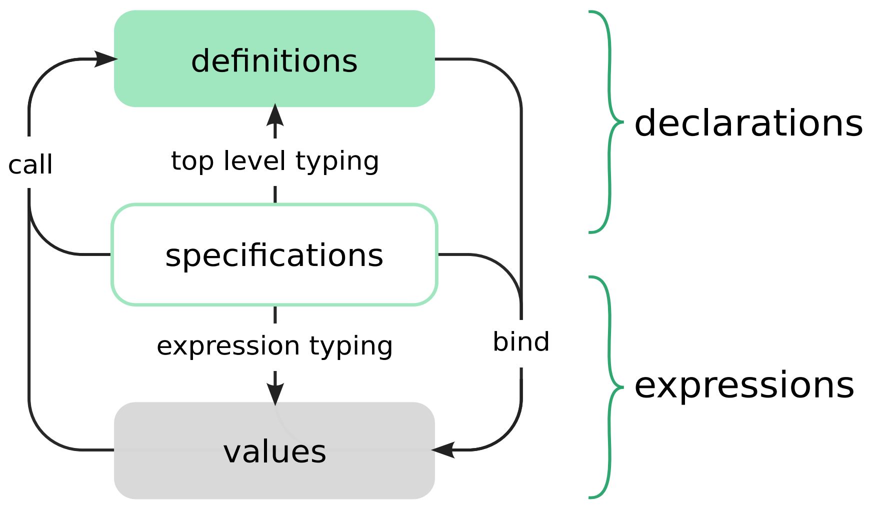

Fig 2: Interplay Between Basic Language Structures

Specifications provide information about the type of a bound value, which can either be made parallel to definitions at the top level, or within expressions. As a general typing character, Kalkyl uses a single colon `:`.

A **value** is defined as a representative belonging to a certain data type. Every evaluation of an expression leads to a concrete and bindable value. For the sake of simplicity, bound values, including functions, are also simply referred to as values.

The term **name binding** means assignment of values to unique names. This includes not only declarations at the top file level, but also let expressions, passing of arguments and assignments of mutable variable as part of actions (sequences with side effects). In local bindings, names may overshadow global identifiers.

Definitions, as a subset of name bindings, make values known. **Specifications** are the related signatures, which can either be stated by the programmer or otherwise get automatically determined by the compiler based on the bound value (type inference).

Is everything really an expression?

In fact, everything in Kalkyl is typed and hence bindable. Consequently, one could say that there are no statements, only expressions. That's true, since even program effects are always attached to certain bindable structures to make them comprehensible. As a result, the finished program represents a well typed structure getting evaluated at runtime for some in- and output. In order to make this possible, even every control structure need to always return a concrete value, similar to a function.

Starting from the top level of the file, all encoded expressions must be bound somehow. In this sense, declarations instruct the compiler to introduce new identifiers of certain kinds, where some of them have a special meaning, such as `program` as the entry point to program execution.

Nevertheless, declarations do not represent instructions as in imperative languages, but rather are to be understood as structures that are not bindable themselves and thus only appear like literals in the source code, whose concrete data types remain unknown.

Declarations as Special Expressions

An idea in Kalkyl is that any kind of construct exhibits a type, say `Declaration Type X` as the data type of a template bound to an identifier `x` which gets expanded into a declaration just by calling `x` somewhere at the top level of a source file to introduce a new type `X`. However, there is currently no affort or any need to enable such in Kalkyl, so the border between declarations and expressions remains closed, at least for now.

Language Levels

Each construct in Kalkyl consists of different parts, mostly specific blocks or clauses in which certain types of identifiers predominate and only a subset of all language features is available:

Level	Occurrence	Subjects
Value	value definition (lowercase)	literals, value constructors
Type	type specification, alias definition (capitalized)	type constructors
Constraint	where-clause, module head	type constructors
Matching	definitional pattern matching, of-construct	constructors
Module	installations	modules, module templates
Indexing	within square brackets after indexable values	indexes, slices
Unit	after numbers, if needed within square brackets	units with SI prefixes

Value Level

Value constructors are preferred over type constructors with the same name. Nevertheless, any type constructor can still be addressed by explicitly specifying, if necessary:

```
Type::[<namespace>.]ₙ<type constructor name>
```

In contrast to the other levels, all features are available.

Type Level

Type constructors are preferred over value constructors.

All lowercase identifiers get interpreted as type variables. Consequently, values can only be accessed with a special character, for which `$` is used. Moreover, such called values must be already known at compile time.

> A distinction between values and types is nevertheless made, but there are no longer impenetrable boundaries between the value and type levels. In fact, all levels are just subsets of the value level with specific preferences.

Matching Level

Lowercase identifiers are interpreted as declarations of parameters standing for any values of the known type. If a name is unneeded and just any value is to be matched, an underscore `_` is sufficient. Beside that, only literal expressions are permitted.

Constraint Level

Only constructors introduced with `concept` may be listed and which must also be fully applied.

Module Level

After `preinstall` and `install`, module calls but also module implementation are permitted; in using-clauses, however, only module calls.

Contextual Meanings

Only a few characters are directly available on an average keyboard, more precisely the ASCII character set as the greatest common denominator internationally. For this reason, no programming language can avoid giving symbols different meanings depending on the context. Kalkyl faces the same problem of getting the balancing act between easy writing and comprehensibility. In order to keep the resulting complexity as low as possible, these ambiguous use cases and their preference are precisely defined:

`=`

Meaning	**Preferred …**
definition	at top level, between `let` / `in`, after has and `with`
equality operator	inside expressions

`->`

Meaning	**Preferred …**
mapping	as function expression and after `case … of`
type punctuation	inside specifications

`;`

separator between top declarations	before declarations on the same line
separator between name bindings	between `let` / `in` and after `with`

`|`

separator between declarations	inside data blocks (with-clause of data declarations)
separator of different cases	between `of` / `->` and between `then` and `else`

`as`

renaming of imports	after `import`
coercion operator	only in value expressions

3 Expressions

This chapter takes a close look at all constructs that can be bound to names, presupposing that every bound expression has a data type and sooner or later evaluates to a concrete value.

Classification

Expressions are formed by literals, calls, binding expressions and control structures.

Literals

- numerals
  - **code points**
  - **whole numbers**
  - **fractional numbers**
- textuals
  - **strings**
  - **buffers**
  - **paths**
- data structures
  - unexpandable

    = **tuples**
    - unnamed
    - named
  - expandable

    = **arrays**

Calls

- matchable

  = **constructors**
- unmatchable
  - **modules**
  - **values**
    - effectless
    - with IO

Binding Expressions

- **let expressions**
- **modules**
- **type quantifiers**
- **functions**

Control Structures

- **conditionals**
- **case expressions**
- **loops and jumps**
- **sequences**

Literals and matchable calls are the only expressions that can appear in patterns.

Since case expressions also introduce new names, these could be interpreted as binding expressions as well. However, this is only a secondary feature and not there main purpose, which is matching in the sense of control flow.

Unification with 1-Tuples

In mathematics, x and (x) are two different things; in programming, however, this leads to misunderstandings because expressions are usually grouped using round brackets. One solution would be to introduce extra syntax for single-element tuples `(x,)`, as in Python; but this doesn't seem particularly intuitive. Another solution would be to use a different type of parentheses for tuples. However, square brackets are reserved typical for lists or indexes and curly brackets for blocks or compound objects in most languages.

Ultimately, these circumstances raise the question of whether it makes sense at all to fundamentally differentiate between a value and a singleton as in mathematics, especially if tuples do not represent expandable structures, but only a means of grouping several values by a fixed order. For these reasons, Kalkyl treats single values and 1-tuples as the same thing:

```
"Hello" = ("Hello") = ("Hello").1 = (("Hello").1).1 = …
```

This merging has the advantage that, on the one hand, no additional syntax is required, and on the other hand, tuples can be used as usual.

Consequently, it doesn't matter how often a tuple consisting of only one element is nested into another 1-tuple:

```
"Hello" = ("Hello") = (("Hello")) = ((("Hello")))
```

Empty Tuple as a Unit Type

If this unification is continued consistently, an empty tuple `()` is nothing other than the only possible value of the tuple type `()`, which is thus also a unit type:

```
("Hello", 12) : (Text, Int)
("Hello")     : (Text)
()            : ()
```

3.1 Type Expressions

Data types are values of the type `Type`; or in other words: type expressions evaluate to values that can typify other values.

In addition to type constructors and type variables, which visually differ from each other in capitalization, the type level also has a few intrinsic operators and expressions that significantly shape the appearance of specifications:

Type Level Entity	Signature or Syntax
kind	Type : Type
constructor	$\square_A$: [Type ->]$_n$ Type
variable	$\square_a$: expr
value call	\$ $\square_a$
labeled type	-$\square_a$ type
tuple type	_ (,) _ : Type Type -> Type
anonymous sum type	_ (+) _ : Type Type -> Type
default value	_ (~) _ : {t : Type} t -> Type
optional type	(?) _ : Type -> Type
~~reactive type~~	~~_ (!) _ : Type -> Type~~
pointer type	(^) _ : Type -> Type
uniqueness type	(*) _ : Type -> Type

refinement type	{expr^{matchable} [, expr^{matchable}] [: type]}
	{$\square_a$ [, $\square_a$, …][: type] | predicate [, predicate]}

The underscores indicate the position of operands.

Types of types are also referred to as “kinds”, but for the sake of simplicity this is not reflected in the type system: `Type : Type`

Type Operators

The corresponding type operator `+` behaves commutatively, and adding the same type multiple times has no effect:

```
A + B = B + A = A + A + B = …
```

Anonymous sum types make it possible to unite existing types directly with each other, instead of defining a new ADT that has a variant for each type. This way not only eliminates the need for extra types, keeping interfaces simpler, but also avoids additional pattern matching.

The tilde operator `~` associates a type with one of its values, which is used by the compiler as the default for the respective parameter. The resulting type of this operation behaves like a synonym, so there arise no incompatibilities:

```
A ~ a = A
```

If a default value already exists, it will be overshadowed – or replaced internally – by the last application:

```
A ~ a1 ~ a2 = A ~ a2
```

The operator `?` adds `None` to the value range of a type, and at the same time sets `None` as its default value.

Pointers are unique by nature, so applying `*` has no effect:

```
*^T = ^T = ^*T
```

Curried Function

In Kalkyl, all functions are curried, including type constructors. Accordingly, after applying the first parameter, a new type function with the remaining parameters is returned:

```
Either          # : Type Type -> Type
Either Int      # : Type -> Type
Either Int Bool # : Type
```

However, the order of application can be changed by using labels.

Labels

Both the parameters of constructors and value level functions can be labeled:

```
IntRange : -min Int -max Int -> Type
```

It has to be taken into account that either all parameters are labeled or none at all

When using functions with labels, the first parameter does not necessarily have to be supplied with an argument before the others, which simplifies partial applications:

```
IntRange -max 100 # : -min Int -> Type
```

The labeling offers another advantage, as parameters can be written one below the other more clearly, which is particularly advantageous for tooltips / IntelliSense in IDEs:

```
product-item :
    -id Int
    -name Text
    -price Float
    -quantity Int ~ 10
    -> -id Int, -name Text, -price Float, -quantity Int
```

> Brackets around resulting tuples are omittable too since the function operator `->` has less precedence than commas.

It should also be mentioned that, in contrast to Haskell, only one mapping arrow is written, which separates the parameters from the return type.

In addition, non-atomic arguments between labels do not require parentheses unless they themselves call functions with labels:

```
item = product-item -id 2297 -name "Mini PC" -price 399.99 – discount
```

Value Promotion

There is a mechanism for picking up arguments by giving them names:

```
f : {x : Int} -> {x + 1}
```

Such argument names make passed values directly accessible at type level, which allows precise relationships between the individual parameters and the result type to be expressed.

But already bound values are likewise available at type level:

```
max = 100

i : Nat $max # Nat : Literal.Natural -> Type
i = 80
```

To call values without being confused with type variables, they need to be prefixed using a dollar sign `$`. In addition, these called values must already be known at compile time, which means that the bound expressions have to be fully evaluable without any runtime or interaction with the “world”.

Indifferently Curried Functions

All parameters, whether labeled or unlabeled, together implicitly form one tuple, so there is no difference in application between the previous example in the section on labels and the signature below:

```
product-item : (-id Int, -name Text, -price Float, -quantity Int ~ 10) ->
    (-id Int, -name Text, -price Float, -quantity Int)
```

This further harmonization means that the compiler first checks whether a passed tuple is applicable to all parameters:

```
product-item (-id 2297, -name "Mini PC", -price 399.99)
```

Or even without labels, but then the order must be strictly adhered to:

```
product-item (2297, "Mini PC", 399.99)
```

If, on the other hand, a tuple is intended for the first parameter and there is the extremely rare case that said tuple is also applicable to all parameters combined, a partial application of the function can nevertheless be expressed using an ellipsis `..`:

```
cmp                # : ((Int, Dec) + Int) Dec -> Bool

cmp (76, 76.0) .. # : Dec -> Bool
```

This syntax is also useful when optional parameters should not be applied automatically.

Tuples automatically break down into individual arguments to serve curried functions; the ellipsis prevents this by indicating that the remaining parameters should still be applicable.

Parameter Declaration and Call Styles

Function types can be specified in two completely equivalent styles:

	Curried Style	**Tuple Style**
parameters	f : x_1 x_2 … -> y_1 y_2 …	f : (x_1, x_2, …) -> (y_1, y_2, …)
— with labels	f : -l_1 x_1 -l_2 x_2 … -> y_1	f : (-l_1 x_1, -l_2 x_2, …) -> y
calling	f arg_1 arg_2 …	f (arg_1, arg_2, …)
— with labels	f -l_1 arg_1 -l_2 arg_2 …	f (-l_1 arg_1, -l_2 arg_2, …)

Depending on the circumstances, it may be more advantageous to specify the parameters in curried or tuple form. However, this choice in no way limits the possibility of applying a function either by individual arguments in curried fashion or by a single tuple.

For example, it is possible to apply only one parameter, resulting in a new function with the remaining parameters, which are subsequently served by a single tuple:

```
# f : A B C -> D
f arg^A (arg^B, Arg^C)
```

Rules

To avoid inconsistency, the following rules must be adhered to:

- There can only be one function arrow in a type, unless a parameter expects a function expression. Hence, if a new function is returned, its parameters must follow the others: `f : a -> (x -> y)` must be written as `f : a x -> y`
- Curried parameters of tuple types are considered nesting.
- Either all parameters are labeled or not at all, referring to both the curried and tuple variants.
- For labeled parameters and arguments, the brackets around the type expression can be omitted.

With a labeled value the following argument also has a label, otherwise brackets are still necessary around the labeled expression, which then represents a named tuple with only one element:

`f -a expr -b expr` = `f (-a expr) (expr)`

However, if the label matches a parameter that is somewhere else, the argument may be applied to it, thereby bypassing the order of the parameters specified in the function type.

Basic Types and Kinds

In Kalkyl, a distinction is made between primitive types, whose values are stored directly in memory, and abstract types, which are formed by references – realized as pointers – to more complex structures of different implementations:

	Primitives	**Abstracted Variant**
boolean	`Bool`	—
integer	`Int`, `Int1`, `Int2`, `Int4`, `Int8`,	`Int min max`
	`Nat`, `Nat1`, `Nat2`, `Nat4` ⊃ `UTF8Code`, `Nat8`	`Nat max`
floating-point	`Float2`, `Float4`, `Float8`, `Float10`	`Float prec`
decimal	—	`Dec prec scale`
textual	`String`, `Path`	`Buffer String`, `Rope`
contiguous	tuple, `t[length]`	`Map String t`, `Array t`

The term “primitive types” is used synonymously for “basic types”.

Since integer literals are always valid values of type `Int`, the respective type and value constructors can expect themselves as arguments and still terminate:

```
Int : -min Int ~ -Inf -max Int ~ +Inf -> Type

Int::Int :
    -val {i : Int | i >= min and i =< max}
    -min Int ~ -Inf
    -max Int ~ +Inf
    -> Int min max

i = Int 3746
```

Using `Int` and requirements, the remaining byte-sized integers can be defined:

```
Int4 : Type

Int4 : {i : Int | i >= -1 * i**31 and i =< i**31 - 1} -> Int4
```

`UTF8Code` is an unsigned 32-bit integer type capable to hold any Unicode code position.

The float subtypes are just synonyms for the individual variants of the general float type, such as `Float Single` or `Float Double`.

`Dec` describes an exact decimal number, where `prec` states the total sum of digits and `scale` the number of decimal places.

The additional unsigned integer type `Size` is architecture dependent and can store the size of any objects. `Index` is the signed variant of it and primarily intended for indexes in Kalkyl, as these can also be negative in order to start from the last element. Since virtually no computer will have more memory than half of `Size` in the foreseeable future, the halved value range should pose absolutely no problem in practice.

Kinds

The types of types and other type level entities are referred to as “kinds” which cannot be defined, but are simply given by the language itself:

- `Type`
    - `Constraint`
    - `Reference`
    - `Resource`
    - `Value`
- `Dimension`

While `Type` stands for all types of objects, `Constraint` is the kind that restricts type variables in where-clauses and specifies module or component interfaces. Dimensions, on the other hand, describe the compatibility between units of measurement.

References are basically values representing addresses to the actual data, which get automatically dereferenced. The chapter on substructural types and pointers provides detailed information on this.

Resources are objects that contain IO values.

Values refer to all objects that are neither references nor pointers, but are present directly in place.

Predefined Data Structures

The first distinction is made between tuples that cannot be enlarged and potentially dynamic data structures such as vectors and maps:

Structure	Type	Literal Construction	Elements are…
lazy value	`Lazy t`	—	
tuple	`(A, B, …)`	(expr$_1$, expr$_2$, …)	only ordered
named tuple	`(-a A, -b B, …)`	(-a expr$_1$, -b expr$_2$, …)	only ordered and named
array	`Array t`	{expr$_1$, expr$_2$, …}	indexed
list	`List t`	{expr$_1$, expr$_2$, …}	indexed
set	`Set t`	{expr$_a$, expr$_b$, …}	hashed
multiset	`Multiset t`	{expr$_a$, expr$_b$, …}	hashed and counted
map	`Map key val`	{expr$_{key}$ => expr$_{val}$, …}	indexed by hashed keys

Since the elements of tuples may have different types, they cannot be accessed through an iterable index like arrays, but only using dot notation:

```
t = (123, True, "Hallo")
n = t.1                    # n →  123
```

Therefore, tuples – whether named or unnamed – represent in fact just a means of grouping multiple values in a convenient way, and are the only native data structure, since they get stored and processed in place, rather than using pointers.

Construction From Literals

A listing in curly brackets is always interpreted as an array literal transferable as possible input to constructors of predefined data structures. With appropriate implementations, an array literal can also be coerced into a specific collection, assuming the required collection type is already known by the context:

```
s = {2, 8, 8, 14} : Set
```

When constructing a set, duplicate elements in an array literal get simply ignored; whereas the multiset counts the frequency of elements.

In this way, maps are also creatable from array literals, whereby the overloaded operator `=>` forms pairs of values as tuples and, in contrast to the comma, has a higher priority, so that parentheses can be omitted:

```
m = Map {"foo" => 1, "bar" => 2, "baz" = 3} # same as: {("foo", 1), …}
```

Maps as Functions with Preconditions

Maps posses the special properties that they automatically become functions with preconditions:

```
f = {0 => "nein", 1 => "ja"} .. # f : {0, 1} -> Text
y = f 1                          # → "ja"
```

If no argument is already specified, the ellipsis `..` can be used to indicate that the map is to be treated as a function.

Size Limiting

Normally, the length is not part of the type, as unique collections can potentially shrink or grow. Nevertheless, the type constructor `Fix` provides the ability to fix collection types to a specific size, which opens up potential optimizations.

> The exact specification of the types is detailed in the respective chapter on the standard library.

In addition, less complex data structures such as arrays offer the possibility of being embedded directly in place; for example, as an element of a tuple, instead of being allocated separately. The type constructor Embed exists for this purpose; in summary:

`Fix : Type Args MemorySize -> Type`	limited to certain capacity
`Embed : Type Args MemorySize -> Type`	fixed and integrated into current layout

Implementation Note

Sets as simplified hashmaps, for example, would also be potentially embeddable since they can be implemented as single memory blocks of elements arranged in a row. Arrayas are therefore inherently easier to convert into optimized C equivalent code:

Kalkyl	**C**

```
type Student max is
    -name Embed String 50
    -grades Embed Array Float4 max
```
	```
	struct Student {
	    char name[50];
	    float grades[MAX_GRADES];
	    int grade_count;
	};
	```

The Kalkyl variant can be more flexible thanks to type parameters and requires fewer fields, since the array information such as capacity and actual length are already part of the array object (as a header before the actual sequence of elements begins).

> Data structures such as strings and arrays are actually processed using abstracted pointers, which is why normally only space equal to `Size` is reserved.

Refinements

Formally speaking, data types in Kalkyl are nothing more than sets of values. For example, in addition to being just a set – or array – at value level,

```
{"foo", "bar", "baz"}
```

would also be a valid type with three possible character strings as the only permissible values. However, since literals in Kalkyl are generic, such a listing requires specifying the wanted type of literals:

```
{1, 3, 4, 7 : Int}
```

For numerous values or theoretically infinite ranges where only a few variants need to be excluded, the set-builder notation is the method of choice:

```
{i : Int | i >= 0, i =< 100}
```

The individual statements after the vertical bar must evaluate to Boolean values separated from each other by a comma, which acts as a weakly binding logical AND.

Besides logical-, comparison- and arithmetic operators, predicates are also permitted, meaning functions that return a Boolean value after complete application:

```
perfect-square? x:Int = let s = floor sqrt (x as Float) in s * s = x

fibonacci? n = let sq = n * n in
   perfect-square? (5 * sq + 4) or perfect-square? (5 * sq - 4)

Fib = {n : Int | fibonacci? n}

x : Fib
x = 35  # → Type error since 35 is not a Fibonacci number!
```

> Capitalized names are aliases for the type level and may only bind values of type `Type` or `Constraint`, or functions with these return types. This is discussed in more detail in the chapter on declarations.

Since functions that return Boolean values may only be bound to names with a question mark attached, they can be called as type-level predicates without being confused with local variables.

Only Total Functions

Refinements can express preconditions when used as types of parameters, or postconditions when they restrict the return. Consequently, there are no partial functions in Kalkyl, since appropriate preconditions define exactly the range of values for which a function works. As a result, Kalkyl does not need and does not have the ability to generate errors directly from pure code, not least because all relevant external states are represented by their own unique values to model “the world” at code level. Moreover, the compiler can automatically generate useful error messages based on refinements.

3.2 Binding Expressions

Let expressions, type quantifications, parameters of lambda expressions, implementations of concepts, as well as named placeholders in case expressions have in common that they all are somehow entangled with expressions, while also introducing locally scoped names for values which overshadow global identifiers of the same name. However, since case expressions primarily function as control structures, they are not considered here.

Let Expressions

Let expressions introduce a lexically scoped, mutually recursive list of declarations limited to the expression that follows:

```
let decl in expr
```

Let expressions can only be used at value level.

Type Quantifications

A subsequent forall-clause explicitly introduces type variables:

```
(>>) : m a -> m b -> m b forall m, a, b where Action m
```

Optionally, the exact type of each variable is specifiable:

```
(>>) : m a -> m b -> m b forall m: Type -> Type, a: Type, b: Type where …
```

However, specifying the forall-clause manually is cumbersome and usually unnecessary because the compiler can derive it. If the type variables are nevertheless explicitly stated, their scope also extends to all local name bindings, both at the type and value level.

In addition, type variables are constrainable by a subsequent where-clause.

In a figurative sense, universal quantifications can be understood as indirect parameters that are automatically assigned a type depending on the usage by the caller.

Function Expressions

Lambdas, less theoretically also called function expressions, form parameterized values of a function type:

```
\a b -> a + b
```

with the corresponding type:

```
\a b -> c where Adding a b
```

The character string `->` is neither an operator at value nor at type level, but simply syntax just like the backslash `\` announcing parameters of a function expression.

> The backslash `\` has no further meaning in Kalkyl to facilitate parsing of the language.

Naturally Generic

Kalkyl only restricts the value range of functions as much as appears necessary to ensure that all applied operations work correctly. Consequently, the identical mapping would not impose any constraints and thus be absolutely generic:

```
id x = x # id : t -> t
```

Currying and Offset Rule

Functions in Kalkyl are curried; accordingly, this expression

```
\a -> \b -> a + b
```

is completely equivalent to the previous example, likewise the following expressions:

```
(\a b -> a + b) 1 2 ≡ (\a b -> a + b)(1) 2 ≡ (\a b -> a + b)(1)(2)
```

Parentheses are superfluous with atomic arguments as long as there is at least one space between them and the function value, or a line break with exactly one indentation depth:

```
f (x -> x * x) # f : (Int -> Int) Int -> Int
    5
```

Variadic Functions

The last parameter but also the last return can be variable:

```
f : A B .. -> C D ..
```

Unlike usual, the arguments are not automatically collected as an array but accumulated using an intermediate function, which can be (re)implemented for each type. Nevertheless, all arguments are combinable into an array to pass them as a whole:

$\texttt{f expr}_1^A \ \{\texttt{expr}_2^B, \texttt{expr}_3^B, \texttt{expr}_4^B\}$

In case of indefinite multiple returns, a receiver is expected that fixes the exact number, which can be another function or multiple name binding:

$\texttt{c, d}_1\texttt{, d}_2\texttt{, d}_3 \texttt{ = f expr}_1^A \texttt{ expr}_2^B \texttt{ expr}_3^B \texttt{ expr}_4^B$

Modules

This section covers some aspects in advance from an expression level perspective.

Modules are values that implement concepts in different ways:

```
  module constraint (has decl | is expr^Tuple)
| module decl
| concept :: expr^Tuple
```

Additional definitions in the has-block that are not part of the module interface described by the concept remain inaccessible from outside.

Expressions that evaluate to modules or to functions returning a module may only be assigned to a closed namespace:

```
concept Incrementing t has incr : t -> t

incrementing = module Incrementing Int has incr i = i + 1
```

Module contents are called using dot notation:

```
x = incrementing.incr 5 # → x = 6
```

If the module interface is already known from the context, the corresponding constraint does not have to be specified in the head of the module expression:

```
incrementing : Incrementing Int

incrementing = module incr i = i + 1
```

Module Implementation by Tuple Expression

As a shortcut to direct definition within a has-block, a tuple can instead be passed through an is-clause, which provides implementations for all methods in the order of their specification in the respective concept:

```
incrementing = module Incrementing Int is i -> i + 1
```

Or just the concept is mentioned beforehand while relying on type inference:

```
incrementing = Incrementing :: i -> i + 1 # extreme shorthand syntax
```

Matching tuples get coerced into the expected module.

Modules as Parameters

In Kalkyl, modules are first-class citizens:

```
incr1 Incrementing::i = i.incr 1
```

Without a specified concept, the compiler cannot know the methods.

Parameters representing modules can also be installed directly to replace global instances:

```
incr1 Incrementing::i = incr 1 using i
```

3.3 Control Structures

All control structures, even loops, always return a value in the spirit of an expression-oriented language. However, since loops can contain mutable objects, they are discussed in the subsection on uniqueness types.

Conditionals

Multiple conditions are separated by a vertical bar instead of verbal repetition using `else if` to appear less verbose and favor single-line expressions:

```
[if | unless] cond then expr [| cond then expr ..] else expr
```

The vertical bar as a separator is only mandatory between two conditions within a line, and can otherwise be omitted:

```
if
    cond then expr
    cond then expr
    …
    else expr
```

> Due to the highly simplified syntax of the if-construct, there is no need for an equivalent to guards in Haskell at the declaration level.

Furthermore, `if` and `unless` are applicable like functions:

```
[if | unless] cond expr_then expr_else
```

or, in combination with `else`, as ternary operators:

```
expr [if | unless] cond else expr
```

Case Expressions

As an alternative to definitions, patterns can also be matched within expressions:

```
case expr of pattern -> expr [| pattern -> expr]_n

pattern.singleton = literal | deconstruction | □_a | $□_a | &□_a | _ | ..
pattern.deconstruction = □_A [singleton ..]
pattern = singleton [| singleton ..]
```

Here, too, the vertical bar between cases in different lines can be omitted.

The programmer is responsible for ensuring that all conceivable cases are taken into account to avoid errors at runtime. To prevent this, any value can be intercepted using an underscore `_` or variable (→ deconstruction). The ellipse `..` can be used to express that the remaining arguments of a constructor are irrelevant.

Within patterns, values can be invoked using the `$` naming operator to distinguish them from variables of a deconstruction.

As with conditionals, the expression of the first matching case, from top to bottom, gets evaluated and returned.

Matching Arguments of Function Expressions

If several parameters of a lambda expression are to be matched at the same time, they do not first have to be combined into a tuple for a case expression:

```
f = \ 1 2 3 -> 6 | 4 5 6 -> 15 | .. -> 0
```

is the same as:

```
f = \a b c -> case a, b, c of 1, 2, 3 -> 6 | 4, 5, 6 -> 15 | .. -> 0
```

Variant Expressions

Control structures in Kalkyl are predestined to return expressions of different types depending on the condition:

```
let x = True in if x then x else "nope"
```

The type of the overall expression is `Bool + Text`, since either a Boolean value or a character string is returned depending on the condition. Values of such nameless – or also literal – sum types always require a case discrimination, just like with conventional sum types:

```
if x in? Bool then … else …
```

> The overloadable operator `in?` tests whether the value on the left occurs in the value on the right, which can be any conceivable collection. Data types are thus understood as value sets.

Alternatively, a decomposition takes place in case expressions by manual typing:

```
case x of _ : Bool -> … | _ : Text -> …
```

Furthermore, the constructors or literals of different types can be compared directly next to each other:

```
case x of False | "nope" -> False | _ -> True
```

The advantage of nameless sum types lies in the fact that there are no new constructors; hence, no cumbersome deconstructions have to be carried out just for the sake of sum types. For example, a `Maybe` type with its own data constructors `Nothing` and `Just` is no longer necessary:

```
33 : None + Int
```

The type operator `?` offers a shorthand notation for optional types:

```
33 : ?Int # same as: Int + None ~ None
```

Sequences by Evaluation Order

In contrast to imperative languages, definitions of a Kalkyl file are not executed from top to bottom, but merely represent names that bind expressions. Hence, there are no blocks of statements with relevant order either, so that the source code behaves more like a sheet of paper with mathematical formulas. For example, a constant used in a function can be defined anywhere in the following lines of the same scope. Nevertheless, even in such a declarative language there is an inherent order, which the compiler can interpret as control flow:

```
b = exprb
f x = a ○ (b □ x)
a = expra
```

It does not matter whether **a** is introduced before or after **f**.

Due to parentheses, operator associativity, and other rules that dictate which subexpression is to be evaluated first to obtain a total value, a clear order is given.

Assuming left-associative operators, **a** is evaluated first, followed by the parenthesized expression, which overrides any operator precedence between ○ and □.

This functional approach means that instead of instructions executed according to their order, functions with concrete return types must be applied, which linked together form an overall expression:

```
echo    : Text *IO -> *IO
(>>)    : (a -> b) (b -> c) a -> c
program : *IO -> *IO
program = echo "Hallo, " >> echo "Welt!"
```

***IO** is a unique type that represents external states and is thus only known at runtime. The chapter on substructural types provides precise details.

In this way, complex actions can be mathematically formulated as chained computations whose algorithmically determined evaluation order implicitly models a control flow.

3.4 Operators

Operators are special punctuations that behave like functions. However, some operators are significantly restricted in their applicability.

As part of a concept, most operators are overloadable for new types. Unless otherwise stated, infix operators are always left-associative.

Specifying Operators

Signature	Explanation
`(:)   : t {t : Type} -> t`	specifying the type of an expression
`(as)  : a {b : Type} -> b`	type conversion requiring module for `Castable a b`
`(is?) : t {t : Type} -> Bool`	check whether expression evaluates to a specific type

Intrinsic Type Operators

Signature	Explanation
`(* _)     : Type -> Type`	uniqueness type
`(^ _)     : Type -> Type`	pointer type
`(~)       : {t : Type} t -> Type`	default value
`(? _)     : Type -> Type`	option type with `None` as default
`(_ !)     : Type -> Type`	lazily evaluating type

Data Operators

Signature	Concept	Characteristic
`(_ in? _)  : a b -> c`	`Containing`	
`(_ , _)    : a b -> a b`	—	tuple building
`(_ => _)   : k v -> Pair k v`	—	key-value building
`(_ >< _)   : a b -> c`	`Cartesian`	
`(_ .)      : n -> Nat+ where Castable n …`	`Differencing`	Numbering

Just as with the semicolon and flow operator, the key-value builder behaves like the comma, but with stronger precedence to eliminate the need for parentheses:

`{1 => "foo", 3 => "baz"}` = `{(1, "foo"), (3, "baz")}`

Logical Operators

Symbols	Related Concept
`or`	`Alternative a b~a c~b`
`xor`	`Contravalent a b~a c~b`
`not`, `and`, `nand`, `nor`, `xnor`	`Boolean b where Alternative b, Contravalent b`

An implementation for `Alternative ?a a` should be preinstalled as a Kalkyl-specific variant of the Elvis operator `?:` as it exists in other languages.

Comparison Operators

Signature		Related Concept	Alternative
`(=)`	`: t t -> l`	`Equal t l`	`equal?`
`(/=)`	`: t t -> l`	`Equal t l`	`unequal?`
`(>)`	`: t t -> l`	`Comparable t l where Equal t l`	`greater?`
`(<)`	`: t t -> l`	`Comparable t l where Equal t l`	`less?`
`(/<)`	`: t t -> l`	`Comparable t l where Equal t l`	`greater-or-equal?`
`(/>)`	`: t t -> l`	`Comparable t l where Equal t l`	`less-or-equal?`
`(>=)`	`: t t -> l`	`Comparable t l where Equal t l`	`greater-or-equal?`
`(=<)`	`: t t -> l`	`Comparable t l where Equal t l`	`less-or-equal?`

General Operators

Signature		Related Concept	Characteristic
`(_ + _)`	`: a b -> c`	`Adding`	commutative
`(_ - _)`	`: a b -> c`	`Reducing`	
`(_ * _)`	`: a b -> c`	`Multiplying`	
`(_ ** _)`	`: a b -> c`	`Potentiating`	
`(_ / _)`	`: a b -> c`	`Dividing`	
`(_ // _)`	`: a b -> c`	`Extracting`	

Markup Builder

Signature		Related Concept
`(:>)`		
`(<:)`	`: elem attr -> Markup lang`	`Attributing elem attr lang`
`(</)`	`: elem cont -> Markup lang`	`Nesting elem cont lang`

Composing and Sequencing Operators

Signature		Related Concept	Characteristic
`(_ <> _)`	`: a b -> c`	`Concating`	noncommutative
`(_ >> _)`	`: a b -> c`	`FlowingRight`	
`(_ ; _)`	`: a b -> b`	—	just sequencing
`(_ << _)`	`: a b -> c`	`FlowingLeft`	right-associative
`(_ *> _)`	`: a b -> c`	`TransmittingRight`	
`(_ <* _)`	`: a b -> c`	`TransmittingLeft`	right-associative
`(_ || _)`	`: f1 f2 -> f3`	—	evaluating in parallel

The semicolon `;` behaves like the right flow `>>` operator as it returns the second operand. The only difference between them is the different precedence, which makes bracketing unnecessary depending on the context.

Assignment Operators for Flow Variables

Signature		Explanation
`(:=)`	`: (Var t) t -> ()`	assignment
`(<=)`	`: (Var t) t -> ()`	assignment by copying and borrowing
`(<-)`	`: (Var t) (x -> t y) -> (x -> y)`	extracting a result by binding it

Memory Related Operators

Signature		Explanation
`(_ ^)`	`: ^t -> t`	dereferencing
`@`	`: t`	address operator

Operator Precedence

18	`.` `x^`
17	`::` `:/` `$` `@`
16	function application including methods `&`
15	`*type` `^type` `type!`
14	`do`
13	`**` `//`
12	`*` `/` `><` `\`
11	`+` `-`
10	`~` `==` `/=` `>` `<` `>=` `=<`
9	`<>` `||`
8	`>>` `<<` `*>` `<*` `^>` `<^`
7	`:=` `<=` `<-`
6	`in?` `not`
5	`and` `nand`
4	`or` `nor` `xor` `xnor`
3	`as` `:`
2	`=>`
1	`,`
0	`;`

3.5 Evaluation Strategies

By default, expressions in Kalkyl are strict, meaning they are evaluated in place regardless of their actual use. Nevertheless, Kalkyl provides naming conventions and a type operator to evaluate only when needed.

Static Evaluation

All expressions that neither contain IO values nor require a runtime are statically evaluable, i.e. their values can be determined at compile time and are therefore permitted as arguments of type constructors. This applies to literals and function calls with only literal input, regardless of whether the literals are bound to a name.

> In imperative languages, a keyword such as "const" tells the compiler that a statically evaluable expression is present, but restricted to literals and basic operations. In Kalkyl, such limitations are not necessary, since program effects get mapped to code level by special values and thus become tangible, which conversely means that their absence enables static evaluation.

However, for the sake of simplicity, no syntactical distinction is made between values that can be evaluated statically and values that are only known at runtime. If a value is used at type level, the compiler only checks for the absence of IO and pointers:

```
max = 100
i = 58 : Int 0 $max #
```

> The dollar sign `$` is there only to ensure that called values at type level are not confused as type variables.

Outsourcing

For larger data objects that are already known at compilation time, there is the option of storing them in separate text files. Such value files can be read directly by the compiler using the include function:

```
# include : Path -> a where Cast Str a
```

```
help-content = include "content/help.txt"
```

Since the include always starts from the project's root directory, a relative path does not have to be explicitly stated. Furthermore, the reference to the folder `resources` – or alternatively `assets` – can be omitted, as `include` always searches in this first, if it exists:

```
elem_data = include /chemistry/pt.csv : Map Pt.Symbol Pt.Data
```

Using specific instances for the `Cast String t` concept, text data in any format can be read in during compile time, so that there is no negative impact on the program. Since the compiler also excludes any errors, such values can be used without any `*IO` involved.

Laziness

4 Declarations

Every program written in Kalkyl consists of declarations, which are processed according to their well-defined meaning. In this sense, declarations represent commands to the compiler to make values accessible or to bind them in some other way, for example as part of specifications.

Basic Syntax

The syntax of all declarations can be derived from the following basic principles:

Specifications

Specifications are announced with a colon `:`, also called typing operator:

```
□ : type
```

Definitions

To bind an expression to a new identifier, the equal sign `=` or keyword `is` serves as a definition symbol:

```
□  [pattern] = expr
keyword □ₐ [param] is expr
```

Definitions with an equal sign are also referred to as "equations".

Only values and data constructors can be defined in the form of equations, each differentiated from one another by capitalization.

In contrast to all other declarations, several equations can exist for the same identifier, each with different patterns. Definitions with `is` are, however, excluded from this possibility, since they bind very specific expressions such as a path to a repository, a concrete type expression or the conversion to another unit of measurement.

References

To refer to multiple existing identifiers in a given context, a specific keyword clarifies the purpose:

```
keyword □ [, □]ₙ
```

The general term "references" in relation to declarations denotes listings of existing identifiers, which mostly appear as clauses in other declarations.

Nestings

Declarations that provide context for other declarations to which they are related in some way introduce the subordinate scope with **has** or **with**. Unlike **has**, **with** announces definitions whose scope is limited to the surrounding construct:

```
keyword □ [param] has decl with decl
```

Correct indentation must be observed, namely exactly one depth, which by default is either a tab or, if previously set, a certain number of expected spaces.

Overview

Kind of Identifier	Syntax
component	`component` $\square_A$ `[: constraint | provides {id}]`

namespace	`use (` $\square_A^{\text{Component}}$ `|` $\square_a^{\text{Module}}$ `[lit]`$_n$`) [unqualified | under` $\square_A$`]`
	`[with ids]`

namespace	`section [`$\square_A$ `has ] decl`
	`[with decl] ↲`

[specification]	`□ : type`
value	$\square_a$ `[pattern] = expr` `[using {local-module}] [with decl] ↲`

type + data	`type` $\square_A$ `[param]`
	`[is type | has data]`
	`[deriving {template}] ↲`

type + data	`(subtype | supertype) [param] of type` $\square_A$
	`[is type | has data]`
	`[deriving {template}] ↲`

[data definition]	`[id`$_{\text{type}}$`::]`$\square_A$ `[pattern] = expr`
	`[using {module}] [with decl] ↲`

measuring unit	`(metric | nonmetric) □ (: dim | is quantity)`

constraint
any others
	`concept` $\square_A$ `[param] [| param]`
	`[given {fundep}]`
	`[forall {type-var}]`
	`[where {constraint}]`
	`[has decl] ↲`

alias	`alias` $\square_A$ `[pattern] is expr`$^{\text{Type}}$ `↲`
[installation]	`(install | preinstall) module`

```
 identifier =   [type::][□_A .]_n □_A [as [□_A .]_n □_A]
              | [@ □_A .]_n □_a [unqualified | as [□_A .]_n □_A]

        ids = identifier [(, | ↲) identifier]

constructor = □_A [(expr^Type)_n | : expr^Type]

       data = constructor [(| | ↲) constructor]

 constraint = □_A [expr^Type]_n

     module =   module □_A [param]
                    [forall {type-var}]
                    [where {constraint}]
                    [has decl] ↲
              | □_A^Constraint :: lit^Tuple
              | □_a^Module [lit]_n
```

Except for installed modules, all declarations have in common that they introduce new identifiers, either as references to existing bindings or as definitions.

> The specification often mentions clauses and blocks. The difference between these two is that a clause binds a specific expression, if necessary several separated by a comma; whereas blocks refer to subordinate declarations.

As a purely functional language, Kalkyl has no global states, only immutable name bindings. Due to this inherent referential transparency, the subtle distinction between constants and immutable bindings, expressed in some languages by different keywords, becomes unnecessary, which also results in a greatly simplified definition syntax.

4.1 Values

All expressions assigned to a lower-case identifier are referred to as values:

```
hello-de = "Hallo Welt!"
```

Value bindings occur either at the top file level including some has -blocks, or locally in with-blocks and let expressions. However, parameters could also be seen as another form of value binding, where an argument is assigned a name whose visibility remains limited to the scope of a function.

Macros

Macros represent bound values that must be known at compile time. Consequently, only functional control structures, including function values, as well as static constructor calls, literals and a few primitive operations as a subset of Kalkyl are available. However, this restriction allows values to be called in types and patterns:

```
t = True

f $t    = 1
f False = 0
```

Parametric macros may only receive literals or other macros as input. In contrast to constructors, placeholders or deconstructing variables are not permitted.

The compiler checks whether a value is actually known at compilation time when it is called with the dollar sign `$`.

4.2 Function Definitions

Functions are defined either declaratively or as expressions:

	Declaration Style	**Expression Style**
plain	`f a b … = expr`	`f = \a b … -> expr`
matching detailed syntax	f pattern$_1$ = expr$_1$ f pattern$_2$ = expr$_2$ f … = …	f = \ x -> case x of pattern$_1$ -> expr$_1$ pattern$_2$ -> expr$_2$ … -> …

> Function expressions only need to be enclosed in parentheses if the parameters do not immediately follow a keyword or punctuation mark such as `=`.

In the declarative style, the parameters or patterns appear one after the other on the left-hand side of the equal sign, in contrast to a function expression to the right.

Both variants behave identically and can be arbitrary combined with each other:

```
f a = \b -> a + b # f : a -> b -> c where Adding a b c
```

The parentheses-free syntax comes from the fact that functions are curried:

```
x = f 5 12
```

or

```
x = (f 5) 12
```

is the same as

```
x = f(5)(12)
```

> Parentheses are not required between atomic arguments such as mere literals.

Consequently, functions are partially applicable, strongly favored by the currying, and – at least declared ones – also capable of recursion by their name.

Pattern Matching

Moreover, functions are definable as a set of independent equations:

```
answer True  = "Yeah"
answer False = "Nope"
```

Starting from the top, the first applicable equation will be used. Also, as with case expressions, underscores, ellipses and macros are available.

However, all equations of a function must be located one after the other within the same namespace.

Empty Expressions in Functions

A special case is given by parameters of type `()`, which as a unite type only has one valid value `()` with absolutely no semantic meaning – in contrast to `None` – so that such parameters do not even need to be applied:

```
f () = …
x = f
```

This means that empty expressions behave similarly to optional values. Consequently, if the function itself is to be referenced, ellipsis is required:

```
g = f ..
```

Lack of Meaningful Return

Functions that return an empty expression are only executed for the purpose of certain effects such as assignments:

```
# (:=) : (Var t) t -> ()
# (;) : a b -> b
# process : *?R -> ()
f () = x := new exprₓ; process x; x # f : () -> *?R where Runtime
```

At this point, it should only be said in advance that the function `f` is still “pure” despite internal assignments and heap allocations, since it always delivers the same result at runtime under identical conditions. The further sections on uniqueness types and references provide more explanation.

Program Expression

The entry point – or initial node – of every Kalkyl program is called “program”,

```
program : *IO -> *IO
```

which accepts the “current state of the world” by some IO and returns a “changed state of the world” as some IO. More details about this are explained in the chapter on substructural types.

4.3 Data Types

There is one overarching syntactical construct to define new types and their related data constructors in a generalized algebraic way:

```
type CountryCode has … | CZ | DE | DJ | DK | …

type StreetOrPOBox t has
   Street Text
   POBox t

City = Text
Zipcode = Text
   Mere type aliases to emphasize the purpose.

type MailingAddress t has
   MailingAddress (StreetOrPOBox t) City Zipcode CountryCode

addr = MailingAddress (Street "Max-Mustermann-Str. 9")
   "Musterhausen" (Int4 12345) DE
# addr : MailingAddress Int4
```

After the keyword **data** stands the name of a new type constructor, followed by possible parameters of the kind **Type** unless specified manually.

> Only type constructors without parameters or after their complete application are called types.

The keyword has announces a block in which all value constructors belonging to the new type – or type family when generic – are declared, while separated from each other by a vertical bar. However, an explicit separator **|** is only mandatory between data within a line, otherwise the vertical bar can be omitted.

> New data types usually have one or more data – or value – constructors, which together are often simply referred to as the “data” – or values – of a type, especially in the case of their full application.

New types are also definable as dataless by omitting the has -block:

```
type Nothingness
```

Mere synonyms get declared without the keyword **type** through a type expression in an is-clause instead of a has -block because synonyms do not introduce new types and hence do not have their own data constructors.

> In the case of parameterized data constructors, the related type is called a **product type**, while the values passed as arguments can be regained through **deconstructing** (→ pattern matching). Each fully applied value constructor represents a **variant** of its type or, in other words, an element of the value set described by the related type declaration. If there are several value constructors, a **sum type** is used in technical terms.

Construction of Named Tuples

Kalkyl automatically defines data constructors based on their specification by returning all arguments as retyped tuples (→ identity map). Consequently, labeled parameters result in a named tuple.

```
type MailingAddress t has MailingAddress
    -str-or-po StreetOrPOBox t
    -city Text
    -zipcode Text
    -country CountryCode
```

> In relation to the initial example, labels not only improve readability, but also make aliases unnecessary, which often only obscure the real type without any gain and hence should not be the first choice.

Labels are also available in shorthand notation of data declarations to communicate the purpose of arguments and allow access to them using dot notation after construction:

```
addr.city
```

Parameterless constructors are thus nothing more than empty tuples `()`.

Seeming Constructor Overloading

It would be extremely impractical to outright prohibit name collisions between constructors, not least because in the vast majority of cases it is already clear from the context whether a type constructor is meant or to which type a value constructor belongs:

```
type Contrast has Black | White

type Color has Black | White | Blue | …

w : Color
w = White
```

The value constructors of different types within the same namespace may have identifiers clashing with each other, but this comes with the cost that if the type is not already known from the context, it must be specified by an additional type-related qualification:

```
b = Contrast::Black
```

In contrast to qualifications through namespaces, type contexts made with a double colon `::` are only necessary in cases of ambiguity.

In order to avoid unintuitive situations, there is also the restriction that type and value constructors can only be named identically if they belong together, i.e. were introduced by the same declaration, for instance:

```
type ID has ID Int8
```

Supertypes

The data constructors of an existing type can be incorporated directly:

```
type Optional t has t::* | None deriving coercible -from t
```

All foreign data constructors must be adopted, either individually at a specific position or all at once using the syntax `::*`. This complete integration leads to an inherent convertibility into the supertype, so that a corresponding instance for coercions can be derived.

Such supertypes offer the advantage that existing data constructors can be reused instead of wrapping them in a new one,

```
type Maybe t Just t | Nothing
```

which simplifies pattern matching in particular.

Generalize Algebraic Data Types

Writing the parameter types of a value constructor one after the other directly following the name is actually just a convenient shortcut for complete data signatures:

```
type MailingAddress has
    MailingAddress: StreetOrPOBox City Zipcode CountryCode -> MailingAddress
```

Both ways of declaring value constructors can be freely combined with one another,

```
MailingAddress StreetOrPOBox City : Zipcode CountryCode -> MailingAddress
```

which is particularly useful if the return type needs to be stated, for example in the case of a parameterized type:

```
type ID _ has
    Email : Text -> ID Text | PK Int : ID Int
```

> If, as in this example, the parameter name is unneeded, an underscore is sufficient to indicate that a parameter is present.

Due to the already defined return types of the data constructors, the value range of the type parameter is limited to a few specific cases and thus automatically forms a refinement of the kind `Type`:

```
ID : {Text, Int} -> Type
```

This fixation offers the advantage that when using such a specialized type constructor, other variants are already excluded, so that they no longer have to be taken into account when matching patterns. In addition, type inference is made easier since the exact type is already known automatically through the value constructors.

Value Dependent Types

To a certain extent, type constructors are also parameterizable by values:

```
type Partial t undef has
   Partial -val {v : t} -undef {undef : Set t} : {v | v not in? undef}

a = Partial 58 {0} # → a : Partial i {0} where Integer i
```

The compiler interprets variables of refinements that have the same name as one of the superordinate type parameters as a **coupling**, so that value-level arguments get automatically passed on to the type constructor. Such couplings then appear as `value promotions` in the types of the respective constructors to reflect dependencies:

```
Partial : -t {t : Type} -undef Set t -> Type
```

```
Partial::Partial : -val t -undef {undef : Set t} -> Partial t undef
```

> Combined with refinements, value-dependent type constructors provide the ability to create customized type variants without having to declare new constructors for every possible case.

Or without extra type information on the individual refinement variables, since this can be inferred from the application of the type constructor:

```
Partial::Partial : -val t -undef {undef} -> Partial t undef
```

> The syntactically simplified refinement `{undef}` does not define any type variable, since the type, namely `Set t`, is already known, but rather introduces a variable for its value and thus promotes the passed argument to the type level. Therefore, the curly brackets are essential as a syntactical minimum in order to still distinguish between type variables and value variables.

By coupling parameters of the data constructor with those of its corresponding type through refinement variables, the compiler no longer expects arguments for the respective parameters of the value constructor if the type is already known, i.e. the type constructor has already been fully applied:

```
x : Partial Int4 {0, 1}
x = Partial 37
```

If there is only one value constructor, and with just one missing argument, even the constructor no longer needs to be called explicitly if the type is known:

```
x = 37
```

Smart Constructors

Algebraic data types, taken by itself, offer no guarantees about the values that can be deconstructed. Nevertheless, it may be quite desirable to permit only particular arguments. For this reason, declarations of data constructors can be provided with a pre- and postconditions, whose compliance is checked when all relevant parameters are applied:

```
type Domain has Domain
    -subd {subd : ?Text}
    -host {host : Text}
    -sld {sld : ?Text}
    -tld {tld : Text}
    -> {(subd, host, sld, tld) |
        subd&length + host&length + sld&length + tld&length =< 253}

k = Domain -host "kalkyl" -tld "dev"
```

In this example, the method `&lenght` returns 0 if the argument is `None`.

The name introduced by the set builder allows access to the passed value within the overall specification. Any checks can be written this way, as long as the expression evaluates to a Boolean value.

If there is only one simple value constructor needed, the has-block can be replaced by the is-clause, causing the compiler to automatically create a standard value constructor:

```
type Domain is
    -subd {subd : ?Text}
    -host {host : Text}
    -sld {sld : ?Text}
    -tld {tld : Text}
where
    subd&length + host&length + sld&length + tld&length =< 253
```

To improve the readability of multi-line refinements, the keyword `where` can be used instead of the vertical bar `|` introducing predicates.

Defusing

If there is no 100% guarantee that a smart constructor – or any other function with conditions – will be called without a value conflict, the possible program-terminating error can be converted to a “None” value:

```
# maybe : (x -> y) -> (x -> y?)
domain = Domain &maybe -host "kalkyl" -tld "dev"
```

Custom Implementations

The return type of a value constructor can be freely specified:

```
type Concat t has Concat t t -> t where Concatenating t
```

However, if a data type other than the type associated with the value constructor is stated, or if the return does not correspond to a tuple described by its parameters, the compiler cannot derive an automatic implementation. Instead, the programmer is expected to provide a value constructor definition himself:

```
Concat::Concat a b = a <> b
```

To avoid confusion with type functions, it is mandatory to name the associated type constructor, separated by a double colon `::`.

A value constructor can also be implemented manually if there is no different return type, which is useful, for example, to guarantee the required value range instead of or in addition to a refinement:

```
type Floor has Floor : r -> Integer where Rational r

Floor::Floor = floor
```

Such custom implementations are suitable when an always valid range of values can be guaranteed in a sensible way. A post condition would be unnecessary in this case.

Subtypes

If all value constructors of a data type B return a tuple of the same type A, B is a subtype of A, also called the supertype of B. Smart value constructors with pre- or postconditions can be used to restrict the value range of supertype A, so that subtype B represents a proper subset of A:

```
type Even has Even {i : Int | i &mod 2 = 0} deriving coercion
```

or with complete data specification:

```
type Even has ~ : {i : Int} -> {i | i &mod 2 = 0} deriving coercion

x = Even 6 + 8 : Int # automatically converted into `Int`
```

> If the value constructor is named the same as the type constructor, the tilde `~` can be used as an abbreviation character.

Or as a simplified declaration using the is-clause:

```
type Even is i : Int | i &mod 2 = 0
```

> The brackets around an overall tuple or set can be omitted in is-clauses.

Instead of literals, individual variants are also excludable:

```
type Weekday has Mon | Tue | Wed | Thu | Fri | Sat | Sun

type Workday is {d : Weekday | d /= Sun}
```

> The compiler automatically creates instances for the concept `Equal`.

Since the new type represents a subset, values can easily be converted back into objects of the original type if necessary, provided an instance for concept `Coercible subtype supertyp` is defined.

Type Aliases and Macros

Kalkyl distinguishes between type aliases, which are preserved when querying the specification of an identifier; and type macros, which get replaced by the compiler:

	Alias	**Macro**
syntax	`alias` $\square_A$ `[param] is type`	$\square_a$ `[pattern] = expr`$^{\text{Type}}$
behavior	compatible and displayed	replaced and hence not displayed

Another difference is that aliases have their own data constructor, provided that the type they are equated with also has a data constructor of the same name:

```
type ByteSize has Tiny | Small | Normal | Big

type Int size has
    Int : {val : i where Integer i} {size : ByteSize} -> {val}

# Int::Int : i {size : ByteSize} -> Int size

Int8 is Int Big

i = Int8 1234567890 # i : Int8
```

In contrast, macros do not appear in the specifications at the end:

```
toc : ?Int4 -> Type

toc None = Str
toc i = {i}, Str

c : toc 9
c = 9, "Lorem Ipsum" # c : {9}, Str
```

Such bound type expressions can in turn be used as macros at the type level.

Reuse of Existing Constructors

For more complex data summing values of different types, it is recommended to just define a synonym for a nameless sum type instead of introducing completely new data:

```
IdentityVerification is Person + Int
```

The compiler can then infer the sum type from the type space of the alias:

```
pid = IdentityVerification::Int 2876051550
```

In the event of name collisions between data constructors of different types, additional qualification can be made using the space of the respective type:

```
pid = IdentityVerification::Int::Int 2876051550
```

or just

```
pid = IdentityVerification::Int::2876051550
```

Declared Named Tuples

As already explained before, all labeled parameters form one named tuple, which, unless a custom implementation is made, represent also the result of the data constructor:

```
type Person has
   Person -name Text -age Int 1 150 -residence Text

p = Person "Ulf" -residence "Leipzig" -age 36
name = p.name
```

If only one data constructor named like the related type constructor is needed, the tuple can be specified directly after the alternative keyword `is`:

```
type Person is -name Text, -age Int 1 150, -residence Text
```

Based on the given tuples, the compiler automatically derives the associated value constructor; but either only named or unnamed elements may be specified.

Flexibly Applicable Record Constructors

Thanks to the profound currying in Kalkyl, it is still possible to pass arguments for each field of a tuple parameter directly:

```
# Person::Person : -name Text -age Int 0 150 -residence Text -> Person
Person (-name "Ulf", -age 36, -residence "Leipzig")
Person ("Ulf", 36, "Leipzig")
Person -name "Ulf" -age 36 "Leipzig"
Person "Ulf" "Leipzig" 36
```

If the constructor – and generally every function - has additional parameters, the other arguments can subsequently be passed as usual. Based on the specification, the compiler is still able to figure out which value serves which parameter. If an argument is missing for a parameter that expects a tuple, the compiler assumes that the individual elements are preferably and directly assigned first, with field names optionally appearing as labels.

Updating

Similar to Haskell, there is a handy way to update existing records instead of having to create them from scratch:

```
p1 = Person "Horst" 55 "Magdeburg"
p2 = p1 "Manfred" 42
another-Horst = p1 -residence "Jena"
```

If there is a requirement, compliance is checked after each update.

Merging

Tuples, whether named or unnamed, can be combined into an unnested overall record:

```
p2d = -x 1.7, -y 2.5
p3d = p2d, -z 6.3
```

Since a tuple can only consist of either unnamed or named fields throughout, the compiler interprets the presence of unnamed elements with record values next to named fields as a request to merge them into a new unnested tuple. The same principle applies when defining new record types:

```
type P2D is -x, -y Float64

type P3D is P2D, -z Float64 deriving coercible P2D
```

> As soon as the individual records have fields with the same name, the compiler issues an error to avoid inconsistencies.

Merging offers the advantage that all functions processing values of the included data are equally applicable to values of the new encompassing type, provided that all of its value constructors return corresponding variants. Accordingly, a respective instance for coercion can be derived.

The merging data constructor either completely incorporates existing objects, or creates the fields ad hoc with existing individual values:

```
a = P2D 1.2 3.4
b = P3D a 5.6
c = P3D 1.2 3.4 5.6
```

Under the hood, the object to be merged could simply be copied statically – provided it is a constant known at compile time; or the required stack gets fixed in advance, provided the type is not polymorphic and thus its space requirement precisely known; or an existing dynamically created object gets reallocated for the additional fields.

Units of Measurement

To declare units of measurement, some work must be done beforehand:

```
# type BaseDimension has T | L | M | I | 0 | N | J

# type Dimension

MeasuringUnit : Dimension -> Type
MeasuringUnit _ = Type

Quantity : -num Type -units MeasuringUnit dim -> Type
Quantity _ _ = Type

#! intrinsic
preinstall module Multiplying
    (BaseDimension + Dimension) ~ Dimension | c -> a b

# and preinstalled modules for the operators `**` and `/` …
```

The additional functional dependency, related to the parameter names of the respective concept, further restricts possible installed modules, which in this case enables the compiler to infer the operand types based on the known return **Dimension**; since further combinations with other types are not needed here.

Using the three operators `*`, `**` and `/`, any dimension can be defined in the regular language of Kalkyl. Due to the corresponding rules of precedence, no parentheses are required: `**` > `*` > `/`

The power operator is actually redundant because L^2 can be expressed just as well as **L*L**; but it doesn't hurt to have an additional notation that corresponds to mathematical conventions.

The type constructor **MeasuringUnit** returns the set of all unit symbols of a given dimension. On the other hand, **Quantity** constructs a quantity type describing the combination of a numerical value with a unit.

Declaration of Metric and Nonmetric Units

Units of measurement are declared by specifying the dimension:

```
metric m : L

x = 50 m          # x : Quantity i m where Integer i
y = 49:Int8 m + x # y : Quantity Int8 m
```

The keyword **metric** initiates the declaration of a unit of measurement and causes the compiler to introduce additional units of measurement with metric prefixes such as “mili” `m*` and “kilo” `k*`. If this is unwanted, a non-metric unit can be defined instead using the keyword **nonmetric**:

```
nonmetric inch of L
```

Unit Matching and Deconstructing

Units of measurement can be understood as unary postfix constructors. Consequently, pattern matching is possible as well to recover quantified numerical values:

```
calc-back (x cm) = x / 100
calc-back (x dm) = x / 10
calc-back (x m) = x
calc-back …
```

The individual metric prefixes form different variants of the base unit.

Derived Units of Measurement

In addition, Kalkyl allows new units to be derived from existing ones:

```
metric N is kg m / s**2
```

This causes the compiler to automatically install appropriate conversion rules.

Unit Conversion

With declarations, not only a value is bound to a name, but also the identifier itself. In fact, the symbol `m` is associated with two values, namely a quantity-constructing function and `m` as an identifier of type `MeasuringUnit dim`:

```
m : MeasuringUnit dim : num -> Quantity num m
```

This additional meaning allows units of measurement to be passed as identifying symbols, which is necessary to carry out conversions:

```
concept Conversion num from to has
    convert : Quantity num from -> (Quantity num to | from&dim = to&dim)

# Conversion : -num -from MeasuringUnit d -to MeasuringUnit d -> Constraint

as : from -> (to: Type) -> to where Cast from to
   | Quantity num from -> (to: MeasuringUnit dim) -> Quantity num to
     where Conversion num from to

metric K : 0
metric °C : 0
nonmetric °F : 0

preinstall module Conversion °F °C has
     convert (x °F) °C = 0.(5) (x - 32.0) °C

preinstall module Conversion °C K has
     convert (x °C) K = x + 273.15 K

t = 64 °F as °C
```

Using the intrinsic method `dim` of unit symbols, which returns the dimension, a postcondition is formulated ensuring that units of measurement are only convertible into other units of the same dimension.

Specific Physical Quantities

It is usually not practical to fix the unit of measurement directly in the type as physical quantities often have different measures. Refinements can be used to define such quantities for which several different units of the same dimension are permissible:

```
Length n = Quantity n m | m : {m, inch}
```

The second parameter of `Quantity` remains generic to a limited extent by allowing a certain range of units. However, in order to convert one unit of measurement into another, a respective instance for the cast concept is nevertheless required.

4.4 Concepts

Concepts correspond to type classes in Haskell and thus are primarily used for overloading of values and even constructors in a systematically fashion:

```
concept Continuous t has
   next : t -> t
```

> The values declared within a concept are called methods. Furthermore, the convention of naming concepts after adjectives is recommended.

A concept declaration introduces a new identifier that appears as a constraint on type parameters in signatures of its methods or in generic values calling them:

```
next : t -> t where Continuous t
```

Constraints are summarized in a where-clause as conditions that a type must fulfill so that those methods work for its values. To satisfy such a constraint, a type-specific implementation for the concept is required, which can be done in two ways, either coherently or by a local module installation.

> To avoid confusion with classes in object-oriented languages and emphasize the importance as a means of abstraction, type classes are referred to as “concepts” in Kalkyl, also because the alternative term “interface” is too general, since in fact everything that can be called has an “interface” somehow.

Preinstallations

The mechanism to enable ad hoc polymorphism necessitates that concept implementations are globally coherent. This is ensured by having exactly one preinstalled module for a specific type either in the translation unit of the respective concept or in the unit of its type declaration, so that the compiler can automatically find the type-matching implementation:

```
type Direction has Top | Left | Bottom | Right

preinstall module Continuous Direction has
   next = \ Top -> Left | Left -> Bottom | Bottom -> Right | Right -> Top

x = next Left # → x = Bottom
```

Thanks to this coherence of preinstallations, the programmer does not have to specify the type, as the appropriate module gets automatically selected based on method usage. This also has the practical side effect that as long as only generic methods are used, the overall construct remains unspecific, which means that algorithms can be developed in a completely generic way.

Local Module Installations

Global uniqueness is the price of ad hoc polymorphism, but a price that experience shows programmers prefer over non-automatic solutions. Nevertheless, in some cases it may be necessary to replace an existing instance with another one or to provide an implementation for a foreign type that is not present in the compilation units of the respective concept or type declaration. Therefore, Kalkyl provides a second mechanism for using alternative implementations, regardless of any pre-installed modules:

```
crosswise = module Continuous Direction has
    next = \ Top -> Bottom | Left -> Right | Bottom -> Top | Right -> Left

x = next Left using crosswise # → x = Right
```

Or implemented in place while relying on type inference:

```
x = next Left using Continuous :: Top -> Bottom | Left -> Right | …
```

> In the using-clause, modules may only be called or defined using the short syntax `::`, which is solely intended for modules that consist of just one or two methods and not as a replacement for proper module implementations with `module`.

Any number of fully applied modules can be given, whose scope is limited to the previous definition; but direct module implementations are not permitted. This interchangeability is also the reason why instances of concepts are called modules in Kalkyl.

The methods of a bound module can also be called directly, provided that only that specific implementation is needed:

```
x = crosswise.next Left # Crosswise.next : Direction -> Direction
```

First-Class Modules

The initial case example shows that modules are bound to names like values. Consequently, they must also have a type:

```
crosswise : Continuous Direction
```

In any case, at least the concept must be known so that the compiler knows which method is meant; the rest can be done by type deduction:

```
duplex Continuous::mode initial = initial : Direction, mode.next initial

# duplex : (Continuous Direction) Direction -> Direction Direction

d = duplex Crosswise Bottom # → d = Bottom, Top
```

Unless a specification is given separately, the module name must be preceded by the underlying concept.

Multi-parameter Concepts

Concepts can have zero to – theoretically – any number of parameters. However, from a practical point of view, it rarely makes sense to have more than three or four parameters. A Kalkyl compiler should therefore allow at least up to 6 type parameters.

The basic rule for concepts is that every type parameter must also appear in the type of every method so that the compiler can determine the appropriate implementation:

```
concept Concating a b~a c~b has
    #! infixl 6
    (<>) : a -> b -> c

preinstall module Concating (b -> c) (a -> b) (a -> c) has
    (<>) g f = \x -> g (f x)
```

> Kalkyl's standard library makes heavy use of very generalized multi-parameter concepts that contain only a single operator overloadable for different purposes.

It is advisable to use the first argument as the default value for the other parameters, which accommodates the common case when the same type applies to each parameter:

```
preinstall module Concating Text with …
```

Restriction Regarding Nameless Sum Types

Furthermore, it should be noted that if a concept parameter is part of an unnamed sum type, it is not considered to be properly involved, since the compiler may not be able to determine the exact implementation:

```
concept Concating a b~a c~b has
    #! infixl 6
    (<>) : a + Text -> b -> c
```

> If a text value is passed to the operator, it is impossible for the compiler to automatically figure out the correct implementation.

For defined supertypes, the required module can at least be found through an explicit type context:

```
type ID is Text + Integer # → ID::ID : Text + Integer -> ID

install module Contemplative ID has …

x = ID "XYZ" <> ID::Integer 123 # or ID "XYZ" <> ID (Integer 123)
```

Functional Dependencies

For type classes, especially multiparametric ones, where the return type of methods is generic, or a concept parameter is part of a sum type, the compiler can no longer automatically determine the appropriate module:

```
x = "Hallo," <> " Welt!"
```

→ Error because the type of the concept parameter c cannot be determined.

Consequently, we have to tell the operation's return type, which is cumbersome:

```
x = "Hallo," <> " Welt!" : Str
```

This obvious disadvantage defeats the purpose of polymorphic operators. Hence, it doesn't really make sense for the third type parameter `c` to be completely free. For this reason, a functional dependency between the parameters should be definable, which conveys to the compiler: If `a` and `b` are known, then `c` is also known.

```
concept Concating a b~a c~b given a b => c has
    #! infixl 6
    (<>) : a -> b -> c
```

The additional condition states that although any type can still be used for `c`, only one specific case of `c` is permitted for each combination of `a` and `b`. Consequently, functional dependencies between concept parameters limit the possible number of globally coherent modules. Related to the last concept, if there is already an implementation for `Concating Str`, no further implicit module such as `Concating Str Str Bool` can be offered. In the vast majority of cases, however, this restriction is not a problem, since it hardly makes sense, for instance, to overload the concatenation operator's return type independently. Nevertheless, this multi-parameter concept allows to provide an implicit module that assigns an element to an existing list:

```
preinstall module Concating (List t) t (List t) has (<>) = List.append
```

Ultimately, anything other than an enlarged list as a return wouldn't make sense here.

Shorthand Notations

Referring to the previous example, the functional dependency can also be written as

```
concept Concating a b~a | c~b has …
```

which, moreover, causes the compiler to not necessarily expect an argument for the third parameter `c` in where-clauses, making specifications more concise.

When `a => b` and `b => a`, there is the shorthand notation `a <=> b`, which means: If an implicit module `C` T_1 T_2 is given for the concept `C a b given a <=> b`, then there may not be another instance where `a =` T_1 or `b =` T_2. Consequently, it is sufficient if the type for the first or second parameter is already known to infer the other.

Nullary Concepts

Concepts without parameters represent a special case, as only one global instance is possible, which makes no difference whether the methods are simply defined as ordinary values. Therefore, nullary concepts are primarily useful for local implementations of their methods to model implicit parameters with which entire compilation units can be indirectly parameterized:

```
component LanguageSetting provides Language, LanguageSetting, lang

type Language has DE | EN

concept LanguageSetting has lang : Language

# @Lang : Lang -> Module LanguageSetting
lang x = module LanguageSetting has lang = x
```

Just a handy module template.

```
# app : IO where LanguageSetting
app = echo << lang of DE -> "Hallo Welt!" | EN -> "Hello world!"
```

And then in another compilation unit:

```
component GermanInterface

use LanguageSetting unqualified

install lang DE # or: install LanguageSetting::DE

program = do
    app
    …
```

Conceptual Types

Concepts serve the purpose of providing an interface for which different implementations can be offered depending on the type. Since constructors are nothing more than constants or functions, it makes sense to be able to overload them too:

```
concept JSON-Serializable a has
   type JSON-Rep a has
     JSON-Rep : a -> Text deriving Castable Text

preinstall module JSON-Serializable Bool has
   JSON-Rep::JSON-Rep = True -> "true" | False -> "false"
```

> The automatically derived preinstallation for converting to text allows the value to be recovered for intermediate processing.

It would also be conceivable to offer a normal method instead of a constructor to translate various objects into JSON. However, this approach provides additional type safety and easier communication through an explicit type. Furthermore, encapsulating text through a new type makes it possible, for example, to overload existing operators for text, specifically for the purpose of generating JSON.

All concept parameters, as with normal methods, must appear in both the type and data declarations, as long as they are not functionally dependent. However, a value constructor within a concept only has to be implemented by a module if there is a specific return type, otherwise the compiler can derive the definition as usual.

Module Templates

Bound parameterized modules are referred to as module templates:

```
continuous-direction order = module Continuous Direction with
   next = \
     Top -> order.0 | Bottom -> order.2
     Left -> order.1 | Right -> order.3

duplex mode initial = (initial, next initial)
   using continuous-direction Bottom Top Right Left
```

However, the actual main motivation for first-class modules is the ability to program templates in order to automatically generate standard implementations of concepts directly for a new type in a deriving clause:

```
concept Continuous t forall d : {d | Return d = t} has
   The refinement ensures that the return type is t so that only value constructors of type t are allowed.
   next : d -> d
     The abstraction through d makes it possible to also include parametric value constructors.

continuous t = Continuous t has
   next ctor = let
          ctors = constructors t
          i = (constructors t &index ctor) + 1
          max = ctors &count
     in
          ctors[i if i < max else 0]

# index : Array (Constructor t) -> Constructor t -> ObjIndex
# count : Array (Constructor t) -> ObjIndex

type Direction has Top | Left | Bottom | Right deriving c-ontinuous
```

> The exact specification of `constructors` can be found in the section “Properties of Sum Types” regarding the Base library.

When using module templates in deriving clauses, the new type is implicitly passed as the first argument.

Implementable Features

Concepts are also a means for systematically introducing features into the language. Accordingly, each type that meets hypothetical conditions can provide an instance for such a special concept, which enables certain mechanisms.

Indexes and Slices

For all objects whose types provide a global modul of the following concepts, individual elements can be accessed using index / slice syntax or dot-notation:

Concept	Enabled Syntax	Expression Type

`Sliceable`	`expr[int]`
	`expr[start: | :end | start:end[:step]]`
		`ObjIndex`

`Hashing`	`expr[expr`$_{\text{hashable}}$`]`	`t where Hashable t`
`Named`	`expr[text`$_{\text{field}}$`] | expr . field`	`FieldName t`

With slices, as with ordinary indexing, negative values result in a count starting from the end of a data structure. In addition, reversed sections can also be obtained in that the start expression describes a larger index than the target position.

If an attempt is made to access an element that does not exist using an index, `Undefined` is returned. Unless the number of elements is already known at compile time, `t + Undefined` is always returned instead of `t`, which requires appropriate handling.

Type Coercion

Instances of the concept `Coercible` cause the compiler to automatically convert values to required types.

Generic Literals

Kalkyl strives to remain generic as much as possible by specifying literals with a concrete type only when required:

Value	Literal Origin Type	Generic with Constraint	Example
integers	`Literal.Integer`	`i where Integer i`	`6286`
rational numbers	`Literal.Rational`	`t where Rational r`	`2445.83`
data block	`Block t len`	`d where Data d`	`{9, 91, 5, 2}`

Irrational numbers have to be expressed as approximations in the form of rational numbers.

The constraints require implementations of concepts that include the method

```
cast : from -> to
```

to automatically convert literal values into the requested subtypes.

OS Dependent Installations

Many important functions are part of an operating system's API. Concepts offer the opportunity to define a uniform interface for Kalkyl code. Additional attributes preceding installations indicate which OS each concept implementation is intended for:

```
type RTLD has Lazy | Now | Global | Local # Run-Time Linker Directives

type Plugin = -path Path, -flag RTLD

type PluginHandle = ^Void, *IO

concept Resourcing options handle | options <-> handle has
    open : options *IO -> *handle
    close : *handle -> *IO

preinstall module Resourcing::Unix Plugin PluginHandle has
    header = ./dlfcn.h
    open opt io = ccall header "dlopen" opt, io
    close hdl = ccall header "dlclose" hdl.1 >> hdl.2

preinstall module Resourcing::Windows Plugin PluginHandle has
    header = ./windows.h
    open opt io = ccall header "LoadLibrary" opt.path, io
    close hdl = ccall header "FreeLibrary" hdl.1 >> hdl.2
```

> The concept `Resourcing` represents a generally binding interface for opening and closing resources. Consequently, operating system-specific instances are unavoidable.

Normally this code would be invalid because a concept has been implemented multiple times for the same types. However, the attributes cause the compiler to only use the preinstallation for the respective platform. So if the program is compiled on a Unix-like system such as Linux or macOS, the appropriate module is used while the Windows version gets discarded.

4.5 Components

Each Kalkyl file `*.k` represents a separately compiled unit called **component** providing a scope for declarations of values, constructors as well as modules.

A certain part of the declarations is made accessible to other compilation units through a well-defined interface listed at the beginning of the file:

```
component Example provides Foo, Bar, baz as baas, …
```

> If no provides-clause is given, nothing will be exported.

The component name must match the file name: `Foo-Baz.k` → `component Foo-Baz …`

By default, the content of a component is not accessible from outside. Data constructors must either be listed individually `T::D` or exported all at once using the syntax `T::*`. Furthermore, it is possible to rename public members for other components using `as`.

Conceptualized Components

Instead of directly listing all entities provided by the component, it is also possible to define the interface using a concept:

```
component Example : Exemplary

concept Exemplary has …
```

> The concept does not necessarily have to be declared in the same file, but can also be imported from another component.

In this way, components behave no differently than modules. In fact, a component is a kind of parameterless module just one level higher in the organization, where the entire file itself forms a module expression.

Conceptualized Components primarily play a role as plugins that implement the interface of an existing application. But it is also conceivable to use components for providing specific configurations:

```
# ./ de.k
component DE : Language

…
```

To tell the compiler that a local component should be installed instead of a module, a leading dot is required:

```
# ./ example.k
component Example

install .DE
```

Without the dot, the compiler would otherwise assume a known module. For components to be installed from parent directories, an additional dot must be stated for each further level `..DE`.

Importing

Contents of other components are introduced using the use-declaration:

```
use Math
```

The compiler automatically makes all public entities of a component accessible under its given export name; nevertheless an alternative namespace can be stated using `under`:

```
use Math under M
```

The name of the component (export name) only acts as the compiler's first choice of namespace. This mechanism can be bypassed with the modifier `unqualified` afterwards, so that the content is instead imported directly into the global namespace:

```
use Math unqualified
```

Rather than adopting everything into the global namespace, there is the option of only selecting certain identifiers by adding a with-clause:

```
use Math with Complex as C, pow, pi under Constance # or: as Constance~
```

This variant can also be combined with as-clauses and the use of namespaces, where the tilde is available as an abbreviation to avoid repeating the identifier to be imported.

Main Component

The main component is the compilation unit that has the same name as the directory in which it is located:

- `./liba`
  - `A.k // main component`
  - `B.k // subcomponent of A`
  - `C/`
    - `C.k // internal main component`
- `./x`
  - `X.k // main component`
  - `Y.k // subcomponent of X`
  - `…`

> The prefix `lib` is not part of the component name, but only tells the compiler that a program library is present with corresponding implications.

All other components of that directory are not callable from the outside. Their content, which should be publicly accessible to external components, must be re-exported from the main component forming a library API. For applications, it is the program definition of the main component that represents the entry point.

The root of the source directory may only contain folders for main components, each representing a subproject and written in lowercase to indicate the compilation name.

> "Subproject" is a general term to summarize all top-level main components, which can be either program libraries, applications or plugins.

The main component must always be mentioned in the declaration of a subcomponent:

```
component A.B provides …
```

Using this information, the compiler is able to automatically detect the relationships between components and the level of nesting within a subproject.

Local Imports

To simplify importing identifiers from other local compilation units, each component, no matter where it is located, is always viewed from the root of the source directory instead of relative to the respective importing unit:

component	A	A.B	A.C.A	A.C	A.C.D	X
`liba/A.k`	–	–	–	–	–	`use A`
`liba/B.k`	`use A.B`	–	`use A.B`	`use A.B`	`use A.B`	–
`liba/C/A.k`	–	–	–	`use A.C.A`	`use A.C.A`	–
`liba/C/C.k`	`use A.C`	`use A.C`	–	–	–	–
`liba/C/D/D.k`	–	–	`use A.C.D`	`use A.C.D`	–	–
`x/X.k`	–	–	–	–	–	–

The path of a compilation unit is always the same, no matter in which file.

Theoretically, the component `X.Y` could import the main component `X`. However, this does not make much sense, since only `X` can be called from outside and circular dependencies are not allowed to ensure a clear order between all components.

The lack of distinction between folders and files makes it easier to subsequently split a component into subcomponents without causing interface breaks.

Relative Import Paths

Since absolute import paths within subprojects – which are actually relative to the source directory – can be cumbersome, there is the alternative of relative imports with a preceding dot to express that a component in the same directory is meant, whereas two dots refer to the parent directory:

component	A	A.B	A.C.A	A.C	A.C.D	X
`liba/a.k`	–	–	–	–	–	`use A`
`liba/b.k`	`use .B`	–	`use ..B`	`use ..B`	`use ..B`	–
`liba/c/a.k`	–	–	–	`use .A`	`use .A`	–
`liba/c/c.k`	`use .C`	`use .C`	–	–	–	–
`liba/c/d/d.k`	–	–	`use .D`	`use .D`	–	–
`x/x.k`	–	–	–	–	–	–

For each higher directory level, another dot must be added.

The dot notation is only applicable to components of the same subproject.

Re-exporting

Except for the main component, all other components of a directory cannot be accessed directly. Referring to the previous example, to use `liba/C/D/D.k` from `liba/A.k`, the public content of `liba/C/D/D.k` would have to be re-exported from `liba/C/C.k`:

```
component A.C provides D unqualified

use .D
```

If a subcomponent is not needed by the re-exporting component itself, the use-declaration can be omitted by making a relative reference in the providings:

```
component A.C provides .D unqualified
```

Re-exports in relative notation can also be combined with `unqualified` and as-clauses.

Prelude

Each source directory can contain a special component file `Prelude.k` whose providings get automatically imported into every component of the respective directory and all subdirectories. However, subdirectories can overwrite the global prelude by their own.

If no custom prelude is defined, Zuse uses the default prelude, which automatically imports fundamental things such as data types and concepts of the basic library.

~~Data Modules~~

~~Data modules are special Kalkyl files that only represent values, similar to the formats used for this purpose such as INI or JSON. Consequently, declarations are limited to the module header, which describes the interface, and the module body containing mere definitions:~~

```
~~module C~~

~~a = …~~
~~b = …~~
~~…~~
```

~~Other declarations such as ones of types or imports are not permitted. Instead, the header provides a context in which the module is to be developed.~~

~~In addition, all providings and used library types of the component from which the concept was imported are automatically available in the data module. The stated public concept thus defines a precise interface for which a Kalkyl compiler can check a data module in advance. Consequently, data modules are integrable in various ways:~~

~~**at**~~	~~**as**~~	~~**Advantages / Disadvantages**~~
~~compile time~~	~~part of the project~~	~~fastest solution, but inflexible~~
~~runtime~~	~~precompiled / binary~~	~~more efficient than text data, but requires compiler~~
~~runtime~~	~~textual data~~	~~the most flexible, but also the slowest approach~~

~~If the data module is precompiled in advance, translation and type checks at runtime are no longer necessary. However, this approach requires that the module is loaded from a secure source to prevent the execution of malicious code.~~

It is questionable to what extent this is really useful, or whether it cannot simply be replaced by JSON or a custom format as a subset of Kalkyl.

Sections and Namespaces

A section provides a way to further subdivide a compilation unit by having its own scope limiting the visibility of local module installations.

```
section
    install module C A …
    f x = …

g x = … # The implementation for `C A` is not visible here.
```

As another use case, the compiler could consider sections as logical units for incremental translation. However, this is solely a matter for the implementation.

Named Sections

Namespaces can be understood as general domains to which top level names are assignable either indirectly as part of a named section,

```
section A has
    f x = …
```

Top level name bindings are everything that doesn’t stand in with-clauses or let-expressions.

or directly as an addition in dot notation:

```
A.f x = …
```

A section name does not have to be unique, so that multiple sections can share the same namespace, provided no name collisions between their entities occur.

Within a named section, bindings can call each other without qualification, which includes declarations that have been introduced elsewhere under the same namespace.

Local Bindings

If declarations should only be visible to certain unit members, there are two approaches to archive this:

Definitional With-Clause	**With-Clause of a Nameless Section**
def$_{\square}$ with decl	section decl with decl

`fib 0 = 0`
`fib (1 | 2) = 1`
`fib n = (φ**n - (-φ)**(-n)) /`
`    sqr5 &round`
`with`
`    sqr5 = 5**(1/2)`
	`section`
	`    φ = (1 + sqr5) / 2`
	`    fib n = (φ**n - (-φ)**(-n)) /`
	`        sqr5 &round`
	`with`
	`    sqr5  = 5**(1/2)`

Both constructs introduce a new scope to which the declarations defined in the with-clause are limited. The only difference is that the definitional with-clause refers exclusively to all previous equations of the same value.

5 Substructural Types

The main goal in Kalkyl's design is to ensure referential transparency despite program effects such as heap allocations or IO-interacting with the outside world. More precisely, the wanted programming language must allow direct access to dynamic memory and other resources without losing the mathematical character of its functions. However, these objectives exclude an externally implemented abstraction in the form of a runtime system for maintaining referential transparency. Rather, only library functions, implemented in the language itself, should be necessary. To meet these seemingly contradictory requirements, Kalkyl is designed based on a substructural type model, which enables effects through a set of rules in a purely functional language without shielding resources from the programmer:

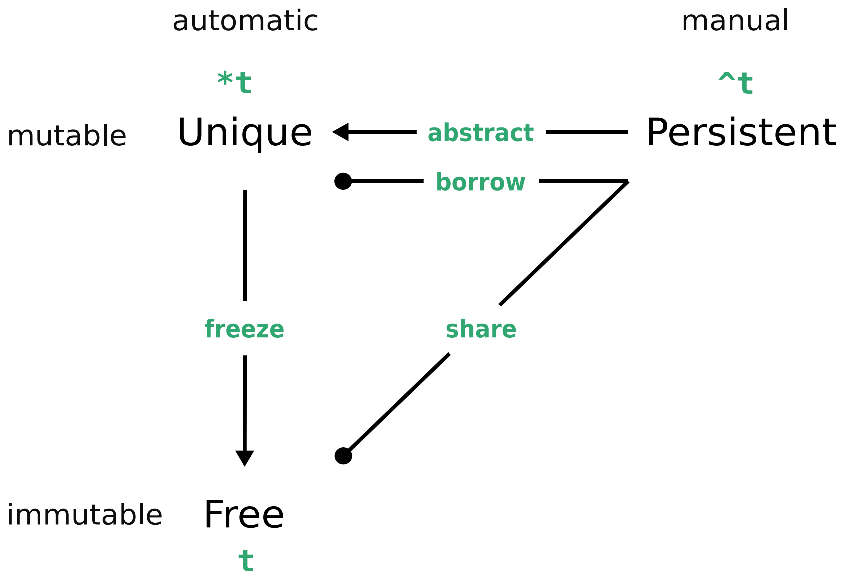

Fig 3: Model of Kalkyl’s Substructural Type System

> This overview maps the additional substructural information of data types in order to determine the permitted use of their values and thus guarantee referential transparency, memory security and correct resource handling.

The uniqueness types were adopted from Clean, whereas Kalkyl's persistent types correspond to the “steadfastness” postulated by Philip Wadler; in other words, types whose values are also unique but must not lose their uniqueness, which guarantees correct handling of allocated memory. The term “free”, on the other hand, simply means non-unique values such as global constants or immutable let bindings.

Persistent types, or "pointers" for short, serve the purpose of implementing highly efficient dynamic data structures in Kalkyl itself, while certain rules as well as protection mechanisms ensure memory security, as far as this is possible or seems sensible.

Storage Classes

Kalkyl knows three storage types:

Type	Implemented as	Allocation at	Content is
static	code / data segment	compile time	immutable
dynamic	heap	runtime	mutable
automatic	stack	call time	immutable or mutable

Kalkyl does not know static memory for mutable objects, since uniqueness – or the ability to be mutable – is only allowed for parameters or local bindings within expressions. Dynamic memory, on the other hand, must be mutable in principle, as it requires special handling.

Nevertheless, the main function **`program`** offers the potential to allocate memory at compile time instead of expanding the stack, since **`program`** is only called once, namely at program start, so that problems with multiple parallel calls cannot arise.

> Theoretically, all local bindings of the main function – and even the entire stack of every function call inside **`program`** – could be made static to generate more performant output. But this should be the sole responsibility of the compiler.

Substructural Properties, Usage and Storage Location

Generally speaking, a mutable value can have at most one reference at a time, while immutable objects are arbitrarily referenceable. Furthermore, a distinction is made as to whether a type is simply “unique” with a disposable value, or has a “persistent” value pointing to a dynamically allocated memory location.

The destructivness and discardability of unique values requires that the memory location is managed automatically, which means that only the stack comes into question, whereas persistent types represent heap objects normally:

Mode	Type	Attributes	Derived Usage	Storage
unique	**`*t`**	**mutable** automatic	→ once at a time or loss of **`*`** → discardable	automatic
free	**`t`**	**immutable** automatic	→ referenceable → discardable	static or automatic
persistent unique + linear	**`^t`**	**mutable** manual	→ once at a time → must be freed	dynamic

In order to allow memory safe destructive updates, all mutable values must be somehow "unique", i.e. only one reference can refer to them at the same time. To ensure this, the unique value itself must come from a source that is equally unique in some way or where it is guaranteed that no references exist. These conditions can be solved in different ways: initialization with a literal, an implicitly passed memory state at program runtime, or a direct parameter of the main function that symbolically represents the "state of the world".

5.1 Uniqueness Types

Uniqueness types ensure that values are used in a single-threaded way, meaning they are never called by multiple parts of the program at the same time without losing their uniqueness and thus the ability to be mutable. This allows safe destructive updates without giving up referential transparency, while the compiler checks the code for compliance with the rules.

To specify a unique type, the asterisk `*` serves as a unary type operator with low precedence over verbal type constructors:

```
append : (*Array t) t -> *Array t
```

> This function expects an array with no other reference pointing to it and can thus make changes without side effects, whereupon returning the original array instead of a copy.

Formation of Unique Values

To obtain a unique value, an existing unique object is required:

```
File.open : Path *IO -> *File
```

> The very first unique IO value can only be obtained as an argument of the main program. All interactions with the "outside world" always require either a parameter of type `*IO` or another unique "world object" like a file as a substitute to maintain referential transparency.

Another possibility to create unique objects is offered by the special function `new`:

```
init x = Array::new x # init : data t -> *Array t where Runtime
```

> The initial uniqueness in this case comes from the state of the heap at runtime, which is automatically passed through an implicit parameter – expressed by the concept `Runtime` – when the program starts.

Briefly anticipated, `new` is a special method which, when implemented for a pointer to a data structure, allows heap allocations with automatic release – similar to RAII.

To preserve referential transparency, there is the restriction that unique objects may only be used inside functions with one or more unique Arguments. In the simplest case, a parameterless function is sufficient for heap objects, since `Runtime` already specifies a unique parameter implicitly:

```
init () = Array::new 1 2 3 4 # init : () -> *Array Int where Runtime
```

Regarding resources such as files, this is indirectly enforced by the need for a unique IO value as the initial input.

Loss of Uniqueness

A unique value does not necessarily have to be used exactly once, but loses its uniqueness if ownership is unclear due to multiple references, which consequently makes the value immutable:

```
f x = let
    a = A::new x
    b = exprb ○ a
    c = exprc △ a # Error because △ : X *A -> Y
in c
```

In this example, **a** is used multiple times, resulting in the loss of its uniqueness, although the operation △ requires a unique argument, since data may be overwritten in place.

As soon as the uniqueness is lost and the function does not return such a heap object, the dispose method gets automatically called. Uniqueness types thus serve, in combination with the mechanism just described, as a simple means of automatically managing dynamic memory without having to resort to a GC.

Approaching Imperative Syntax

Since most functions do not destroy unique objects but pass them back for further processing, their return must be bound to a name:

```
x = let                         # x : *Vector t where Runtime
    a = new Vector 1 2 3 4
    b = a &append 6             # append : (*Vector t) t -> (*Vector t)
    c = b &…
in
    c
```

IO and Sequencing

Kalkyl deliberately avoids monads and encapsulating of states from the rest of the code, as this is not composeable and generally difficult to convey. Instead, the programmer should work directly with these states representing program effects. This is possible because the entry point, called `program`, has a parameter of type `*IO`, whose value models the physical world as seen by the program:

```
program: *IO -> *IO
```

The idea is that this unique IO value needs to be threaded through all functions that somehow communicate with the outside world. In this way, referential transparency remains preserved without abstracting away external states:

```
# echo : Text *IO -> *IO
# get : *IO -> Text *IO

program io = let
    io2 = echo "Hello, what’s your name?" io
    name, io3 = get io2
in
    echo ("Nice to meet you " <> name) io3
```

> The name bindings `name` and `io3` can also occur before the first one `io2`, since the compiler would still figure out the actually required order of evaluation.

Since `program` expects a unique IO value back, the IO argument may only be used exactly once at the same time in order to maintain its uniqueness. Therefore, involved functions need to return a new unique IO value in parallel with any results. This leads to an evaluation order, although the arrangement of definitions plays no role in let-expressions. But the logic that one expression must first be evaluated to get the next value nevertheless implies an order that ultimately forms the program flow.

Flow Operators

Since it can quickly become cumbersome to constantly introduce new names or nest function calls in complex programs, Kalkyl offers so-called flow operators such as `>>`, which are heavily overloaded to thread returned state-representing values between the individual function calls point-freely :

```
# (>>) : a b -> c where FlowingRight a b c

# preinstall module FlowingRight (x -> a) (a -> y) (x -> y) has
#     (>>) f g x = g (f x)

program = echo "Hello, what’s your name?" >>
    get >> (name, io -> echo ("Nice to meet you " <> name) io)
```

Like a pipeline, a data stream takes place between the individual subprograms, each of which carries out independent actions receiving the result of the previous one as input.

> In fact all flow operators are generic. The exact interface and implementation details including `IO` can be found in the respective subchapters on the Base library.

Approaching Imperative Syntax

With **do**, individual actions are likewise composable into an overall function that can be equated with **program**, so that the ***IO** argument does not have to be passed explicitly:

```
# do : first -> second where Doing first second

program = do
    echo "Hello, what’s your name?"
    get >> (name io -> echo ("Nice to meet you " <> name) io)
```

The special function **do** is made variadic through a concept, whose first implementation maps identically while another module returns a new function for **second**, which processes the first argument or result of existing composition with another argument using the flow operator **>>**.

Sequence Variables

In many cases it is more intuitive and practical to cache results in variables instead of passing them on immediately. For this purpose, Kalkyl offers sequential variables:

```
# (<-) : (Var t) (r -> t r) -> (r -> r)

program = do
    echo "Hello, what’s your name?"
    name <- get
    echo ("Nice to meet you " <> name)
```

The operator **<-** takes an action, pops out the first result to bind it to a sequence variable, and returns a new function without the popped result.

The action-related assignment operator **<-** introduces a flow variable, which, unlike bindings of declarations or let expressions, is only visible in the subsequent control flow and therefore depends on the evaluation order. This allows flow variables to overshadow preceding sequential bindings of the same name, regardless of what type they are.

Assignments

Resource Management

There is a general API in Kalkyl to access external resources:

```
type IO
The only way to get valid values is the IO parameter of `program`.

concept Resourcing handle | options given handle <=> options has
    open : options *IO -> *handle
    close : *handle -> *IO

concept Reading handle | options buffer has
    read : options *handle -> *buffer *handle

concept Writing handle | buffer has
    write : *buffer *handle -> *handle
```

Instances for Ergonomic Program Flow

```
use module FlowingRight (hdl -> hdl) (hdl -> a hdl) (hdl -> a hdl) has
    (>>) f g = function x -> g (f x)

use module FlowingRight (hdl -> a hdl) (a hdl -> b hdl) (hdl -> b hdl) has
    (>>) f g = function x -> g (f x)

use module FlowingRight (hdl -> a hdl) (a hdl -> hdl) (a hdl -> hdl) has
    (>>) f g = function x -> g (f x)
```

Since the main program requires the unique IO argument to be returned, resources must be closed in any case and thus behave linearly like pointers, from a practical perspective.

```
try : (*IO -> *handle) *handle.. *IO -> *IO where Resourcing handle

program = try
    open ./test.txt
```

Standard Streams

```
type Stdin t has
    Line : Stdin ()
    Exact ObjSize : Stdin ObjSize
    Until (Array Byte) : Stdin (Array Byte)

type Stdout has Stdout t where Cast t Text

type Stderr has Stderr Text

#! intrinsic
preinstall module Reading IO (Stdin ()) Text
#! intrinsic
preinstall module Reading IO (Stdin ObjSize) (Array Byte)
#! intrinsic
preinstall module Reading IO (Stdin (Array Byte)) (Array Byte)
#! intrinsic
preinstall module Writing IO Stdout
#! intrinsic
preinstall module Writing IO Stderr
```

Example

```
program = write (Stdout "Hallo Welt!")

program = write (Stdout "Pls enter your name: ") >>
    read In::Line >> (buf io -> write Stdout ("\nHello " <> buf) io)
```

Or piped using the special variadic function `do`:

```
program = do
    write << Stdout "Pls enter your name: "
        # or: write (Stdout "Pls enter your name: ")
    read Stdin::Line
    write Stdout ("\nHello " <> buf)
```

Explicit Console Functions

If IO interactions with the console are outsourced to their own functions and are not written directly in `program`, it can improve readability to express this in the function type, since `IO` as a type does not say much about the way the side effect occurs:

```
abstract type Console is *IO # Console : Resource
```
Types whose constructors contain IO are automatically considered resource types.

```
preinstall module Resourcing Console (Type Console) has
    open opt io = Console io
    close (Console io) = io


concept Printable t context has
    print : t context -> context
    # printf : t context -> context

preinstall module Printable x Console where Cast x Text has
    print x (Console io) = Console << write (Stdout x) io


say-hello name io = open Console io >> print name
    # say-hello : x IO -> Console where Cast x Text
```

Errors

```
type ErrNo has
    #! C
        ENOENT : Nat1
        #! only Unix
            …
```

Files

C	K	Constructor	∃! file	¬∃! file	Operations	Position
`r`	`R`	`Read`	—	→ failure	read	beginning
`r+`	`R+`	`Read-Write`	—	→ failure	read + write	beginning
`w`	`W`	`Rewrite`	truncate	create	write	beginning
`w+`	`W+`	`Read-Rewrite`	truncate	create	read + write	beginning
`a`	`A`	`Append`	—	create	write	end
`a+`	`A+`	`Read-Append`	—	create	read + write	end

```
type FileMode has
    Read/R | Read-Write/R+
    Rewrite/W | Read-Rewrite/W+
    Append/A | Read-Append/A+

preinstall Coercible


#! C "FILE"
C.File : Pointer

type File mode is
    -io IO
    -data C.File
    -errno ErrNo
    -mode {mode : FileMode}


# File : FileMode -> Type

File R+

scan : File r:{R, R+, W+, A+}
```

Concurrency and Parallelism

```
fork : *IO -> *IO ..
join : *IO .. -> *IO

(||) : (x1 -> y1) (x2 -> y2) -> (x1 x2 -> y1 y2)


start io = let a, b = fork io in join a b

start io = fork io >> f || g || h >> join
```

Loops

```
abstract type Jump t has Break t~() | Continue t~()

break =  Break
continue = Continue # continue : t~() -> Jump t

preinstall module FlowingRight (Jump a) (Jump b) (Jump b) has
    (>>) a b = b

preinstall module FlowingRight a (Jump b) (Jump b) has
    (>>) a b = b

preinstall module FlowingRight (Jump a) b (Jump b) has
    (>>) (Continue a) b = Continue b
    (>>) (Break a) b = Continue b
```

```
# iterate : x (x -> Jump x) -> x

echo n str io = iterate 0 io (x io ->
    if x >= n then
        break x, io
    else
        x + 1, write str io
)
# counted-echo : *IO -> Int *IO
```

In this example, the assigned expression can be simplified point-freely if parameter names are not needed:

```
echo n str = iterate 0 _ function x io ->
    if x >= n then
        break x, io
    else
        x + 1, write str io


echo n str io = let rec x io =
        n, io if x >= n else rec (x + 1) (write str io)
    in rec 0 io
```

Loop Kind	**Syntax**
endless	`loop` expr$_{\texttt{Jump}}$
head-controlled	`while` cond `loop` expr$_{\texttt{Jump}}$
foot-controlled	`loop` expr$_{\texttt{Bool}}$ `while` cond
collection-controlled	`for` var `in` collection `loop` expr$_{\texttt{Jump}}$

The abort command has an optional first argument, which can be used to specify the exact loop that is to be aborted, where 1 stands for the first – the outermost.

5.2 Pointers

Pointers in Kalkyl serve the sole purpose of managing dynamically allocated memory and are either “smart” or abstracted as references behaving like free or unique values:

`^t`	smart pointer to dynamic memory optimized to raw pointers with compiler flag to waive runtime security guarantees

`^s`
`forall s : Memory.Size`
	unspecific smart pointer to dynamic memory
	castable into any pointer whose dereferenced values fit in

`Valid ^t`	like `^t` but with the condition that only a valid object is referenced
`Subpointer a b`	tuple of entangled smart pointers pointer `^a` from which subpointer `b` can be reached
`*c where Disposal c`	abstracted pointer still unique and thus with updatable content that get automatically released
`c where Disposal c`	abstracted pointer to immutable content after construction including mutable intermediate states no changes are allowed

In addition, Kalkyl internally distinguishes between pointers whose addresses are on the stack and subpointers whose memory addresses are on the heap, so that they can only be accessed by dereferencing another pointer. For practical reasons and to keep signatures simple, no distinction is made between these two variants at the type level.

The constructed type `Valid ^t` is merely an alias for `{p : ^t | p /= Null}`.

Since a pointer whose address resides on the stack can and must be used exactly once at a given time, it is often necessary to return inner pointers together with the "main pointer". In fact, `Sub` is just a convenient type constructor to form such a pointer tuple,

```
type Subpointer a b~a has Subpointer -main ^a -sub ^b
# Subpointer : {t} Type~t -> Type
```

and to provide additional semantic meaning for the compiler, according to which the second pointer is dependent on the first one and behaves affinely, meaning that it does not have to be used as long as the first pointer is returned.

References

References are pointers that get automatically dereferenced and therefore appear like values to the programmer. However, this is only possible if the concept `Disposal` is implemented for the respective pointer type. This will be discussed in more detail in the following sections.

Storage Locations of Pointers, References and Values

Depending on where the memory address of a pointer is stored, a distinction is made between main- and subpointers:

	Type	Address on	Derefer	in C
main pointer	`^x`	stack	→ heap	`t *`
subpointer	`^x`	heap	→ heap	`t *`
related pointers	`Subpointer a b`	a: stack, b: heap	→ heap	
unique reference	`*T`	stack | heap	→ heap	`t *`
reference	`T`	data stack | heap	→ data → heap	`t *const`
value	`T`	data | stack | heap	—	`const t`

A referenced object is only storable in the data segment if it can be statically constructed, like a simple array. Therefore, a mechanism is required to allow the compiler to create more complex structures such as linked lists or hash maps at compile time.

Restrictions

Pointers to stack objects are not allowed to prevent accidental release or return from a function. If a stack object has to change, it must be created uniquely, so that when passed to a function, the compiler can reuse the existing memory space. Hence, pointers in Kalkyl are only intended to develop dynamic data structures, since uniqueness types and higher order functions safely cover the other use cases of pointers.

Furthermore, Kalkyl does not allow pointer arithmetic, as this is likewise not necessary since there are sufficient alternatives with flexible tuples and arrays to safely partition allocated memory.

In contrast to C, immutable pointer variables to mutable objects are not possible either, because the heap space simply cannot be released safely using an immutable pointer variable, as immutable values are freely usable. This constellation only occurs when sharing disposable objects, where after construction no data manipulation is allowed, while the implemented destruction method gets automatically called after leaving the scope, unless the object is returned.

Dynamic Memory Allocation

Using the tools provided by Kalkyl, a binding interface for managing memory is defined:

```
component Memory provides
    (^_), Runtime, Size, Allocating, Releasing, Reallocating, init, copy

(^_) : Type + Size -> Type
```

```
concept Runtime
```

Can only be implemented automatically at runtime and contains the heap state for initial uniqueness.

```
concept Allocating t | options given t <=> options has
    malloc : options -> ^t where Runtime

concept Releasing t where Allocating t has
    free : ^t -> ()

concept Reallocating a options | b where
    Allocating {a, b}, Releasing {a, b}
has
    remalloc : ^a options -> ^b where Runtime

init : t~$default -> ^t where Default t, Allocating t, Runtime
```

Guarantees that always initialized memory is allocated, thus performs better than `malloc` in safe mode.

```
copy : -dest ^t -src ^t -count ?Nat -> -dest ^t -src ^t
```

If count is `Null`, the entire object is copied, otherwise only `count` bytes from source to destination.

> The alias `Memory.Size` is primarily used to guarantee consistent preconditions everywhere and to exclude undefined behavior when translating to C or calling C system functions.

The constraint `Runtime` prevents the compiler from attempting to evaluate expressions that contain dynamic allocations at compile time.

This conceptual design allows OS-specific implementations to be provided idiomatically:

```
preinstall module Allocating t {t : Type}

preinstall module Allocating s {x | x&cast Nat+ = s}

preinstall module Allocating s {s : Nat+}

preinstall module Releasing t

preinstall module Releasing s:Nat+

preinstall module Reallocating a:Nat+ {b : Nat+} b
```

Since more specific modules are preferred over general ones, the programmer can overload `malloc` etc. for his data structures (→ Example: Linked List).

Memory Management of Blocks

Arrays with `Array t` represent an abstracted variant; bare data blocks, on the other hand, are created with the type syntax `t[n]`. However, Kalkyl requires that the size of a block of n-many elements is at least specified by the type system:

```
abstract type Array t is
    -length {l : Nat | l =< c} -capacity {c : Nat+} -data t[c]
```

For this reason, arrays stated by `t[n]` can only be used within tuples that record the available space with an additional element, overall value parameter or fixed number in the block type, which is intended to enable safe handling through potential checks. For example, the compiler knows that if an array `Nat[10]` resides in the middle of a tuple, an index for accessing its elements can only be between -10 and 9 in order to refine the type for further operations `Int -10 9`, also allowing negative indexes for backward counting.

However, the built-in implementation of `remalloc` can only increase or decrease the size of tuple objects whose last element is a primitive array with its capacity stored by the second-to-last element:

```
# Tuple : -min Nat ~ 0 -max Nat ~ $max-bound -> Type

preinstall module Reallocating
    {t : Tuple -min 2 | t&second-last is? {a: Nat+}, t&last is? t[a]}
    {b : Nat+}
    {t | t&second-last is? {b}, t&last is? t[b]}
```

If inner arrays are to be enlarged, much more complex operations are required, most likely with intermediate copies, so that for efficiency reasons it is better to work with pointers to further allocations.

Dereferencing

Since pointers – obtained through `malloc` – behave linearly, they must be used exactly once. This requires that processing functions either return such a pointer or finally consume it with `free`. Just as with unique values, there are binding rules that nevertheless allow a certain flexibility in dealing with pointers without losing referential transparency:

```
f () = let
    a = malloc (Int8, Int8)
    a^.1 = 123456789
    a^.2 = 987654321
    b = foo a                   # foo : ^(Int8, Int8) -> ^(Int8, Int8)
in b                            # f : () -> ^(Int8, Int8) where Runtime
```

> Due to the assumed uniqueness and the linear pointer behavior, the order of definitions does not play a role, so that the referential transparency is maintained despite dynamic allocation.

Since a pointer must be used exactly once to exclude multiple references or a missing release, this logically requires that the memory address is also unique. Furthermore, the uniqueness of the memory address implies that the content pointed to by the pointer is equally unique, meaning that a pointer can be dereferenced exactly once to assign a value. This dereferencing assignment is not considered an application of the pointer, since here the pointer variable only appears as a kind of namespace on the right side of the equal sign.

Since dereferencing requires a pointer from a previous allocation, there is an inherent chronology of evaluation regardless of the declaration order, which means that pointer handling still behaves like referentially transparent code, assuming a runtime constraint.

> Code describing dynamic allocations is not viewed by the compiler as being statistically evaluable, which in turn requires an additional unique value that is only known at runtime.

Pattern Matching With Pointers

The content to which a pointer refers can be matched with literals and constructors as usual, provided that dereferencing is explicitly carried out:

```
f x = case x^ of Int8 0 -> false | _ -> true

# f : ^Int8 -> Bool
```

However, this code must cause an error, since dereferencing to read values for pattern matching is not considered an application of the pointer – just as with unique values.

A pointer must either be returned or consumed by another function, which as its final action releases the dynamic memory:

```
f x = case x^ of Int8 0 -> false, x | _ -> true, x

# f : ^Int8 -> Bool ^Int8
```

As before, the rule still applies that functions that return pointers may only be used in other functions:

```
g () = case f (init Int8 0) of
    False, ptr -> free ptr


# g : () -> Bool, ^Int8 where Runtime
```

> A pointer is only available when the program starts, after which the constraint gets automatically satisfied. Therefore, in any case, the evaluation is deferred until runtime.

Unique values and therefore also pointers are not globally boundable, but can only be used locally in functions, so that at least a parameterless function must be defined.

Borrowing and Sharing

To apply existing functions to pointers, there are two intrinsic means, each of which returns the pointer renewed:

```
borrow : ^x (*x -> *x y) -> ^x y
```
Enables safe manipulation of data without pointer dereferencing.

```
alias Unique t is {*t, ^t}
```

```
share : (Unique x) (x -> y) -> (Unique x) y
```
Allows temporarily free use of unique values, including data referenced by a pointer.

Since pointers are also unique objects, it is perfectly safe to treat their content temporarily like ordinary values, as long as the original pointer is returned afterwards.

Additional Security Mechanisms

The direct use of raw pointers is not intended. Instead, Kalkyl follows the strategy that all safety mechanisms that must be present in the runtime code to signal memory errors can only be optimized away by a compiler option for all or certain components, which is primarily purposed for data structures that have proven to be error-free or are so simple that the programmer can already recognize without in-depth analysis that no memory errors can occur here.

> But even without the additional security guarantees of smart pointers, Kalkyl is nevertheless inherently much safer than C or unsafe Rust due to its refinement and uniqueness types as well as persistent pointers.

Additional Information

To guarantee memory safety through error messages, the following additional information is required for each allocated block:

`main`	pointer that addresses the very first heap object of the data structure
`owner`	pointer to the heap object that first stored that address
`incoming`	number of all pointers that refer to this heap object
`outgoing`	number of all pointers with which this heap object addresses others

For simplicity, the compiler could plan extra memory for these additional details in each allocation and automatically implement them as a struct. Furthermore, it is sufficient if incoming and outgoing only store 32-bit numbers, so that on 64-bit systems the additional memory requirement is no longer 24 bytes.

```
# a — b — c — d

# redispose : ^t -> ^t

c^.prev = a
a^.next = redispose a^.next^.next
```

Since there are no more pointers left that refer to b, b gets released.

```
type Tree a has Empty | Node a (^Tree a) (^Tree a) deriving Equal

preinstall module MemoryReleasing (Tree a) has free ptr = do


rewrite-tree : *Tree a -> *Tree a where Equal a
rewrite-tree
    Empty = Empty
    (Node 1 (Node 1 Null Null)^ Empty^) =
        (Node 1 (Node 1 Null Null)^ Empty^)
    (Node val left right) =
        Node val (rewriteTree left) (rewriteTree right)
```

Algebraic data types that contain pointers must themselves be pointers to ensure safe handling.

```
type Tree a has Empty | Node a (Ref Tree a) (Ref Tree a) deriving Equal

preinstall module MemoryReleasing (Tree a) has free ptr = do


rewrite-tree : *Tree a -> *Tree a where Equal a
rewrite-tree
    Empty = Empty
    (Node 1 (Node 1 Null Null)^ Empty^) =
        (Node 1 (Node 1 Null Null)^ Empty^)
    (Node val left right) =
        Node val (rewriteTree left) (rewriteTree right)
```

Pointer Entanglement

Even simple data structures such as a doubly linked list show a fundamental problem with pointers: If the next-to-last element is deleted without first deleting the last element, the only pointer that refers to the memory address of the last element gets lost. The result would be a memory leak. To avoid this, it is usually the programmer's responsibility not to make such mistakes. But Kalkyl offers a mechanism that entangles pointers with each other:

```
type Node t is -data t, -prev ^Node t, -next ^Node t

l = let
    p1 = malloc (Node Str)
    p2 = malloc (Node Str)
    p1^ = "Lorem ipsum", Null, p2
    p2^ = "Etiam ultricies", Null, Null
in p1
```

> Since pointers are steadfast (unique and linear), the variable `p2` can only be used exactly once, so it is guaranteed that the node referenced by `p1` stores the only reference.

Or simplified by calling the init function in the dereferenced context of the first pointer:

```
l = let
    p1 = malloc (Node Str)
    p1^ = "Lorem ipsum", Null, init (Node "Etiam ultricies" Null Null)
in p1
```

By assigning a pointer in the dereferenced context of another, both get entangled. Internally, metadata about the relationship between the pointers `p1 ^> p2` is then stored and updated with each memory action such as `free`. If it turns out that an attempt was made to release `p1` before `p2` was freed, an error can be issued. However, to provide such protective mechanisms, the information must be stored centrally.

One possible solution would be for the smart main pointer `p1` on the stack to have an additional attribute that counts the number of subpointers. Only when this counter is at 0 can the main pointer be released without an error message. In this way, the programmer is forced to write proper code, while the compiler can easily optimize away those safety guarantees in "proven code" if wanted.

Example: Linked List

```
type Node t is -data t, -next ^Node ~ Null

# (:=) : (Var *a) a -> ()
# (:=) : (Dereferenced ^a b) b -> ()

extend node data = let
    # traverse : Sub ^Node t -> Sub ^Node t
    traverse Null Null = Node data
    traverse first Null =
        if first^.next = Null then
            Node::first^.next := init (Node data) >> first, first^.next
        else
            traverse first first^.next
    traverse first current =
        if current^.next = Null then
            Node::current^.next := init (Node data) >> first, current^.next
        else
            traverse first current^.next
in
    traverse node Null

# extend : Valid (^Node t) -> Sub (^Node t) where Runtime
```

Imperative and also more concise implementation:

```
extend node data = do
```

    Copy memory address from `node` to `current` and borrow all rights until `node` is reused:

```
    current <= node
    while current^.next /= Null loop current := current^.next
    current^.next := init (Node data)
    last := current^.next
```

    Once `node` is used, `current` is no longer available as it may still be the original pointer:

```
    Subpointer node last

free-nodes first =
    current, next := first, Null >> while current^.next /= Null loop
        next := current^.next >> free current >> current := next
# free-nodes : ^Node -> ()
```

The individual actions can also stand one below the other without flow operators, since `loop` links them together like `do`.

Pointer Abstraction

For daily work, it is rather inconvenient to handle pointers directly all the time. Instead of using garbage collection, Kalkyl relies on a mechanism comparable to RAII, which offers the advantage that the inefficiency and other problems of GCs are avoided, while the programmer can still use dynamic data structures comfortably like in modern scripting languages without having to worry about correct memory management. This is possible if a pointer type representing a dynamic data structure has implemented the core concept `Disposable` providing methods for construction and deconstruction:

```
alias Disposable t | data is Allocating t data, Releasing t

new : data -> *t where Disposable t, Runtime

dispose : *t -> () where Disposable t
```

The concept `Disposable` can only be implemented by abstracted pointer types:

```
type LinkedList t is ^Node t # LinkedList : Type -> Reference

preinstall module Disposable (LinkedList t) (Array t) has
    new arr = do
        if not arr&empty? then
            first := init (Node arr[0])
            current <= first
            for data in arr[1:] loop
                current^.next := init (Node data)
                current := current^.next
            first
        else
            Null

    dispose = free-nodes

x = () -> LinkedList::new {58, 3, 0, 49, 1}
```

The compiler automatically calls the destructor of a disposable object when its scope ends. If the uniqueness is lost, a heap object created with `new` cannot be destroyed prematurely using `dispose`, but gets released earliest after the function has finished, provided it is not returned.

At this point it should be emphasized again that objects created with `new` can only be subsequently manipulated if their uniqueness is preserved.

Static Pointer Abstractions

While arrays can be constructed as a mere block in the data segment, this is much more complicated for linked lists and other data structures. However, referentially transparent code can be executed at compile time to tell the compiler exactly how such data structures are to be constructed statically, which allows much more performant programs to be developed, since only mutable structures actually need to be created at runtime as heap objects.

It would be conceivable to provide a separate concept for this, whose method is called at compile time to construct the data structure in question:

```
abstract concept Immediate t data where Disposable t data has
    construct : data -> t
```

Another conceivable approach would be for the compiler to automatically derive the corresponding construction based on the implementation of the Disposable concept by translating the request functions `malloc`, `init` and `remalloc` alternatively, so that instead of pointers to heap objects, pointers to objects created in the data segment are returned.

Summary of Rules

Rules to maintain pointer linearity:

1. All rules for unique objects also apply to pointers, with the only difference being that pointers must be used exactly once.
2. A pointer can be dereferenced exactly once without being considered an application to assign a value; for tuples and arrays, either each element individually or as a whole. The same principle continues with dereferenced pointers to pointers.
3. A dereferenced pointer can only be assigned a memory address of another pointer in the form of pointer entanglement.
4. Pointers can be dereferenced multiple times to only read values (to the right of the definition character `=`), unless they are memory addresses.

Protective Mechanisms

In secure mode, certain protection mechanisms must be implemented automatically when dealing with pointers:

- After calling `malloc`, the pointer must be checked for non-null before the first dereferencing to assign a value takes place.
- Before the first read access, it must be guaranteed whether a value has already been assigned. For tuples and arrays, this check is ultimately required for each element.
- It must be checked at compile time – made possible by linearity – that every pointer acquired by `malloc` is sooner or later finally consumed by calling `free`.
- A pointer whose memory address has been assigned as a value to another pointer must be entangled with it. Such entanglement means that the pointer containing the address of the another pointer – whether as a single value or element of a sequence of values – can only be released if the captured allocated memory of the stored address has been freed first.
- Since `malloc` and `remalloc` can return null pointers, functions that operate on such pointers must take this case into account – through conditionals or pattern matching, otherwise the compiler automatically infers `^t \ Null` as a precondition for parameters and postcondition for the return value, which eliminates the need for subsequent checks.

Instead of reflecting raw pointers at type level, the optimizing removal of all guarantees should be activated by a compiler flag in order not to unnecessarily complicate the type system. This seems much more sensible, because a test version with additional protection code that returns error messages can be created from the same code base as an extremely optimized release without any overhead if more performance is required through unproteced raw pointers.

Implementation Solutions

5.3 Overall Picture

On the surface, Kalkyl does not distinguish between values and references in order to keep the type system simple but also the language itself. In any case, such a distinction is superfluous from the programmer's point of view because the compiler is much better at deciding when it makes sense to pass an argument as a value or a reference. In particular, it seems inefficient to reference values that are not larger than a memory address, so that the value itself could just as well be copied. Here, references are usually only used to change existing variables in a procedural programming manner:

```
void increment(int *num) {
    (*num)++;
}

int main() {
    int value = 10;
    printf("Before increment: %d\n", value);
    increment(&value);
    printf("After increment: %d\n", value);
    return 0;
}
```

> For the sake of simplicity, references are understood here as restricted pointers that are automatically dereferenced. This C code is simply intended to illustrate that you can simulate this with pointers and some self-discipline.

However, Kalkyl is designed to have only pure functions; therefore, the corresponding C code must resort to value parameters:

```
int increment(int num) {
    return num + 1;
}

int main() {
    int value = 10;
    printf("Before increment: %d\n", value);
    value = increment(value);
    printf("After increment: %d\n", value);
    return 0;
}
```

Furthermore, an equivalent program in Kalkyl would introduce another variable instead of overwriting the existing one to exclude unwanted side effects:

```
increment num = num + 1

program =
    let value = 10 in echo f"Before increment: $value\n" >>
    let value2 = increment value in echo f"After increment: $value2\n"
```

Although the program may seem less efficient at first glance, the referential transparency of Kalkyl allows the compiler to radically simplify the Interlinked computation steps, so that in the end a C program can be generated as output, which is comparable to the procedural variant described first.

As an alternative to let expressions, there are – as already described in previous sections – flow variables, which can make the code appear more imperative (and at the same time ease optimizations for the compiler), although here the evaluation order and syntax only give the impression that variables are being assigned new values:

```
program =
    value := 10 >> echo f"Before increment: $value\n" >>
    value := increment value >> echo f"After increment: $value\n"
```

> In contrast to let expressions, flow variables are not capable of self-calling (recursion), so that if a variable with the same name appears to the right of the operator `:=`, it must in any case be an already defined variable that gets overshadowed in subsequent evaluations.

The order of evaluation is determined by the flow operator, which is left-associative, so that a program flow can be modeled similarly to imperative languages, but which is made up of expressions. Due to this inherent sequence and the principle of overshadowing, a flow variable, although unique, can be used once in each calculation step – i.e. operands of the flow operator. The compiler should be able to reuse the memory of a previously overshadowed flow variable of the same type in the output code.

`Compiler.refer : t -> t`

`Compiler.copy : t -> t`

6 Foreign Function Interface

Interoperability with other programming languages is achieved using the C calling convention as the greatest common denominator. By directly supporting machine-level data types, pointers and manual memory management, but also through the initial implementation with C as an intermediate language, Kalkyl is inherently suitable for interacting directly with C programs. As a result, marshalling between data representations of C and Kalkyl types is unnecessary.

This chapter breaks down all the details of how code can be used reciprocally between C and Kalkyl. For this purpose, the rules for translating Kalkyl to C are explained.

C Data Types

Most of Kalkyl's basic data types correspond directly to their C counterparts. For cases where this does not apply, supplementary types in the C namespace offer direct support, whose values the compiler can handle directly:

Integer Types

C		**Kalkyl**	
bool		Bool	
unsigned char	(signed) char	Unsafe.Nat8	Unsafe.Int8
unsigned short	(signed) short	Unsafe.NatShort	Unsafe.IntShort
unsigned int	(signed) int	Unsafe.NatBasic	Unsafe.IntBasic
unsigned long	(signed) long	Unsafe.NatLong	Unsafe.IntLong
unsigned long long	(signed) long long	Unsafe.Nat64	Unsafe.Int64
uintptr_t	ptrdiff_t	Unsafe.Ptr	Unsafe.PtrDiff
size_t	ssize_t	Unsafe.Nat	Unsafe.Int
uint8_t	int8_t	Unsafe.Nat8	Unsafe.Int8
uint16_t	int16_t	Unsafe.Nat16	Unsafe.Int8
uint32_t	int32_t	Unsafe.Nat32	Unsafe.Int8
uint64_t	int64_t	Unsafe.Nat64	Unsafe.Int8

Other Types

C			Kalkyl		
void	void *		()	^t	
float	double	long double	Float32	Float64	Float80
char	char16_t	char32_t	Unsafe.Char	Unsafe.Char16	Unsafe.Char32

7 Reference Compiler

The reference compiler “Zuse” combines project management and program translation into one efficient tool. A main objective here is to minimize the need for manual settings through binding conventions.

7.1 Translation Model

7.2 Project Structure

Zuse's interface is designed with a common project structure in mind to provide simplified handling:

- `assets/` or `resources/`
- `builds/`
- `docs/`
- `include/`
- `output/`
  - `c/`
  - `obj/`
  - `…`
- `plans/`
  - `libexample/`
    - `example.k`
    - `prelude.k`
    - `…`
  - `example/`
    - `example.k`
    - `prelude.k`
    - `…`
  - `example_plugin-name/`
    - `plugin-name.k`
    - `prelude.k`
    - `…`
- `settings`
- `tests`
  - `a.k`
  - `…`
- `zuse.db`

The subdirectories listed here are just those assumed by default.

Configuration of the compiler and all project meta data are stored in the local project data base `zuse.db`.

The `assets` – or `resources` – directory contains files that do not represent program code, but are yet integrated into the program such as images.

The `output` directory is automatically generated when needed during compilation and contains all intermediate files. However, finished compilations, whether as executable files, plugins or program libraries, are stored in the `builds` folder. Like `output`, the `include` directory is created automatically when header files are generated from a library project.

Programs and data for testing should be stored under `tests`. Equally self-explanatory is `docs` for project documentation.

The actual program code of the project is organized in subfolders under `plans`, each of which represents a sub-project consisting of a main component. Main components can be libraries with the prefix `lib`, applications of any kind, or plugins preceded by the name of the related application and a separating underscore.

Using this mandatory naming convention, the compiler is able to automatically infer the purpose to name and handle builds accordingly. Zuse therefore supports any number of sub-projects, which are closely related to each other and share resources. This offers the advantage that instead of a large monolithic library, several smaller components can be developed together but still separately reused.

File-level Insignificance

The plan name must be in lowercase and match the name of its main component subjected to the following rules:

- The compiler does not distinguish between upper- / lowercase,
- and does not interpret hyphens as significant.

Therefore, if a plan is called `foo-baz`, only `Foobaz` with or without hyphens (`Foo-baz`) and with or without capital letters (`Foo-Baz`) in between can be chosen as the corresponding component name.

These insignificance rules are designed to ensure a systematic mapping between capitalized component identifiers and application or library names, where lowercase is the norm. Furthermore, there is also the intention to exclude libraries with almost identical names in the same repository, which only differ from each other by the presence of hyphens or capitalization.

Underscores `_`, on the other hand, are only allowed to provide additional information that the compiler expects, for example the preceding naming of the application for which a plan is intended as its plugin.

Repositories and Libraries

Components obtained from repositories are referred to as libraries:

```
repository GitHub.Thyringer is /github.com/thyringer

section GitHub.Thyringer

use GitHub.Thyringer:/Kabel.1.4
```

> Since `github.com` is of particular importance, the namespace `GitHub` should be reserved and the compiler be able to automatically resolve capitalized account names, so that the explicit definition as a repository is unnecessary.

To distinguish such external components from local ones, they are referenced only together with the name of their repository separated by the special symbol `:/` to avoid confusion. In this way, repository names don’t collide with components.

Furthermore, the library version can be specified using dot notation. Consequently, it is not a problem to use different versions of the same external component in parallel, as long as they are imported under different namespaces.

The alias `Public` already exists for referencing to Kalkyl's standard public repository:

```
repository Public is /repo.kalkyl.dev
use Public:/Base.1.6
```

> With explicit repositories, a decentralized package system is enabled for Kalkyl.

Project-wide Dependency Management

Main components offer the special feature that their library imports are visible to all subordinate components, which allows dependencies to be defined uniformly for an entire subproject. This approach eliminates the need for separate configuration files for dependency management.

using the form `MAJOR.MINOR.REVISION.BUILDNUMBER`, where:

- MAJOR is a major release (usually many new features or changes to the UI or underlying OS)
- MINOR is a minor release (perhaps some new features) on a previous major release
- REVISION is usually a fix for a previous minor release (no new functionality)
- BUILDNUMBER is incremented for each latest build of a revision.

7.3 Usage

The command line interface of the reference compiler corresponds to the conventions:

`zuse ( option | subcommand [arg] [suboption [arg]]ₙ )`

Furthermore, single-letter shorthand notations are also provided for the most important subcommands. All possible combinations of short options are explicitly broken down since only very few make sense.

All relevant data entered is saved in a local project database.

Options that expect Boolean values **`%bool`** can be operated with various values, including all cases:

`1` | **`t`** | **`true`** | **`y`** | **`yes`** | **`0`** | **`f`** | **`false`** | **`n`** | **`no`**

Options that expect an argument are separated from each other by either a space or an equal sign. For reasons of simplicity, this is not explicitly mentioned in the next sections.

The compiler's interface is kept small, not least because project management is mainly done in the interactive environment with Kalkyl itself, instead of offering a separate subcommand with additional options for everything. This offers two fundamental advantages: firstly, the interface is easier to maintain, and secondly, it is easier for programmers to use.

General Options

All options, including their short variants, can also be entered without hyphens like subcommands.

`-?` / `-h` / `--help`

Get an overview of all commands

`-V` / `-v` / `--version`

Show compiler version

Project Creation

The general subcommand to create a new project in the current working directory is

```
new title [--indent %int:[1, 8]]
```

> The title describes the project's own name or brand in any spelling with special characters, and is usually capitalized.

and can be supplemented with one or more of the following options:

	Option	Args	Naming	Explanation: Creates a/an…
`a`	`app`	c	<name>	application
`l`	`lib`	c	`lib`<name>	program library
`p`	`plugin`	c app	<app>_<name>	extension of an existing program
`t`	`test`	c [x]	[<x>_]<name>	component to test other components

The first argument specifies the name of the main component and must always be capitalized according to the rules for component identifiers. The second argument specifies the main component name of the subproject to extend or test.

According to the binding conventions, Zuse creates folders in the project subdirectory `components` with respective files of their main components.

> Basically, there is no need to create a new project using the command `new`, since a project directory can just as easily be created manually according to the expected structure. Zuse automatically creates missing files and folders.

Additional subprojects can be created manually under `./components`.

If no further information is provided, Zuse assumes a project with code indented by tabs (value → 1). All values greater than 1 specify the number of spaces that are counted as one indentation.

Development and Distribution

All commands require a directory **`components`** with at least one component which represents the program, a public library interface or plugin implementation.

`n`	`new`	`[C.]`$_n$`C`$_n$	Create one or more new components (upper case).
`n`	`new`	p	Create one or more new plans (lower case)
`l`	`lint`	`[p]`	Check code for errors without generating an executable.
`t`	`translate`	`[p]` `[--to …]`	Translate to C `--to C` (default for `x`) or something else like.
`b`	`build`	`[p]`	Compile and link to an executable or library. By default, Zuse translates a module with public functionality into a program library with a C interface. The designation of an executable, shared library or plugin depends on the folder name in the component directory
`r`	`run`	`[p]`	Build if necessary and run with the given arguments.
	`release`	`[p]`	Like `build` but applying all specified optimizations.
`a`	`archive`	`[p]`	Build shared lib and install it at all destinations.
`s`	`share`	`[p]`	Build shared lib and install it at all destinations.
`k`	`kalkylize`	`[p]`	For each header file, create a component with the same name that provides the corresponding C functions.
	`clean`		Delete all generated intermediate files and executables. Usually this command is rarely required, as Zuse does this automatically when recompiling modified modules.
	`doc`	`[c.]`$_n$`c`$_n$ `…`	Generate HTML `--html` and/or MD `--md` documents for each component and output them under `docs/`~.
`d`	`distribute`	`[p]`	Publish the project(s) to all repositories flagged as uploading.
`p`	`distribute`	`[p]`	Publish the project(s) to all repositories flagged as uploading.

Instead of controlling the compilation and other processes through numerous subtle command line options, such settings should be made using the program shell.

```
zuse build libkabel.a

example.k → example.g → example.c → example.o → example
```

	Linux / FreeBSD	macOS	Windows
Application	`<name>` (no prefix)	`<name>.app`	`<name>.exe`
Plugin	`<name>.so`	`<name>.bundle`	`<name>.dll`
Dynamic Lib	`lib<name>.so.<vers>`	`lib<name>.dylib`	`<name>.dll`
Static Lib	`lib<name>.a`	`lib<name>.a`	`<name>.lib`

Language Shell

The language shell is called by the compiler's program name `zuse` without a subcommand. Within this interactive environment, entered expressions get evaluated and displayed directly, provided that an matching installation for `Displayable` exists.

```
user@pc:~$ zuse
Kalkyl 0.1
>
```

There are a handful of single-letter named built-in functions available within the language shell, which always overshadow all other names of loaded or imported modules (single-letter names outside of functions should be avoided anyway, so overshadowing is pretty unlikely):

`t / type : expr -> Type`

Returns the type of an expression.

`i / Zuse.info : Kalkyl.ID -> Text`

Returns detailed information about an identifier. For example, all related data constructors are broken down for a given type constructor, or all members of a module's public interface, respectively concepts in general as well.

`l / Zuse.load : -module Path -context *IO -> *IO`

Load a module. If a module is already loaded, it will be replaced. Declarations in the REPL remain unaffected.

`Zuse.reload : *IO -> *IO`

Reload the current module.

`Zuse.unload : *IO -> *IO`

Unload the current module.

`Zuse.clear : *IO -> *IO`

Clean REPL of all local name bindings but keep current module loaded.

`q / Zuse.quit : *IO -> *IO`

Exit the language shell.

—

Functions of type `*IO -> *IO` are executed in the language immediately after input.

Project Configuration

Zuse expects a data module `zuse.k` in the root directory of the project, which provides additional information for each subproject:

```
module Zuse.Project

title = "Awesome Example Collection"

description = "Associated libraries that are incredibly impressive."

subprojects = Set
    Subproject "libexample-a" 1.2.3 Unlicense Set
    Subproject "libexample-b" 4.5.6 Unlicense Set
        Person
            -email "max.mustermann@example.de"
            -name ("Max", "Mustermann")
            -from 2024
            -object "B.f"

maintainers = Set
    Organization "info@awesome.com" "Awesome Corporation" 2024

website = "awesome-example-collection.org"
```

~~In-file Configuration~~

~~As an alternative to the project-wide configuration, the kind of indent – and other compiler information – can also be specified for each component individually, directly after the shebang and before the first declarations:~~

```
#! /usr/bin/env kalkyl
#! indent 4
#! unsecured
# …

component …
```

~~The character string `#!` is used to distinguish settings from other comments. For simplicity, the compiler is expected to look for such information only in the first lines.~~

To simplify parsing, no compiler options should be included within a file.

Testing

The compiler supports two basic methods of tests:

	Component Test	**Integration Test**
location	directory of the respective plan	project directory `tests`
reference point	related to a single component	related to the public library API
visibility	access to non-public members	no access to internals
command	`zuse test Example`	`zuse run Example [args]`

Component tests are written directly in the component in question, which provides access to all internals:

```
component Example

section Test has
    a = expra : {lit}          # expected result given by a refinement
    b = {expr1, expr2, …} &map f # b : Array …
    p = program x1             # p : (Array String) *IO -> *IO
    p = program x2
with
   x1 = {string1, string2, …}
   x2 = …
```

All definitions in the namespace `Test` of a component get evaluated in sequence from top to bottom when testing – with `zuse test Example`. Definitions under `Test` may only have one parameter of type `*IO`. To test a function with multiple values at the same time, a mapping of a list of tuples is recommended, with each providing arguments for all parameters. In addition, refinements provide a tool to precisely determine the expected return value, so that Kalkyl itself already offers everything to formulate extensive component tests.

In the normal compilation process, test sections get simply discarded, just like comments.

To test internal components, they must be referenced absolutely, for example

```
zuse test Example.B
```

for the subcomponent `./plans/libexample/b.k`.

Integration tests, on the other hand, are written in separate files in the project directory `tests`, whose program implementations get built and executed directly using Zuse's run subcommand. The test components can be named as wanted, especially to avoid confusion with plans.

Debugging

7.4 Plugins

~~By default, Zuse first translates Kalkyl code into C and then just calls an external C compiler like GCC or Clang. However, much more efficiency could be reached by compiling Kalkyl directly for specific platforms, which is why a flexible plugin system gives third parties the opportunity to extend Zuse. For this purpose, a distinction is made between support for a real or virtual instruction set:~~

~~**Target Language**~~	~~**File Naming**~~	~~**Examples**~~
~~machine code~~ ~~inclusive for VM~~	~~`zuse_`<name>`.so`~~	~~`zuse_x86-64.so`~~ ~~`zuse_wasm.so`~~

~~The respective options `-arch` can be used to compile for a specific architecture/platform:~~

~~`zuse b --arch wasm`~~

~~This causes Zuse to search for a plugin with the passed value as name, first globally as a system library, and then locally in the compiler's configuration directory.~~

Instead of a plugin system, Zuse's development is focused on C, WASM and, later, BC (LLVM bitcode) as target outputs. JS could also be considered as an alternative to WASM.

8 Base Library

Kalkyl's library, which provides all the basic functionalities of the language, is called "Base". The base library is deliberately kept flat so that the vast majority of identifiers are directly available and do not have to be laboriously qualified through numerous sub-namespaces. This chapter lists the specifications of all base content. The implementation, however, is not the subject of this report. Nevertheless, there are references to possible solutions in some places.

8.1 Base Types

Fundamental Sum Types and Units

```
type Bool has True | False

type None has None
(? _) t = t + None ~ None

metric B : 1
```

Integer Types

```
## Type constructor for precise intervals of valid integer values, where the
compiler automatically selects the best implementation.
type Int min max has Int :
    -val {val : i where Integer i | val in [2**127, 2**127 - 1]}
    -min i ~ $min-bound # or: -min {min : i} ~ $arch-min
    -max i ~ $max-bound
    -> val
```

For min- and max-bound, the compiler must provide a module depending on the local architecture.

```
alias Int8 is Int -min -1 * 2**7 -max 2**7 - 1
alias Int16 is Int -min -1 * 2**15 -max 2**15 - 1
alias Int32 is Int -min -1 * 2**31 -max 2**31 - 1
alias Int64 is Int -min -1 * 2**63 -max 2**63 - 1
alias Int128 is Int -min -1 * 2**127 -max 2**127 - 1

section Num has
    ## Integer type whose values can take on arbitrarily large values.
    type Int is {val : i where Integer i}
```

Natural Types

```
type Nat min max has Nat :
    -val {val : i where Integer i | val in [0, 2**128 - 1]}
    -min i ~ 0
    -max i ~ $max-bound

type Count is Nat -min 1

alias Nat8 is Nat -max 2**8 - 1
alias Nat16 is Nat -max 2**16 - 1
alias Nat32 is Nat -max 2**32 - 1
alias Nat64 is Nat -max 2**64 - 1
alias Nat128 is Nat -max 2**128 - 1

type Byte adopts N8

type UTF8Code is {n : Nat32 | n in [0, 4103061439]}

section Num has
    ## Integer type whose values can take on arbitrarily large values.
    type Nat is {val : i where Integer i | val >= 0}
```

Decimal and Floating-Point Types

```
type Float16 is {val : r where Rational r | val in [min, max]}
type Float32 is {val : r where Rational r | val in [min, max]}
type Float64 is {val : r where Rational r | val in [min, max]}
type Float80 is {val : r where Rational r | val in [min, max]}

type Num min max has Num :
    -val {val : r where Rational r | val in [min, max]}
    -min r ~ -Inf
    -max r ~ Inf
    -> val

type Dec prec scale has Dec :
    -val {val : r where Rational r |
        val&prec in [1, prec], val&scale in [0, scale]}
    -prec Num.Int
    -scale {n | n < prec}
    -> val
```

String Implementation

Strings in Kalkyl must be implemented binary safe by storing the length before the beginning of the text instead of using a null terminator:

```
abstract type Internal.String is -length {n : Nat}, -data Byte[n]

type String has
    String : s -> Internal.String where Castable s String
    String s = cast s

# String::String : s -> String where Castable s String
The castable parameter is only possible because `Internal.String` acts as a kind of bootstrapping type.

preinstall module Memory.Allocating String String has
    malloc str = let
        ptr = malloc (Nat&need + str.length) : ^String
        ptr^ = str
        Compiler always copies non-unique objects, even if a reference was passed for more efficiency.
    in ptr

x () = new "Hallo, Welt" # x : () -> *String where Runtime
```

No specific implementation is required for Releasing, since `String`, as a tuple object, does not have any fields with additional pointers that would have to be released first.

Only Magic Needed

The compiler must automatically convert string literals into objects of the type `Internal.String`, which includes counting the characters. This means that the string lengths are already known at runtime, which makes their processing significantly more efficient.

In addition, the compiler must automatically recognize from the return type of the only data constructor `String::String` that `Internal.String` and `String` are identical, so that `String` can already be used as a synonym for `Internal.String` in constraints.

Other Textual Data Types

```
type Display t adopts String
```

8.2 Concepts

Defaults and Conversions

```
concept Default t has
    default : t

@Default t val = module Default t is val


concept Castable from to has
    cast : from -> to

alias Serializable t is Castable t (Serial t)

alias Displayable t is Castable t (Display t)

# echo : t *IO -> *IO where Displayable t

concept Coercible from to where Cast from to
```

Causes compiler to perform automatic type casts.

~~`concept Displayable t has`~~

~~Class of types whose values are convertible to human-readable representations, but without any guarantee that such converted objects can be recovered from their textual representations.~~

~~`display : t -> Text`~~

~~Returns human-readable text representation of an object.~~

```
concept Serializable t where
    Castable t (Serial t), Castable (Serial t) t
has
```

Types whose values are serializable.

```
    type Serial t has ~ t : Text deriving Coercible Text
```

Unspecific Instances

```
preinstall module Cast t t is x -> x
```

Boundaries

```
concept Bounded a has
    min, max : a
```

Properties of Sum Types

```
type Constructor t has Constructor d forall d : {d | Return d = t}

abstract concept ADT t has
```

Cannot be implemented directly; global modules are automatically created for each types.

```
    constructors : -t Type -> Array (Constructor t)
```

Array of all unapplied data constructors of a type in the order in which they were declared.

```
concept Enumerated t | count where Variant t, Comparable t has
    type EnumIndex t has
      EnumIndex (t + {i : Int2 | i >= -1*count, i < count}) : i

    pred, succ : t -> t

    variant : EnumIndex t -> t

    variants :
      -from EnumIndex t
      -then EnumIndex t ?
      -to   EnumIndex t ?
      -> Array t n
```

Logical Operations

```
concept Boolean b~Bool has
   (not) : b -> b
   (and), (nand) : b b -> b
   (or), (nor), (xor), (xnor) : b b -> b


concept Bitwise i where Boolean i has
   shiftl, shiftr : i i -> i


(shl) = shiftl
(shr) = shiftr

(is?) : t {t : Type} -> Bool
(as?) : t Type -> Bool
```

Comparison Operations

```
concept Equal t logical/l~Bool has
   equal? : t t -> l

unequal? a b = not (equal? a b)

(==) = equal?
(/=) = unequal?

concept Comparable t logical/l~Bool where Equal t l has
   greater?, greater-or-equal?, less?, less-or-equal? : t t -> l

(>) = greater?
(<) = less?
(>=) = greater-or-equal?
(=<) = less-or-equal?
```

General Properties

```
concept Adding a b~a | c~b has
    Intended for commutative operations only.
    #! infixl 6
    (+) : a b -> c

concept Multiplying a b~a | c~b has
    (*) : a b -> c

concept Potentiating a b~a | c~b has
    (**) : a b -> c

concept Reducing a b~a | c~b has
    (-) : a b -> c

concept Dividing a b~a | c~b has
    (/) : a b -> c

concept Extracting a b~a | c~b has
    (//) : a b -> c
```

Increment and Decrement

```
concept Incrementing a | b~a def c~a has
Allows use of the increment operator.
    increment : a b~def -> c

concept Decrementing a | b~a def c~a has
Allows use of the decrement operator.
    decrement : a b~def -> c
```

Concatenation

```
concept Concating a b~a | c:{a, b}~a has # or: Concat(enat)ing
```

Corresponds to semigroups; only intended for non-commutative operations.

```
    (<>) : a b -> c

concat : a .. a + b -> c where Concating a {a, b} c


concept Containing t where Concating t has
```

Corresponds to monoids.

```
    empty : t
    contains? : t t -> Bool


concept Inverting t with Concatenating t has
```

Corresponds to groups.

```
    invert :: t -> t
```

~~class Joining a b~a | c:{a, b}~a where Concating a b c has~~
~~(<>>) : a b -> c~~
~~(<<>) : a b -> c~~
~~(<<>>) : a b -> c~~

Unspecific Instances

```
use module Concating (b -> c) (a -> b) (a -> c) has
    (<>) g f = \x -> g (f x)
```

Properties of Numeric Types

```
concept Numeric n where Adding n, Subtracting n has
   Operations useful on numbers and other numerical objects such as vectors and matrices.
   add : -augend n -addend n -> n
   multiply : -multiplier n -multiplicand n -> n
   neg : n -> n
   abs : n -> n


concept Integer i | exp where
   Numeric i, Multiplying i, Potentiating i exp, Dividing i, Comparable i
has
   Includes all operations applying without restriction to the whole numbers.
   pow : -base i -exponent exp -> i
   mod/modulo : -dividend i -divisor i -> i
   rem/remainder : -dividend i -divisor i -> i


install module Integer Int32 {i : Int32 | i >= 0} has
    pow base exp = do
        result := 1
        loop
            if (exp and 1)
                result := result * base
            exp := exp &shr 1
            if exp = 0 then
                break
            else
                base := base * base
        result
    …


concept Rational r i given r => i where
   Integer r, Castable Literal.Rational r
has
   Includes all operations being applicable to the rational numbers without any restrictions.
   ceil : r -> r
   floor : r -> r
   trunc/truncate : r -> i
   recip/reciprocal : r -> r
   round : -num r -decimals Nat -> r
      Symmetric rounding.
```

```
concept Real r where Rational r has
    Functions incompleted for the domain of real numbers and possibly returning irrational numbers.
    nrt : -radicand r -index r -> r
    nrt rnd idx = pow rnd (1 / idx)
    #
    sqrt : -radicand r -> r
    sqrt rnd = pow rnd (1 / 2)
    #
    crt : -radicand r -> r
    crt rnd = pow rnd (1 / 3)
      If necessary, can be implemented separately for reasons of optimization.
    lb : -power r -> r
      Binary logarithms.
    lg : -power r -> r
      Decadic logarithms.
    ln : -power r -> r
      Natural logarithms.
    log : -power r -> -base r -> r


concept Trigonometric r where Real r has
    sin, cos, tan, asin, acos, atan,
      sinh, cosh, tanh, asinh, acosh, atanh : r -> r


pi / π
    = [3.14159 26535 89793 23846] : r where Real r
e / euler
    = [2.71828 18284 59045 23536] : r where Real r
golden-ratio / phi / ϕ
    = [1.61803 39887 49894 84820] : r where Real r

sqrt2
    = [1.41421 35623 73095] : r where Real r
sqrt3
    = [1.73205 08075 68877] : r where Real r
sqrt5
    = [2.23606 79774 99789] : r where Real r
```

Sequencing

```
concept TransmittingLeft a b | c~a has
    (<*) : a b -> c

concept TransmittingRight a b | c~b has
    (*>) : a b -> c

concept FlowingLeft a b | c~a has
    (<<) : a b -> c

concept FlowingRight a b | c~b has
    (>>) : a b -> c

concept Do first second has
    do : first -> second
```

Unspecific Instances

```
preinstall module FlowingRight (x -> a) (a -> y) (x -> y) has
   (>>) f g x = g (f x)


preinstall module Doing x x by (x -> x)

preinstall module Doing a (b -> c) where FlowingRight a b c by (>>)

#[
   x := 7 >> echo x ≡ wrap 7 >> \ x -> echo x

   x <= get-line >> echo x ≡ get-line >> \ x -> echo x
]#
```

Assignments

`(:=) : (Var a) a -> f () where Active f`

Values of type 'Var' stand for variable names stated ad hoc.

`(<=) : (Var a) (f a) -> f () where Active f`

Assigns a value resulting from an action.

`(+=) : (Var a) b -> f () where Active f, Adding a b a`

Values of type 'Var' stand for variable names stated ad hoc.

`(-=) : (Var a) b -> f () where Active f, Reducing a b a`

Values of type 'Var' stand for variable names stated ad hoc.

```
#[
   x := 7 >> echo x ≡ wrap 7 >> \ x -> echo x

   x <= get-line >> echo x ≡ get-line >> \ x -> echo x
]#
```

Functors

```
concept Functor f forall a, b where
    TransmittingRight (f a) b (f b), TransmittingLeft a (f b) (f a)
has
    map : -fn (a -> b) -data f a -> (f b)
```
Applies a function to values of a container (functor).

```
concept Applicative f forall a, b where
    Functor f, FlowingRight (f a) (f b), FlowingLeft (f a) (f b)
has
    wrap : a -> (f a)
    multimap : -fn f (a -> b) -data f a -> (f b)
```
Applies a wrapped function (fn) or series of mappings to values in the same context (functor).

```
#apply : -f (a -> b) -val a -> f b where Applicative f
#apply = \f arg => wrap (f arg)
```

```
concept Alternative f forall elem where Neutral (f elem) has
```
Container types only wrapping up referentially transparent values.
```
    some : (f elem) -> (f (Array elem))
```
Keep trying until it succeeds at least once, and then keep doing it until it fails.
```
    many : (f elem) -> (f (Array elem))
```
Do this as many times as you can until failure.

```
concept Declarative f where Applicative f with
```
Implementation allows use of the jump command `return`.
```
    unwrap : (f a) -> a
```

```
concept Monadic f forall a, b where
    Applicative f, FlowingRight (f a) (a -> f b) (f b)
has
```
Functions linking contextual values.
```
    join : (f (f a)) -> (f a)
    join x = x >> \
      # '\' without parameters is just shorthand for identity function.
```

```
use module Concating (a -> f b) (b -> f c) (a -> f c) forall
    a, b, f where Active f
has
    (<>) = \ f1 f2 -> (\a -> f1 a >> f2)
```
Chains two functions with encapsulated results (→ Kleisli operator).

Bifunctors

```
concept Bifunctor f has
    bimap : -f1 (a -> b) -f2 (c -> d) -context f a c -> (f b d)
    Bimap.first : (a -> b) (f a c) -> (f b c)
    Bimap.second : (b -> c) (f a b) -> (f a c)
```

Properties of Composite Types

```
concept Sliceable r | elem has
```

Instantiation of this concept allows access to elements by index and slice syntax.

```
    item : -range r -key ObjIndex -> elem


concept Countable c elem has
    count : c elem -> Nat


concept Structured s has
    (in?) : (s elem) elem -> Bool


concept Collecting coll | elem has
    collect : elem elem -> coll


concept SetLike s where Adding s, Reducing s has
    subset?, proper-subset?, superset?, proper-superset?
      : -subset c elem -superset c elem -> Bool

    union, intersection : c -> c ..


concept Mappable c has
```

Converts a data structure that maps values to each other into a function.

```
     mapping : -collection c x y -otherwise y -> (x -> y)
```

Folding and Traversing

```
concept Foldable t forall elem where Concatenating elem has

   fold : t elem -> elem

   fold-map : (a -> elem) -> t a -> elem

   foldl :
     (a a -> a) (t a) -> a,
     (a b -> a) a (t b) -> a

   foldr :
     (a a -> a) (t a) -> a,
     (a b -> b) b (t a) -> b

   Data.foldr : (a b -> b) b (t a) -> b

   Data.foldl : (b a -> b) b (t a) -> b

   null : (t a) -> Bool

   length : (t a) -> ObjSize

   elem : a (t a) -> Bool where Equal a

   maximum : (t a) -> a where Comparable a

   minimum : (t a) -> a where Comparable a


concept Traversable t forall f, m where
   Functor t, Foldable t, Applicative f, Monad m
has
  traverse : (a -> f b) (t a) -> f (t b)

  sequenceA : t (f a) -> f (t a)

  mapM : (a -> m b) (t a) -> m (t b)

  sequence : t (m a) -> m (t a)
```

These concepts are only adopted 1:1 from Haskell and still require major revision.

Variadic Functions

```
concept VarargLinkage a b~a c~b given a b => c has
   link : a b -> c

concept Variadic param result has
   variadic : param -> result where VarargLinkage param

preinstall module Variadic p p has
   variadic = function x -> x

preinstall module Variadic p (p -> result) where Variadic p result has
   variadic a b = variadic (link a b)
```

Example: Sum Function

```
@SummateVarargs = module VarargLinkage n where Numeric n is (+)

# summate : n -> r where Numeric n, Variadic n r
summate = variadic using SummateVarargs
```

Or if a module only contains very few methods:

```
summate = variadic using VarargLinkage::(+)
```

Compiler Interface

The compiler interface includes a helper library whose types generally describe software metadata and are therefore not limited to Kalkyl's project management.

```
component Software provides License, Version, Author

type Copyleft has
    AGPL | CPL | EPL | GPL | LGPL | MPL
    Custom Text

type Permissive has
    Apache | BSD0 | BSD2 | BSD3 | BSD4 | CC0 | ISC | MIT | Unlicense
    Custom Text

type License has
    Copyleft::* | Permissive::* | Proprietary

type Version is -major Int 0 9999, -minor Int 0 9999, -patch Int 0 9999

type Author x~() has
    Person
        # Natural person in the legal sense.
        -email Text -name (-first Text, -last Text) -from Nat2 -to ?Nat2 +x
    Organization
        # Juridical person in the legal sense.
        -email Text -name Text -from Nat2 -to ?Nat2 +x
```

In contrast, all Zuse-specific functionalities are in a separate component:

```
component Zuse

use Software

type Repo adopts Path Absolute



concept Project has
    title        : Text
    description  : ?Text
    maintainers  : Set Software.Author
    website      : ?Text
    dependencies : ?Map Repo (Map Plan Version)

type Subproject =
    -name Text,
    -version Software.Version,
    -licenses Set Software.License,
    -copyright Set Software.Author (-object ?Text)
```

A Limits

This appended chapter lists all the technical and purely arbitrary limits up to which Kalkyl is formally considered valid or the compiler should enable translation.

The limits are primarily intended to facilitate the implementation, to optimize the compilation process, or to just enforce a certain degree of reasonableness. Accordingly, it makes little sense to allow names that are much longer than 30 letters (without namespace), because no programmer would seriously want to use them (especially since the C standard only guarantees 31 significant characters for external identifiers). In addition, such lengths make it difficult to display errors properly in the terminal.

It should also be noted that Kalkyl was developed as a "project language", which has numerous implications: For instance, it doesn't really make sense to allow Kalkyl source files with more than 10,000 lines of code because this goes against all reason. Therefore, only a maximum of 9,999 lines of code are guaranteed to be compilable. In SQL files, however, which are often used as dumps, a line limit would be extremely inconvenient; but not so in Kalkyl.

Maximum number of allowed lines of code	9999
maximum allowed length of a line of code (not counting indentations)	199
Maximum indentation depth	9
Maximum possible number of characters per name	30
Maximum nesting level of namespaces and tuples	9
Maximum number of parameters and tuple values	16
~~Maximum possible length of string and number literals as well as doc comments~~	~~250~~
~~Maximum number of value constructors including inherited~~	~~250~~

If no maximum is specified, e.g. for the number of possible subcomponents or columns of a source file, "size" can always be assumed.

B Errors

Phase		Triggered by …
Warning	0	violations of conventions and indications of possible error
Building / **B**uild Err.	1	wrong compiler usage and IO failures during translation
Parsing / **S**yntax Err.	2	all errors that occur during parsing
Name Resolution Err.	3	missing definitions / unknown names, name overshadowing
Static **A**nalysis Err.	4	unknown type value, violated specification, wrong index, …
Running / **R**untime Err.	5	protections triggered at runtime, missing RAM, stack overflow

Runtime errors cannot be handled by the compiler, and are instead taken into account in advance using sum types or special values such as `Null` for pointers or `NaN` for numbers.

Error Code Structure and Display

```
> Syntax Error S.4 in <component>:<line>:L<level>+<column-from>:<column-to>
                  ┌─
  <line>  │   →  <code>
          └─     ^^^^^^

  <note>
```

Examples

The error codes are structured similarly to version numbers and can be converted into integers by maintaining a fixed number of digits for each component:

```
W.1    →  1
S.14   →  214
```

The phase is expressed by a single digit; individual cases by two digits and, if necessary, padded with zeros

Note

A current list of all error codes can be found in the source file `CompileError.ts`. The list here will only be updated when the error codes are considered stable enough.

Build Errors

Incorrect Usage of Zuse's CLI

1	unknown command
2	unknown option
3	wrong arguments
4	wrong argument for option #

Problems regarding project creation and management of project data

	could not create project #

Problems with folders and files during translation

31	source directory not found
32	no source files found
33	no main module found
34	source file # could not be read
35	output directory could not be created
36	output subdirectory # could not be created
37	unable to access output directory
38	unable to access output subdirectory #
39	could not create file #

Syntax Errors

```
Lexical Analysis
indentation with tabulator
illegal character
illegal word character




```

Name Resolution Errors

Static Analysis Errors

Runtime Errors

C Error Numbers

The C standard only defines the following three values of errno:

Macro	Meaning
`EDOM`	mathematics argument out of domain of function
`EILSEQ`	illegal byte sequence
`ERANGE`	result too large

C Solutions

Here are some brief explanations of how Kalkyl can be translated to C as a cross-platform intermediate language.

Additional Number Literals

Numbers are wrapped in structs if operations are performed at runtime that can fail, in order to store alternatives such as `NaN` for division by zero or the actual number:

```
enum NaN { NaN, FN }; // FN: Finite number

struct Int1_NaN {
    enum NaN state;
    Int1 number;
};
```

An alternative would be to reserve the highest value such as 255 for 1-byte integers to represent "NaN". The only problem here is that there are use cases where the entire value range is required, for example with RGB values.

D Machine Code Generation

This attached chapter summarizes important background information, ideas and initial solutions to translate Kalkyl directly into machine code.

LLVM as a Plugin?

One possibility would be to offer a plugin for Zuse that translates to LLVM IR, although at this point it is questionable whether the enormous complexity of the LLVM framework and actual performance gains justify the effort. The constant changes also mean that languages such as Swift and Rust regularly fork and modify LLVM, which is an absolute no-go for Zuse, if only because of the sheer extra work of maintaining foreign code written – to make matters worse – in C++. It must also be mentioned that all languages relying on LLVM suffer from very long compilation times. And LLVM IR itself is more tailored to C++, therefore not necessarily suitable for purely functional languages and their concepts.

Despite all these criticisms, it cannot be denied that LLVM IR is still the most sensible choice at present, since there are no real alternatives for precompiling into efficiently executable machine code if one wants to natively support as many architectures as possible without having to write a backend for each instruction set. For this reason, regardless of all objections, a separate LLVM plugin for Zuse seems to make sense nonetheless, which leaves the compiler itself independent and slim to avoid at least some of the disadvantages of LLVM.

How useful are backends for the various platforms?

It is not for nothing that the Go compiler uses its own backends precisely because of the problems with LLVM mentioned and for lack of serious alternatives. Unfortunately, the effort involved in developing own backends exceeds the resources of any small development team. Therefore, the first question to be answered is whether it even makes sense to support all conceivable platforms, especially since many architectures are already very outdated.

Furthermore, it is illusory to want to replace C in the embedded area, which is why a compiler emitting C can be more practical here. Mobile applications, on the other hand, are developed in the respective natively supported languages for Android and iOS, although here too the C output from Zuse seems more useful for writing cross-platform code. With these preliminary considerations, possible backends are reduced to the desktop and server area, which keeps the number of processor families somewhat manageable.

Selected platforms for high-performance Backends

Which architectures and operating systems are most important?

Common Operating Systems and Their Binary Formats

Unix: **Linux**, **FreeBS**, Solaris	ELF	Executable and Linkable Format
Windows	COFF / PE	Portable Executable
macOS, iOS, tvOS	Mach-O	Mach Object

Most Relevant Architectures

x86-64 / AMD64	desktop
ARM64 / AArch64	embedded, mobile, but also increasingly for desktop
RISC-V	embedded

In order to offer a usable compiler that does not translate to C or LLVM IR, all common operating systems in their respective combinations with the listed architectures would actually have to be addressed. But even here, some cases can be excluded, namely Windows and Apple, because both systems play no role in the server area, so that support would only be worthwhile for the two Unix systems Linux and FreeBSD anyway, not least because Windows can also run Linux applications with WSL.

However, it must be emphasized again that separate backends are only justified if Kalkyl can actually be translated noticeably much more efficiently into machine code without C or LLVM IR as a detour, which is hardly realistic with regard to GPUs, so LLVM support still seems to be a serious request, especially in view of AI development, where Kalkyl could be a good alternative to Python – and the planned Mojo.

Nevertheless, this discouragement does not speak against an experimental plugin for the two most common instruction set architectures x86-64 and ARM in combination with Linux to cover the server area, if only to estimate the effort and to explore how a purely functional language like Kalkyl without GC can be converted directly into machine instructions.

Direct Output of LLVM Bitcode

An alternative to an LLVM plugin using the C or C++ API directly, so that the entire framework inevitably has to link, would be to output LLVM bitcode directly in addition to C, which can then be processed further by clang as an external program called. This approach would offer the decisive advantage of already providing incremental compilation in a simple way by creating a BC file for each Kalkyl component, so that only changed components would have to be recompiled to bitcode. But the real advantage would be that Zuse can remain completely independent of LLVM and thus very small in size, which ease building and distributing of the compiler. GCC or clang are required anyway to use existing C and C++ libraries, so the integration of bitcode ultimately fits well into the existing pipeline without causing too much additional work.