The Monty Language Specification

Monty's Coconut

Studiengang Informatik
Universität Bremen
D 28 344 Bremen
monty@informatik.uni-bremen.de

Version 1.1
August 18, 2015

Contents

1 Introduction

In the last decades, a vast variety of programming languages have been designed. They support different paradigms of programming, and focus on divergent application areas. A significant distinction is between classical programming languages, which typically are compiled, execute fast and have a static type system, and so-called scripting languages, which, in contrast, are mostly interpreted, dynamically typed, easy to use, but have moderate execution performance.
This document defines *Monty*, a programming language that combines the advantages of programming and scripting languages. *Monty* shall provide a maximal comfort in programming, with focus on performance. The language relies on class-based object-orientation. Everything in *Monty* is an object. In fact every object has a defined value at any time. Values like *nil*, *null* or *void* are not needed. To achieve high performance, the language is based on a static type system. Nevertheless, flexible typing through abstract classes and the possibility to extend objects at runtime improve programming comfort. The design of the syntax has been driven by the ideals of compactness and readability.
This document defines the *Monty* programming language, and includes a rationale for the design decisions taken. It is divided into nine chapters. Starting with the introduction of lexical elements (chapter 2) and continuing with a description of all available objects (chapter 3), the first chapters deal with the basic data model. The following chapters deal with expressions (chapter 4), statements (chapter 5) and declarations (chapter 6). Chapter 7 focuses on classes and the structure of objects, including the concepts polymorphism and inheritance, leading towards a plain introduction of modules and packages (chapter 8). Closing with the rationale in chapter 9, a summary and justification displays the most remarkable decisions in the design process of the *Monty* programming language.

How *Monty* was designed

The language *Monty* has been developed in the student project "Monty's Coconut" (`http://www.informatik.uni-bremen.de/monty/`) of the Masters program in Computer Science at Universität Bremen. The authors are students of that project.

About the name *Monty*

When David A. Watt (University of Glasgow) proposed a language with flexible typing that should combine the syntactic compactness of Python with the object-oriented type system of Java, he called it *Monty*, after the British comedy group "*Monty Python*" [11]. When the idea for the student project was born, David's language and paper have been a major source of inspiration. On a workshop in December 2013, where we presented the project's ideas for the new language to him, David has granted us permission to use the name of his language proposal (which has never been implemented) for our language. ("Coconut" in the name of the project refers to the German title of the movie "Monty Python And The Holy Grail", "Die Ritter der Kokosnuss", "Knights of the Coconut" in English).

Syntax Notation

The form of a Monty program is described by means of a context-free syntax together with context-dependent requirements expressed by narrative rules. The meaning of Monty programs is described by means of narrative rules defining both the effects of each construct and the composition rules for constructs.
The context-free syntax of the language is described using a simple variant of *Backus-Naur Form*. In particular:

- Lower case words in a sans-serif font denote *syntactic categories*, which define syntactic portions of a program, for example: *returnStatement*;
- Boldface words denote reserved words, which appear literally (in normal face) in a program, for example: **`return`**;
- Strings enclosed in single quotes denote delimiters and operators, for example `'<='` and `')'`, which appear literally (then without the enclosing apostrophes) in a program;
- Rules of the form *category* ::= *expression* define the form of a syntactic category. So the rule

  ---

  handler ::= **`handle`** *exception*
  block

  ---

  specifies that the syntactic category *handler* consists of the a reserved word **`handle`**, followed by the syntactic category *exception*, the delimiter `':'`, and the syntactic category *block*.
- A vertical bar "|" (pipe symbol) separates alternative forms that a syntactic category can take:

```
returnStatement ::= return | return expression
```

- Square brackets enclose an optional item. Thus the two following rules are equivalent.

```
returnStatement ::= return [expression]
returnStatement ::= return | return expression
```

- Curly brackets enclose a repeated item. The item may appear zero or more times; the repetitions occur from left to right as with an equivalent left-recursive rule. Thus the two following rules are equivalent.

```
handleStatement ::= try
                    block
                    { handler }
handleStatement ::= try
                    block
                  | handleStatement
                    handler
```

The syntax rules depend on line breaks and require a certain indentation of blocks (see section 2.2 for a precise description of indentation rules). For this reason, three special tokens are inserted during the lexical analysis. The *newline* token ↵ stands for one or more line breaks, while the *indent* ⤳ and *dedent* ⤺ tokens denote an increased or decreased indentation level.

```
block ::= ':' ↵
          ⤳
            ( 'pass' ↵ | blockContent { blockContent } )
          ⤺
```

Samples of *Monty* programs are set in typewriter front (with keywords highlighted). An example of an exception handler could look as follows (Here the ↵, ⤳ and ⤺ Tokens are omitted. Instead, the line breaks and line indentation appear literally as they are required):

```
handle InputError :
    return 0.0
```

2 Lexical Elements

The text of a program consists of lexical elements, each composed of characters; the rules of composition are given in this chapter.

2.1 Comments

A comment designates text that shall document the program, but has no significance for its semantics. It begins with two forward slashes and ends with a new-line character (as defined in 2.2). Comments may contain Unicode characters. The syntax is defined as follows:

Comment ::= '//' {*EveryCharButNewline*} ↵

Where *EveryCharButNewline* stands for any character which is different from *Newline*.
Comments may occur in code as follows:

```
// Line comment.
print("Hello World") // Comments may appear after statements.
```

Comments can be placed anywhere in the program. The indentation rules (see section 5.2) do not apply for comments.

2.2 Layout Characters

Layout characters separate lexical elements. The following ASCII characters are used as layout characters:

1. Horizontal tabulators (\t, decimal code 9)
2. Line feeds (\n, decimal code 10)
3. Carriage returns (\r, decimal code 13)
4. Spaces (decimal code 32)

LayoutCharacter ::= *Space* | ↵
LayoutSequence ::= *LayoutCharacter* { *LayoutCharacter* }

Space ::= '\t' | ' '
Newline ::= ['\r'] '\n' { ['\r'] '\n' }

The function of layout characters in code is twofold:

1. Lexical: Separate lexemes in order to make these distinguishable
2. Syntactical: Indentation as a means to designate blocks

Indentation is defined as follows:

i) A *line* l is a sequence of tokens, which has exactly one $\hookleftarrow$ token. The $\hookleftarrow$ token is the last token in that sequence.

ii) A line's *indentation* is the longest prefix of l, which only contains *Space* characters.

iii) The length of the indentation is called *indentation level*.

The indentation is processed during the lexical analysis:

- A $\rightsquigarrow$ token is inserted if the indentation level of a non-empty line is greater than the indentation level of the previous line.
- A number of $\hookleftarrow$ tokens is inserted, if the indentation level of a non-empty line is less than the indentation level of the previous line and the following condition holds:

  The indentation level of the current line is equal to an indentation level of one of the previous lines.

  The number of $\hookleftarrow$ tokens inserted is equal to the number of indentation levels which exist between the previous and the current line.
- An empty line is a line which only contains layout characters or comments

The line indentation level determines how blocks are grouped, see section 5.1 for details.

2.3 Identifiers

Identifiers consist of letters, digits and underscores. Three kinds of identifiers are distinguished:

```
Identifier   ::= { '_' } LowercaseLetter { IdentifierSymbol }

ConstantIdentifier ::= { '_' } UppercaseLetter
                       { UppercaseLetter | Digit | '_' }

ClassIdentifier    ::= UppercaseLetter { IdentifierSymbol }
                       LowercaseLetter { IdentifierSymbol }

IdentifierCharacter ::= LowercaseLetter | UppercaseLetter | Digit | '_'
```

Digits and letters are defined as follows:

```
Digit           ::= '0' | ... | '9'
LowercaseLetter ::= 'a' | ... | 'z'
UppercaseLetter ::= 'A' | ... | 'Z'
```

The first character of any identifier must be a letter or underscore character. The case of letters is used to keep variable, procedure or module identifiers apart from constant and class identifiers. All types of identifiers only may only contain ASCII characters.

```
DEADBEEF      // constant
_PI           // constant
Network       // class
TcpConnection // class
__connect     // variable, procedure or module
onEvent       // variable, procedure or module
connected     // variable, procedure or module
```

2.4 Keywords

The following reserved words cannot be used as identifiers:

```
abstract and as break class else false handle if import in inherits
initialize not or parent pass raise self skip true try while
```

2.5 Delimiters

Delimiters are lexemes that structure the program:

```
+ - ~ # : , . ( ) [ ] < > := += -= *= /= %= ^=
```

The delimiters `+`, `-`, `<` and `>` are also used as operators.

2.6 Operators

Operators define a reserved set of special characters and keywords that can be used as a syntactic sugar for special method calls. There is a distinction between unary and binary operators:

```
unaryOperator ::= not | '-'
binaryOperator ::= '+' | '-' | '*' | '/' | '%' | '^' | '=' |
                   '!=' | '<' | '>' | '<=' | '>=' | '->' | in
operator ::= unaryOperator | binaryOperator
binaryOperatorLike ::= as | is | and | or
```

Unary operators use the prefix notation, i.e. they are written in front of the function argument, without parenthesis. Binary operators use the infix notation, which means that they are written between the arguments.
The minus (-) is a unary operator (returning the additive inverse of a number) as well as a binary operator (for subtracting numbers).
Additionally, there are four binary operations, which are not translated into method calls. These are **`as`**, **`is`**, **`and`** and **`or`**. These are also described in section 4.4.

2.7 Literals

A literal denotes constant values of a predefined object type, i.e. truth values, integral and floating point numbers, single characters and character strings.

```
literal ::= BoolLiteral
          | IntLiteral
          | FloatLiteral
          | CharacterLiteral
          | StringLiteral
```

Each literal denotes an object, that is an instance of its corresponding data type. For example, **`true`** is of type `Bool`, whereas `123` is an object of type `Int`.

2.7.1 Boolean

A boolean literal is either **`true`** or **`false`**.

```
BoolLiteral ::= true | false
```

2.7.2 Integer

An integer literal consists of digits or letters. Optionally, a positive exponent and the base may be specified.

```
IntExponent ::= ( 'e' | 'E' ) [ '+' ] Digit { Digit }

Base ::= [ '0' | '1' | '2' ] Digit
       | '3' ('0' | ... | '6')

IntLetter ::= 'A' | ... | 'F'
            | 'a' | ... | 'f'

IntLiteral ::= ( Digit { Digit } [ IntExponent ] )
             | ( '0' { IntLetter | Digit } '_' Base )
```

If a base is specified, then the literal must start with the digit `0`. The base specifier follows a leading underscore. An exponent starts with an `e` or `E` and an optional sign character (`-` or `+`). Exponents can only be specified for base 10 and cannot be negative for integers.

```
42
02
0PL1_21
42e3
```

2.7.3 Float

Float numbers consist of an integer part and a fractional part, separated by a dot. Optionally an exponent may be specified.

```
FloatingLiteral ::= Digit { Digit } '.' Digit { Digit }
                    [ FloatExponent ]
FloatExponent ::= ( 'e' | 'E' ) [ '−' | '+' ] Digit { Digit }
```

Note that `.4` and `1.` are not valid float literals, since at least one leading and one trailing trailing digit is required.

```
511.37
0.214
0.000042e+6
```

2.7.4 Character

A character literal is enclosed in single-quotes. Escaping via a backslash is required for single quotes (`'`) as well as for some escape sequences, described in Table 2.1

```
CharacterLiteral ::= '\'' Character '\''
Character ::= UnicodeCharacter | CharacterEscapeSequence
```

A *UnicodeCharacter* is a C0 or C1 control character with the exception of single quotes (`'`) and backslashes. Double quotes may or may not be escaped in `Char` literals, due to the compatibility with `String` literals. *CharacterEscapeSequence* defines a sequence of escaped characters. It is one of the following:

Escape sequence	ASCII name
`\\`	Backslash (\)
`\'`	Single quote (')
`\"`	Double quote (")
`\a`	Acoustic signal (decimal code 7)
`\b`	Backspace (decimal code 8)
`\t`	Horizontal tab (decimal code 9)
`\v`	Vertical tab (decimal code 11)
`\n`	Linefeed (decimal code 10)
`\f`	Formfeed (decimal code 12)
`\r`	Carriage return (decimal code 13)
`\uxxxx`	Unicode character with 16-bit hex value *xxxx*

Table 2.1: Escape sequences in string and character literals

Examples:

```
'a'   // lowercase character
'Z'   // uppercase character
'\n'  // ASCII newline character
'\\'  // backslash character
'ç'   // Unicode character
'\''  // single quote
'\"'  // escaped double quote
'"'  // unescaped double quote
```

2.7.5 String

A string is a sequence of zero or more characters.

```
StringLiteral ::= '"'
                  { CharacterEscapeSequence | UnicodeStringCharacter }
                  '"'
```

UnicodeStringCharacter is a C0 or C1 control character with the exception of double quotes ("), backslashes (\) and line breaks. The same escape sequences as for characters are also valid in strings (see Table 2.1). A string cannot span more than one line.

```
"Hello \n World!"  // Hello
                   //  World!
```

3 Objects

Every entity of *Monty* is an object. An object has a unique reference, referred to as an object's *identity*. Its *type* is defined by a corresponding *class* (which will be introduced in chapter 7). Every object is an instance of its type. The class itself defines the object's internal structure. An object is defined by its *features*. Features of an object are its *attributes* and *methods*. Attributes are constants and variables defined inside an object.
Variables are addresses of objects bound to an identifier. Similarly, constants are addresses of immutable objects bound to some constant identifier. While variables can be updated, constants can only be set once. Methods are functions and procedures defined on an object and will be introduced in the following chapters. An object may also include *dynamic features* which are not defined at compile-time but rather at runtime.
Objects respond to specific *messages*. More precisely they respond to exactly those messages that access an object's feature (see chapter 7 for details). Hence, an object o with features $f_1, \ldots f_{k+l}$, from which $f_1, \ldots, f_k$ are attributes and $f_{k+1}, \ldots, f_l$ are referred to as methods $m_1, \ldots, m_l$ responds to messages of the following form:

Message form	Description
$o.f_i$	yields the feature f_i of o
$o\texttt{->}f_i$	yields the dynamic feature f_i of o
$o.m_i(p_1, \ldots, p_n)$	calls method m_i of o with parameters $p_1, \ldots, p_n$
$o\texttt{->}m_i(p_1, \ldots, p_n)$	calls dynamic method m_i of o with parameters $p_1, \ldots, p_n$

Any feature of an object can be requested by the first two kinds of messages. However, only methods of an object can be called, using the last two kinds of messages. If the feature requested by a message is not defined on the receiving object, a compile error will occur. In case of dynamic features, an *AccessException* will be raised at runtime (see section 5.8 for details on exceptions).
Every object satisfies one and only one of the following conditions:

1. All attributes are constant and can not be changed by any messages. (*immutability*)

2. At least one attribute is a variable and can be changed by specific messages. (*mutability*)

Objects of simple types `Int`, `Float`, `Char`, `Identity`, `Bool` and `String` are immutable. All other types and also all types defined by a user can not be assured to be immutable. Objects of immutable types do not have dynamic

features. Furthermore, every message to such a receiver object preserves the receiver and returns a new object. The following sections will introduce the predefined objects in *Monty*.

3.1 Simple Objects

Simple objects only have one attribute that describes its internal value. It is limited to a specific domain. Therefore, simple objects are also referred to as *scalar values*. The only attribute of a simple object is constant, which makes all predefined simple objects immutable.

3.1.1 Boolean

Boolean objects are of type *Bool*. The value of boolean objects corresponds to the binary domain:

$$\{false, true\}$$

The *Bool* type not only inherits from *Equal* for enabling boolean objects to be checked for equality, but also from *Ordered*, which defines an order on boolean values ($false < true$) and also makes $<=$ available as an implication operator[1]. Boolean objects may be used as arguments for the special **`and`** and **`or`** expressions, which provide logical operations on boolean values. Those expressions are not implemented as operators, since they provide a shortcut-evaluation strategy, which is not possible for method arguments:

Expression	Description
`x and y`	returns *true* if `x` and `y` evaluate to **`true`**. If `x` is **`false`**, `y` is not evaluated.
`x or y`	returns *true* if at least one of `x` or `y` evaluates to **`true`**. If `x` is **`true`**, `y` is not evaluated.

3.1.2 Integer

Integer objects are of type *Int*. The domain of their value is limited to the integral numbers $\mathbb{Z}$.
Integers can be of arbitrary size. Since *Int* is a subclass of the abstract classes *Ordered*, *Equal* and *Arithmetic*, its instances respond to all messages defined in those classes. Additionally the *modulo* operation provides an integer-specific message. Also the arithmetic operators are overloaded to also be compatible with *Float* objects.

[1] Although the direction of the implication arrow does not match its mathematical counterpart

Operation	Description	Usage
`o._mod_(v)`	divides the value of an object *o* by a value *v* and returns the rest	`o % v`

3.1.3 Float

Float objects are implemented as double precision floating point numbers according to the IEEE 754 standard[1]. Since *Float* is a subclass of the abstract classes *Ordered*, *Equal* and *Arithmetic*, its instances respond to all messages defined in those classes.
Additionally the following methods are defined on float objects:

Operation	Description	Usage
`o.round()`	rounds the value of an object *o*	`o.round()`
`o.ceil()`	rounds the value of *o* to the next integer value	`o.ceil()`
`o.floor()`	truncates the fractional part of the value of *o*	`o.ceil()`

3.1.4 Characters

The value of character objects is defined by Unicode characters encoded in UTF-8, each of which correspond to a specific integer value (defined by the Unicode standard). To get the numeric value of a character object the following message can be used:

Operation	Description	Usage
`o.getCode()`	returns the numeric code of the character object *o*	`o.getCode()`

The *Char* class inherits from the abstract class *Ordered*, since the unicode standard assigns an index to each character and thus defines an order on *Char*. Since *Ordered* inherits *Equal*, character objects may also be checked for equality.

3.2 Sequences

Sequence objects are collections whose elements are ordered. The domain of their attributes is generic, restricting the type of their elements to be a subtype of their type parameter (for details on generics see section 7.4).
The following methods define specific messages for sequence objects:

Operation	Description	Usage
`s._at_(i)`	returns the element at index i of sequence s	`s[i]`
`s.getFirst()`	returns the first element of sequence s	`s.getFirst()`
`s.getLast()`	returns the last element of sequence s	`s.getLast()`
`s.getSize()`	returns the length of sequence s	`s.getSize()`
`s.isEmpty()`	returns *true*, if sequence s is empty	`s.isEmpty()`
`s._contains_(e)`	returns *true* if e is element of sequence s	`e in s`
`s1._add_(s2)`	concatenates the sequences s_1 and s_2	`s1 + s2`

The following sequence objects are available:

- Array
- List
- String

3.2.1 Arrays

Array objects are mutable sequences of fixed size. The number of their elements, once determined, can not be changed. The following methods define specific messages, array objects respond to:

Operation	Description	Usage
`a._set_(i,v)`	sets element at index i of array a to value v	`a[i] := v`

3.2.2 Lists

List objects are sequences of flexible size. They are mutable and respond to the messages defined on sequence objects. The following methods define specific messages, list objects respond to:

Operation	Description	Usage
`l._set_(i,v)`	sets element at index i of list l to value v	`l[i] := v`
`l.removeAt(i)`	removes the element at index i out of list l	`l.removeAt(i)`
`l.remove(e)`	removes all elements out of l, which are equal to e	`l.remove(e)`

3.2.3 String

String objects are special sequence objects, whose attributes are limited to the previously introduced domain of Unicode characters. The following methods define specific messages available for string objects:

Operation	Description	Usage
`s.join(seq)`	creates a new string out of a sequence of strings *seq* by folding the sequence, concatenating each element with the sting *s*	`s.join(seq)`
`s.split(sep)`	splits string *s* after the occurrence of each string *sep*	`s.split(sep)`
`s._mul_(n)`	concatenates string *s* exactly *n* times with *s*	`s * n`
`s._add_(t)`	concatenates string *s* and string *t*	`s + t`
`s.substring(o,n)`	returns *n* characters from string *s* starting at index *o*	`s.substring(o,n)`

String objects are immutable, but there is a mutable equivalent *MutableString* class for cases where performance is important while performing many operations on a string object-.

3.2.4 MutableString

The *MutableString* class provies a mutable alternative to *String*. It inherits from *String* and *List* and therefore offers the methods of both classes.

3.3 Maps

Maps are associative arrays, whose elements are not identified by indices, but are associated with some arbitrary key object. Elements inside a map do not have any order. Their type *Map* has two generic type parameters. The first one determines the type of a map's *keys*, the second one determines the type of its *values*.
The following methods define specific messages, map objects respond to:

Operation	Description	Usage
`m.isEmpty()`	returns *true*, if map m is empty	`m.isEmpty()`
`m.getSize()`	returns the number of elements in map m	`m.getSize()`
`m._at_(k)`	returns the element at key k of map s	`m[k]`
`m._set_(k,v)`	sets the element at key k of map m to v	`m[k] := v)`
`m.remove(k)`	removes the element at key k out of map m	`m.remove(k)`
`m.getKeys()`	returns a sequence of all keys in m	`m.getKeys()`
`m.getValues()`	returns a sequence of all values in m	`m.getValues()`
`m._contains_(k)`	returns *true* if k is a key in m	`k in m`
`m.contains(k, v)`	returns *true* if k, v is a key-value pair in m	`m.contains(k, v)`

3.4 Functions and Procedures

Functions and procedures are objects. They consist of an arbitrary number of parameters and a sequence of statements (a *block*, see section 5.1), which is also called the *body* of a procedure. The type of the parameters is defined by the generic type parameters of the *Procedure* class, which is illustrated in Appendix B.
Procedures also respond to one special message: a call. Calling a procedure named *proc* with parameters $p_1, \ldots, p_k$ has the following form:

$$proc(p_1, \ldots, p_k)$$

A call executes all statements of a procedure. Aside from all other messages, procedure calls do not have any response value.
Functions are special procedures. They share the functionality with procedures but additionally have a *return value*. The type of the return value is determined by an additional generic type parameter. Calling a function will return this attribute as a message response.

3.5 Further Objects and the Data Model

Since every object has a type that refers to a class, relations between objects can be visualized as class diagrams. An overview of predefined classes is given in Appendix B.

3.5.1 Object and Identity

The type *Object* is the root of the type hierarchy. All further types inherit from object, directly or indirectly. The type *Object* itself is an *abstract type* (more on abstract classes in chapter 7). It is not possible to instantiate *Object* explicitly. The ID attribute of an object is of type *Identity*. This built-in type encapsulates an objects unique address. By deriving from the abstract type *Equal*, identities are comparable, providing equality checks on objects.

3.5.2 Class

The type *Class* directly inherits from *Object*, since it defines the most general kind of objects. All class objects share two basic features:

1. A name,
2. A set of super classes.

Similarly to procedure objects, classes are callable. Calling a class object triggers the constructor of the class, which leads to the instantiation of an object (for details see chapter 7).

3.5.3 Abstract Types

The data model makes use of *abstract type classes* to bundle common functionality or interfaces that are shared among similar classes (for a definition of abstract classes see subsection 7.2.3).

Equal

The abstract *Equal* class contains abstract methods, which ensure the interface for equivalence-tests on different objects.

Operation	Description	Usage
`o._eq_(p)`	returns **`true`** if *p* equals *o*	`o = p`
`o._neq_(p)`	returns **`true`** if *p* does not equal *o*	`o != p`

Ordered

The abstract *Ordered* class contains abstract methods, which ensure the interface for comparing objects. *Ordered* inherits from *Equal*.

Operation	Description	Usage
`o._lt_(p)`	returns **`true`** if *p* is less than *o*	`o < p`
`o._gt_(p)`	returns **`true`** if *p* is greater than *o*	`o > p`
`o._leq_(p)`	returns **`true`** if *p* is less than or equals *o*	`o <= p`
`o._geq_(p)`	returns **`true`** if *p* is greater than or equals *o*	`o >= p`

Arithmetic

The abstract *Arithmetic* class contains the interface for the five basic arithmetic operations (addition, subtraction, multiplication, division and the power function). The *Int* and *Float* classes inherit from *Arithmetic*. Also, *Arithmetic* inherits from *Ordered*.

Operation	Description	Usage
`o._add_(p)`	adds the value of object *p* to value of object *o*	`o + p`
`o._sub_(p)`	subtracts the value of object *p* from value of object *o*	`o - p`
`o._mul_(p)`	multiplies the value of object *p* by value of object *o*	`o * p`
`o._div_(p)`	divides the value of object *p* by value of object *o*	`o / p`
`o._pow_(p)`	multiplies the value of object *p* by value of object *o*	`o ^ p`

3.5.4 Exception

The abstract type *Exception* is the basic type of all runtime errors. Exceptions are objects, whose type derives from *Exception*. For more information on Exceptions see section 5.8.

Operation	Description	Usage
`e.getMessage()`	returns the message of the exception	`e.getMessage()`
`e.getTrace()`	returns the stack trace of the exception	`e.getTrace()`
`e.getLine()`	returns the line where the exception occurred	`e.getLine()`
`e.getFile()`	returns the file where the exception occurred	`e.getFile()`

4 Expressions

Expressions are syntactical constructions that evaluate to an object. They bind values – introduced in chapter 3 – to syntactical elements.

```
expression ::= literal
             | '(' expression ')'
             | name
             | unaryOperator expression
             | expression binaryOperator expression
             | conditionalExpression
             | aggregation
             | functionCall
```

4.1 Literals

In chapter 2 all available literals have been introduced. Those refer to the objects of type `Int`, `Float`, `Bool`, `Char` and `String`, described in chapter 3. Note that string literals create objects of the immutable `String` class, while mutable strings have to be explicitly converted by using the `MutableString` constructor.

4.2 Collections

The syntax of collections refers to *aggregates*, which compose values to create new expressions. Since all collections are generic, their semantics is not directly expressed by the corresponding syntactical elements. Elements of a collection have to be homogeneous, but they may still differ in their actual type, as long as they are subtypes of the class' type parameter. Syntactically, there are two kinds of collections:

```
collection ::= map
             | sequence
```

Map. Maps correspond to the following syntactical structure:

```
map ::= '[' [ keyValuePair { ',' keyValuePair } ] ']'
keyValuePair ::= expression ':' expression
```

The following examples correspond to valid map objects:

```
["abc": 123]                // Map of string keys and integer values
[42: "Hello", 153: "World"] // Map of integer keys and string values
```

Sequences

There are three types of sequences available:

```
sequence ::= array
           | list
           | string
```

Array. Arrays are represented by the following syntax:

```
array ::= '[' [ expression { ',' expression } ] ']'
        | range
```

When evaluating an expression of the above form, all expressions will be counted first. After that, an array object will be created, whose size equals the element count in the expression. The expressions will then be evaluated (from left to right), binding the resulting objects on their respective indices in the new array object.
Arrays of integers intervals can be created by using a special syntax for *ranges*.

```
range ::= '[' expression '..' expression ']'
```

The first and last expression in a range have to be integer objects. Consider a range r with the expressions being a and b. The relations between a and b result in the following ranges:

Relation of a and b	Range
$a = b$	$[a]$
$a < b$	$[a, a+1, \ldots, b]$
$a > b$	$[a, a-1, \ldots, b]$

List. Lists do not provide special syntax. Creating a list object is similar to the creation of arbitrary objects: by calling the class corresponding to the type of the object. To create a non-empty list, it is possible to parameterize this call with an array, as the following example shows.

```
List()             // creates an empty list
List([1, 2, 3, 4]) // creates a list of integers 1, 2, 3 and 4
```

4.3 Names

Every object can be bound to a name. Referencing that name accesses the associated object. The following kinds of names can be distinguished:

```
name ::= ClassIdentifier | variable

variable ::= variableName
           | expression '.' variableName      // regular access
           | expression '->' variableName     // silent dynamic access
           | expression '[' expression ']'    // subscription

variableName := Identifier | ConstantIdentifier | self
```

The most simple names are *ClassIdentifier*, *Identifier*, *ConstantIdentifier* and the keyword **self**. Evaluating a constant identifier returns the constant bound to that name. Similarly, evaluating an identifier returns a variable, while evaluating a class identifier returns the corresponding class. The semantics of the keyword **self** is described in subsection 7.1.1.
Variables are a special case of names, since they represent an address in memory. Thus they can be used on the left-hand side of an assignment.
Names occur in different scopes. For example the name of an attribute a of some object o_1 may not be known to some object o_2. Qualifying a name resolves that issue. Considering an object o with a feature f, the following qualifiactions are possible:

```
o.f
o->f
```

Qualifying a name first evaluates the name of the *receiver object* o. The name of the feature f is then looked up in the scope of the class of o by using the access operation (`.` or `->` respectively). The result of the access expression is the requested feature. The feature lookup is done via *dynamic dispatch*. This means that the returned feature does not only depend on the static type of the variable containing o, but also the dynamic type of o is considered in order to find the correct feature.
If the requested feature is a procedure (or function), the actual parameter types of the procedure call are used to determine the correct overloaded version ot that procedure (if the procedure is overloaded at all). For a detailed description of the overload mechanism see section 6.6.

4.3.1 Function and Constructor Calls

A function call is a special procedure call. While procedure calls execute the body of a procedure with a number of actual parameters (without returning any value), function calls execute the body of a function similarly, returning

the value of a return statement inside it. Function calls follow the syntax of procedure calls:

```
functionCall ::= expression '(' [ expression { ',' expression } ] ')'
```

In a function call $N(A_1, \ldots, A_k)$, the *name* N must be bound to a non-empty set of overloaded functions F. Exactly one procedure $f \in F$ must be given with k formal parameters of types $t_1, \ldots, t_k$ that *fit best* to the types $u_1, \ldots, u_k$ of the actual parameters $A_1, \ldots, A_k$ (see section 6.6 for details on overload resolution). A function call $N(A_1, \ldots, A_k)$ is executed as follows: The name N is evaluated in the context of the actual parameter types $u_1, \ldots, u_k$ to yield a procedure object f. The actual parameters A_1 ... A_n are evaluated to yield argument objects a_1 ... a_k, which are assigned to the formal parameters $x_1, \ldots, x_k$, which are local variables inside the procedure. Finally, the body of f is executed.
Functions which are bound to classes are called *methods*. Methods are always qualified by the name of the corresponding object. The qualification therefore specifies the receiver object for the function application. If an object calls one of it's own methods, a call has to be qualified by the keyword **`self`**.
Function names can also be represented by *ClassIdentifiers*, providing a shorthand syntax for constructor calls, which will be further introduced in chapter 7. These special function calls evaluate to a new instance of that class.

4.3.2 Subscriptions

Accessing the element at index i of a sequence s is done by using *subscriptions*. There are two cases where subscriptions may occur.

```
t := s[i]   // subscription as rvalue
s[i] := t   // subscription as lvalue
```

Any subscription will first evaluate the expression s to the receiver object. After that, a special operation is invoked on that object, depending on whether the subscription appears on the left-hand side of an assignment or not. If the subscription appears on the left-hand side, the `_set_(i,v)` method is invoked, where i is the index of the subscription and v is replaced by the right-hand side of the assignment. Otherwise, the `_at_(i)` method is invoked, parameterized with the value of the index i.
The two subscription methods are implemented on sequence classes, but may also be implemented in user-defined classes in order to allow subscription.

4.4 Operators

Operators are syntactic sugar for special method calls. Binary operators can be written as $e_1 \otimes e_2$ with e_1, e_2 being expressions and $\otimes$ a binary operator. Unary operators can be written as $\otimes\, e$ with e being an expression and $\otimes$ the

unary operator.
An expression $e_1 \otimes e_2$ is translated into a method call: `e1._opMethod_(e2)`, where `_opMethod_` is a method of e_1. Similarly, an expression $\otimes e_1$ is translated into a method call: `e1._opMethod_()`, where `_opMethod_` is a method of e_1. The exact mapping and a list of available methods for operator notation is shown in the following:

Binary Operators

The following table shows all available binary operators.

Operator	Method Translation	Description
`x + y`	`x._add_(y)`	Addition
`x - y`	`x._sub_(y)`	Subtraction
`x * y`	`x._mul_(y)`	Multiplication
`x / y`	`x._div_(y)`	Division
`x % y`	`x._mod_(y)`	Modulo Division
`x ^ y`	`x._pow_(y)`	Power
`x = y`	`x._eq_(y)`	Equality Check
`x != y`	`x._neq_(y)`	Inequality Check
`x < y`	`x._lt_(y)`	Less-Than Check
`x > y`	`x._gt_(y)`	Greater-Than Check
`x <= y`	`x._leq_(y)`	Less-Than-or-Equal Check
`x >= y`	`x._geq_(y)`	Greater-Than-or-Equal Check
`x in y`	`y._contains_(x)`	Contains Check

Most of the binary operators are left associative. The only exception is the power operator `^`, which is right associative.
When evaluating an expression of the form $e_1 \otimes e_2$, the left expression e_1 is evaluated first, returning the receiver object of the operation $\otimes$. After that, the operation $\otimes$ has to be resolved as a method of the returned object of e_1. Finally, the method is invoked, taking e_2 as a parameter.
The `in`-operator evaluates different. Instead of taking the left expression as a receiver object, the evaluation-strategy is reversed, taking the right expression as the receiver and the left expression as a parameter.

Operator-Like Expressions

There are four further expressions which have a notation similar to binary operators. However, their operations can not be translated into usual method calls:

Operator	Description
`is`	Instance-Check
`as`	Typecast
`and`	Logical Conjunction
`or`	Logical Disjunction

The expression `o is T` checks whether `o` is an instance of the class `T`. Similarly, the expression `e as T` performs a typecast, which causes the expression `e` to be treated as it had the type `T`.
The logical conjunction and disjunction can not be translated into method calls, since their expected evaluation strategy is different from usual method calls. They provide a short-cirquit evaluation. In an expression `x and y`, the term `y` does not have to be evaluated if `x` is already `false`. Similarly, in an expression `a or b`, the term `b` does not have to be evaluated if `a` is already `true`. In order to achieve this short-cirquit evaluation strategy, an expression `x and y` is translated into `y if x else false`, while an expression `a or b` is translated into `true if x, else y` (see also section 4.5). This only works for `x`, `y`, `a` and `b` being of type `Bool`.

Unary Operators

Operator	Method Translation	Description
`-x`	`x.negate_()`	Negation
`not x`	`x.not_()`	Boolean negation

Unary operators of the form $\otimes\, e$ evaluate the expression e first. After that, the operator $\otimes$ is resolved as a method-identifier. Finally, the resolved method is called on the object, yielded by the evaluation of e in the first step.

Precedence

The following precedences affect the order of evaluation:

Precedence	Operator	Description
2	`-`	unary minus
	`not`	logical not
3	`^`	power
4	`*`	multiplication
	`/`	division
	`%`	modulo
5	`+`	addition
	`-`	subtraction
6	`<`	less than
	`<=`	less than or equal
	`>`	bigger than
	`>=`	bigger than or equal
7	`=`	equals
	`!=`	unequals
	`is`	type check
8	`in`	contains check
9	`and`	logical and
10	`or`	logical or
11	`as`	type cast

Evaluating an expression, which only includes operators from the same precedence class, does not affect the default evaluation strategy. However, using operators of different precedence without ensuring the evaluation strategy of the expression by the use of parentheses, follows the above precedences. Enclosing an expression in parentheses does not change its value, so that the expressions (e) and e are equivalent.
Surrounding an expression with parentheses, encapsulates the expression in a way that everything inside the parentheses is evaluated before any operations outside the parentheses can be applied. Together with *Monty*'s default evaluation mechanism, the following examples are equal to the expressions in the comments:

```
1^2^3     // 1^(2^3)
1+2-3     // (1+2)-3
1-2*3+4   // (1-(2*3))+4
```

4.5 Conditional Expressions

Conditional expressions evaluate to different values, depending on a *condition*.

conditionalExpression ::= *expression* 'if' *expression* 'else' *expression*

The expression between both keywords `if` and `else` is called *condition* and has to evaluate to a boolean value. According to the condition, a conditional

expression evaluates to the first expression (in front of the *if*), if the condition is *true*, and to the last expression (behind the *else*) if the condition is *false*. Only one of both sub-expressions will be evaluated.

```
-1 if x > 0 else 1
```

5 Statements

A statement is executed in order to update variables.

```
statement ::= assignment
            | compoundAssignment
            | conditionalStatement
            | loopStatement
            | 'skip' ↵
            | 'break' ↵
            | procedureCall
            | 'return' [ expression ] ↵
            | raiseStatement
            | tryStatement
```

5.1 Block

A sequence of local declarations and statements forms the content of a block. Local declarations are defined in chapter 6. An empty block is denoted by the keyword **pass**.

```
block ::= ':' ↵
        ⤳
          ( 'pass' ↵| blockContent { blockContent } )
        ⤻

blockContent ::= statement { ↵}
               | localDeclaration { ↵}
```

A block is executed by elaborating its local declarations and executing its statements in the order they appear. Indentation rules for blocks are defined in the following subsection. The elaboration of declarations in blocks is defined in section 6.5.

5.2 Structure and Indentation

The structure of a program is determined by indentation. *Monty* is line-oriented such that statements are terminated by line breaks. For convenience reasons, line breaks are ignored if it is obvious that an expression is not terminated at the end of a line. This is the case if the number of opening braces

does not match the numer of closing braces or if the second argument for a binary operator is missing.
Blocks are introduced using the colon character at the end of a line followed by indentation. Two types of blocks are distinguished:

1. Standard blocks are used as the body of a procedure or function, as well as for bodies of **if**, **elif**, **else**, **while**, **try** and **handle** statements. They contain local declarations and statements.

2. Class Bodies contain declarations which are features of a class.

The contents of a block, i.e. statements or declarations, must be indented consistently (i.e. all elements of a block have to have the same indentation level). The indentation of a block must be greater than the indentation of the enclosing block. The syntax definition illustrates that by using ⇝ and ⇜ tokens (see section 2.2).

```
while true:
  // Code block.
  print("Hello World") // One space
  print("Hello World") // One space

while true:
     print("Hello World") // Six spaces
     print("Hello World") // Six spaces

// The following will result in a compiler error as the second statement of
// the block has a different width/sequence.
while true:
      print("Hello World") // Six spaces
    print("Hello World") // Four spaces
```

5.3 Return Statement

A return statement may only occur in the bodies of procedures and functions. It terminates the execution of a procedure, and defines the result value if the procedure is a function.
In a procedure, the return statement has no argument. In the body of a function, a return statement must exist and an argument has to be specified to determine the return value of the function. In a function body, every branch of the control flow must end with a return statement **return** *e*, where *e* is an expression and the type of *e* is compatible to the return type of the function. It is possible to have more than one return statement in a function body, but there must be exactly one return statement where every branch ends in (see the example below).

```
// in both possible branches, a return statement is needed
Int max(Int x, Int y):
    if x >= y:
        return x
    else:
        return y
```

5.4 Assignment

Assignments update variables.

assignment ::= *name* ':=' *expression* ↵

compoundAssignment ::= *name compoundSymbol expression* ↵

compoundSymbol ::= '+=' | '−=' | '*=' | '/=' | '%=' | '^='

In an assignment v `:=` e, the type of the expression e must be compatible with the type of the variable v. (See section 7.3 for details.) An assignment v := e is executed as follows: The variable's name is evaluated to yield a variable v. The expression e is evaluated to an object o. The value of v is updated to o. A compound assignment `v` ⊗`=` `e` is executed to carry out the assignment `v := v` ⊗ `e` where ⊗ is one of the compound assignment operators `+`, `-`, `*`, `/`, `%` and `^`. Examples for assignments are:

```
x := 25                 // identifier x is updated to 25
person.name := "Bob"    // person.name is updated to String "Bob"

i += 1                  // a compound assignment,
                        // which is equivalent to i := i + 1
```

5.5 Conditional Statement

A conditional statement selects a statement block for execution, depending on one or more boolean expressions.

conditionalStatement ::= 'if' *expression block*
{ 'elif' *expression block* }
['else' *block*]

The expressions in a conditional statement must be of type `Bool`; they are called *conditions*. A conditional statement

if C_1 : B_1 **elif** C_2 : $B_2 \ldots$ **elif** C_k : B_k [**else** : B_{k+1}]

is executed as follows:
The conditions $C_1, \ldots, C_k$ are evaluated from left to right, up to the first expression E_i that yields the value *true*. Then the corresponding block B_i is executed. If all conditions evaluate to *false*, the block B_{k+1} is executed, if present.

```
if x = 7:
   print("x equals seven")
elif x = 8:                // optional
   print("x equals eight")
else:                      // optional
   print("x equals, neither 7 nor 8")
```

5.6 Looped Conditional Statement

A conditional loop executes a statement block as long as a condition holds.

loopStatement ::= 'while' *expression block*

The statement block of a loop is called its *body*. A conditional loop statement **while** *C* **:** *B* is executed as follows: The condition *C* is evaluated; if this yields *true*, the block *B* is executed, and the loop **while** *C* **:** *B* is executed again.
The statements *break* and *skip* may only occur in loop bodies. They modify the execution of a loop body as follows: The statement *break* terminates the execution of the innermost enclosing loop. The statement *skip* terminates the current execution of the innermost enclosing loop body, and continues the execution of that loop, by evaluating its condition.

```
Int x := 0
while x < 10:
   print(x)
   Int y := getRandomInt(0,10)
   if y < 3:
      skip    // skips the rest of the current iteration step
   elif y = 4:
      break   // ends the whole loop immediately
   x := x+1
```

5.7 Procedure Call

A procedure call executes the body of a procedure with a number of expressions, which are called its *actual parameters*.

procedureCall ::= *name* '(' [*expression* { ',' *expression* }] ')'

In a procedure call $N(A_1, \ldots, A_k)$, the *name* N must be bound to a set of procedures P. Exactly one procedure $p \in P$ must be given with k formal parameters of types $t_1, \ldots, t_k$ that *fit best* to the types $u_1, \ldots, u_k$ of the actual parameters $A_1, \ldots, A_k$. (See section 6.6 on overload resolution for details.)
A procedure call $N(A_1, \ldots, A_k)$ is executed as follows: The name N is evaluated in the context of the actual parameter types $u_1, \ldots, u_k$ to yield a procedure object p. The actual parameters A_1 ... A_n are evaluated to yield argument objects a_1 ... a_k, which are assigned to the formal parameters $x_1, \ldots, x_k$, which are local variables inside the procedure. The body of p is executed. p may also be a function. In this case, the result value of the call is ignored.

5.8 Exceptions

Exceptions deal with run-time errors and other exceptional situations that may arise during the execution of a program. An *exception* is an object that signals a runtime error in a way so that it can be raised in one context, and handled in another context of the program. The handling of expressions is specified with two statements, which are defined in the following subsections.

5.8.1 Raise Statements

A raise statement raises an exception.

---

raiseStatement ::= 'raise' [*expression*]

In a raise statement **raise** e, *e* denotes an expression of type *Exception*. The execution of a raise statement **raise** e(msg) *raises* the exception named *e* with the error message *msg*.
If the expression *e* is missing, the *raiseStatement* must occur in the block of some handler. In this case, it raises the exception caught by this handler again.

5.8.2 Handle Statements

A *handleStatement* equips a *block* with a sequence of exception handlers.

---

handleStatement ::= 'try' *block*
handler { *handler* }

handler ::= 'handle' [*ClassIdentifier identifier* {',' *ClassIdentifier identifier* }]
block

In a *handleStatement*

try :B **handle** $E_1 e_1$:$B_1 \ldots$ **handle** $E_k e_k$:B_k,

the identifiers occurring in the identifier lists $E_1, \ldots, E_k$ must be declared as exceptions, by use in a raise statement in the program or in the standard library, and, they must be distinct to each other. The names e_i are available as

local variables inside the handler block and refer to the corresponding exception object of type E_i.
A handler of the form h_i = **handle** $e_{i,1} \ldots e_{i,n_i}$:B_i is said to *catch* the exceptions $e_{i,1}, \ldots, e_{i,n_i}$, where $n_i > 0$ and $1 \leqslant i \leqslant i$. We say that a *handleStatement* *catches* all exceptions caught by one of its handlers, and propagates all other exceptions. Furthermore a *block B propagates (all) exceptions* if it does not contain a corresponding *handleStatement*.
During the execution of a program, if the execution of a construct a is defined to consist (in part) of the execution of construct b, then while b is executing, the execution of a is said to d*ynamically enclose* the execution of b. The *innermost dynamically enclosing* execution of a given execution is the dynamically enclosing execution that started most recently.
When an exception e is raised by the execution of a given construct a, the rest of the execution of a is abandoned; that is, any portions of the execution that have not yet taken place are not performed. The construct a is completed. Then:

- If the construct is a *block* of a *handleStatement* that catches e, e is handled by executing that handler; this means that the *block* of the handler is executed instead of the abandoned portion of a.
- Otherwise, e is propagated to the innermost enclosing execution, which means that e is raised again in that context.

6 Declarations

A declaration binds an entity to a name. Every declaration is valid in a certain scope. The visibility rules for declarations and the nesting rules for scopes are described in section 6.5. The following types of delcarations will be described in the following sections:

---

declaration ::= *localDeclaration*
| *classDeclaration*

localDeclaration ::= *variableDeclaration*
| *constantDeclaration*
| *procedureDeclaration*
| *functionDeclaration*

6.1 Variable and Constant Declarations

A variable is a name for an object through which it can be accessed. The variable's type indicates the kind of object that may be assigned to it.

---

variableDeclaration ::= *type Identifier* [':=' *expression*] ↵
constantDeclaration ::= *type ConstantIdentifier* ':=' *expression* ↵

type ::= { *name* '.' } *ClassIdentifier* ['<' *type* { ',' *type*} '>']

When a variable declaration is elaborated, a location is allocated and associated with the variable's name. Variable declarations can be combined with assignments (see section 5.4), which is called *initialization*. It is a short-hand notation for declaring a variable and assigning an object to it. The value of a variable is undefined if an object has not been assigned. If read access on a possibly uninitialized variable is detected, the compilation will fail.
A *constant* is similar to a variable. The only difference is that it must be initialized, and that it cannot be changed afterwards. Constant identifiers must only use uppercase letters.

```
Int i                  // an uninitialized Int variable
String greet := "Hello" // a String variable
Float PI := 3.14159    // a Float constant
```

6.2 Procedure Declarations

A procedure abstracts from a sequence of statements. Procedure declarations bind names to procedures and may be parameterized.

---

procedureDeclaration ::= *procedureHead* ':' *block*

procedureHead ::= *Identifier* '(' [*formalParameters*] ')'

formalParameters ::= *parameter* { ',' *parameter* } { ',' *defaultParameter* }
| *defaultParameter* { ',' *defaultParameter* }

parameter ::= *type Identifier*
defaultParameter ::= *type Identifier* ':=' *expression*

The block of a procedure declaration is called the procedure's *body*. The formal parameters and the body of a procedure form a scope. The actual parameters are available through the names of the formal parameters and can be used as local variables. Names from the enclosing scope are visible within a procedure, unless they are *shadowed*. For the definition of visibility see section 6.5. Once a procedure is called, the values of the argument objects which are passed to the call are then assigned to those variables. Assigning a new object to a parameter will not change the object outside the procedure's body. However, mutating the object itself (i.e. calling a setter-method) leads to side effects outside the procedure's body:

```
Person jDoe := Person("John", "Doe")
Person jSmith := Person("John", "Smith")

johnToJane(Person p, Person q):
    p := Person("Jane", "Doe")  // changes the reference
    q.firstName := "Jane"       // changes the object

johnToJane(jDoe, jSmith)
print(jDoe)     // John Doe
print(jSmith)   // Jane Smith
```

Formal parameters may be initialized in the procedure declaration. In this case they are called *default parameters*. They are syntactic sugar for *overloading*, which is described in section 6.6. When a procedure with default parameters is called, only the mandatory (i.e. not initialized) parameters have to be passed, the default ones are optional. If the argument for a default parameter is missing, the default value will be taken. Default parameters always have to be the last parameters in the list and must not be followed by mandatory parameters. If one parameter has a default value, all of the follwing parameters are expected to be optional. A procedure call $p(E_1, \ldots, E_k)$ where p is a procedure of the form $p(x_1, \ldots, x_m, d_1, \ldots, d_n)$ with $m \leqslant k$ is evaluated

as follows: The actual parameters $E_1, \ldots, E_m$ are assigned to the mandatory formal parameters $x_1, \ldots, x_m$ such that $x_i := E_i$ and $1 \leqslant i \leqslant m$. The remaining actual parameters are assigned to the default parameters such that $d_i := E_{m+i}$ $\forall i \leqslant k - m$. The remaining default parameters are filled with their default values. In fact, this mechanism is a syntactic sugar for overloading (see subsection 6.6.2). The example below shows a procedure having one mandatory parameter:

```
greet(String name):
    print("Hello " + name + ", how are you?")
```

The type of a procedure object depends on the number and type of its parameters. The procedure in the example above expects a `String` parameter. It has the type `Procedure<String>` (see chapter 7 for information on generic types).

6.3 Function Declarations

Functions are a special case of procedures. While proper procedures, as described in the previous section, abstract from statements, function procedures abstract from expressions.

functionDeclaration ::= *type procedureDeclaration*

The **`return`** statement is used to return objects of the functions return type. The body of a function has to contain a return statement for any possible outcome (e.g. in different *if-else* branches) so that it is not possible that the function does not return anything.

```
String stringifyValues(Int a, Float b):
    if a = b:
        return "a and b are " + a.toString()
    else
        return "a is " + a.toString() + " and b is " + b.toString()
```

The type of a function is defined by its parameter and return types. The function in the example above expects a parameter of type `Int` and one of type `Float`, while it returns a `String` object. It has the type `Function<String : Int, Float>`.
Since `Function` is a subtype of `Procedure`, a `Function<X : Y, Z>` object can be used anywhere where a `Procedure<Y, Z>` object is required (but not vice versa, since the return type would be missing, see subsection 7.3.3).

6.4 Class Declarations

A class is a blueprint for an object. When an object is instantiated from a class, its type is the class itself. A class declaration binds a name to a class.

This section briefly describes class declarations. For more information on the semantics of classes see chapter 7.

```
classDeclaration ::= ['abstract'] 'class' ClassIdentifier
                     [ '<' ClassIdentifier { ',' ClassIdentifier } '>' ]
                     ['inherits' typeList]
                     classBody

typeList ::= { name '.' } type
             { ',' { name '.' } type }

classBody ::= ':' ↵
              ↷
                  (featureDeclaration { featureDeclaration } | 'pass')
              ↶

featureDeclaration ::= [accessModifier] ( variableDeclaration
                                        | constantDeclaration
                                        | methodDeclaration )

visibilityModifier ::= '+' | '~' | '#' | '−'

methodDeclaration ::= functionDeclaration
                    | procedureDeclaration
                    | abstractMethodDeclaration

abstractMethodDeclaration ::= 'abstract' [ type ] Identifier
                              '(' [formalParameters] ')' ↵
```

The class features may be declared with their visibility modifiers, which restrict their visibility (described in subsection 6.4.1). Abstract classes and abstract methods are denoted with the keyword **`abstract`**. If some method of a class is abstract, the whole class is abstract.

6.4.1 Visibility Modifiers

The visibility of every feature of a class is determined by a *visibility modifier*. The following visibility modifiers are available in *Monty*:

Symbol	Name	Description
+	*Public*	visible everywhere
~	*Package*	visible inside the package (see section 8.2 for details)
#	*Protected*	only visible inside the class and all its subclasses
−	*Private*	only visible inside the class, itself

If no access modifier is provided, the default visibility is *Package*. The visibility of a feature of some super class can not directly by changed in a subclass,

but an overriding method may define a different visibility modifier than the overridden one (see section 7.3 for more details on overriding and inheritance).

6.5 Visibility and Scopes

A block (as described in section 5.1) is a sequence of statements. Every block forms a scope. Scopes may be nested, since the corresponding blocks may contain other blocks. The following syntactical elements form a block:

- a module declaration
- the body of a class
- the body of a procedure (or a function, respectively), including its formal parameters
- the body of a conditional statement
- the body of a **`while`** loop
- the bodies of **`try`** and **`handle`** statements

Aside from two special cases, a block may contain arbitrary declarations. The first exception is that class declarations may only occur directly within a module, so it is not possible to declare a class within the body of a function, a **`while`**-loop or another class. The second exception is the class body itself. It may only contain declarations, but no statements other than variable declarations (possibly combined with assignments). If a scope A contains another scope B, we call A the *enclosing scope*.
A declaration is visible within the scope in which it is defined. Furthermore it is visible within scopes contained in that scope. An entity which is defined in the current scope is called *local*. Local declarations must be distinguishable from each other. Classes, variables and constants are distinguishable if their names differ (that basically means that a scope may only contain one class, variable or constant declaration with the same name). Procedures and functions have a more complex distinction, called *overloading*, which is described in section 6.6.
Although variable names must not be declared twice within the same scope, names from enclosing scopes may be reused for declarations in inner scopes. If a scope contains a declaration using a name from an enclosing scope, the outer declaration will be *hidden* by the inner one. Then it is only accessible if the enclosing scope has a name (e.g. modules or classes) and thus, a qualified access is possible.
There is a different behavior considering the visibility of variables and other declarations (classes, procedures and functions). Variables are linearly visible. This means that a variable is only visible from its declaration to the end of the block. Procedures and classes are simultaneously visible, meaning that they

are visible in the entire block. Thus, procedure calls and class instantiations may appear before their declaration, as the following example shows:

```
Bool odd(Int num):
    if num = 0:
        return false
    else:
        return even(abs(num)-1)

Bool even(Int num):
    if num = 0:
        return true
    else:
        return odd(abs(num)-1)
```

6.6 Overload Resolution

In contrast to variable and class declarations, different procedures may have the same name, as long as they are distinguishable. Two procedures are distinguishable if they differ in number and type of their parameters. To determine which of the overloaded procedures is actually called, the *best-fit-principle* is used. Procedures with the desired parameter count are considered first. Then it is tested which parameter list fits best to the argument types of the call (according to the type system). If no suitable parameter signature is found, the procedure call can not be resolved. The argument types are incompatible if the type of an argument is not a subclass of the corresponding parameter type.

6.6.1 The Best-Fit-Principle

First, a list of possible candidates is generated, which contains all visible procedures with a matching name. The candidate list is sorted, so that declarations of the current scope occur first while declarations from enclosing scopes follow subsequently. Then the number of the candidate's parameters is checked against the call. All candidates with a parameter count different to the call are eliminated.
To find the declaration that fits best, the types of the parameters are rated according to their relationship to the actual argument types. If an argument and the correlating parameter have incompatible types, the declaration is removed from the candidate list. If the types are exactly equal, the distance of the particular parameter is 0. If the type is compatible, but not exactly equal (e.g. if a `Number` is expected and an `Int` is given), the distance is a number describing the relation between both types. If the given type directly inherits from the expected type, the distance is 1. The distance increases with the

distance between both types in the type hierarchy. If a type inherits from multiple superclasses, the shortest distance in the inheritance hierarchy is taken into account.
The distance of a declaration is the sum of the distance of its parameters. If an exact match is found (i.e. a declaration with a distance of 0), the algorithm stops. If all declarations are rated and no exact match is found, the declaration with the lowest distance is called. If there are two declarations with the same distance being the best fit, the particular case can not be resolved and the compilation fails due to ambiguity.

```
printValue(Object x):
    print("Object: ")
    print(x)

printValue(Number x):
    print("Number: ")
    print(x)

printValue(Int x):
    print("Integer: ")
    print(x)

printValue(7)       // >> Integer: 7
printValue(4.2)     // >> Number: 4.2
printValue("Hello") // >> Object: Hello

class ValuePrinter:
    initializer():
        printValue(7)       // >> Integer: 7
        self.printValue(7)  // >> Integer from method: 7

    printValue(Int x):
        print("Integer from method: ")
        print(x)

ValuePrinter vp := ValuePrinter()
```

In the example above, there are three overloaded versions of the procedure `printValue`. The procedure calls have different argument types. The first call provides an `Int` object. The procedure with the type `Procedure<Int>` has a distance of 0, since the argument type is exactly the parameter type. The procedure `Procedure<Number>` has a distance of 1, because `Int` inherits from `Number`. The `Procedure<Object>` has a distance of 3, because `Int` inherits from `Number` which inherits from `Ordered` which inherits from `Object`. In this case, the `Procedure<Int>` is an exact match. The second procedure call provides an argument of type `Float`. Since this type inherits from `Number`, the

second procedure is called. The third call provides a `String` argument. Since `String` is not related to either `Int` or `Number`, the only fitting declaration is the first one. The method `printValue` of the class `ValuePrinter` can not be confused with the other procedures, since it can only be accessed by using the **`self`** keyword from inside the class.
The following example shows possible ambiguities:

```
printTwoItems(Object x, String y):
    print("Object:" + x.toString())
    print("String:" + y)

printTwoItems(String x, Object y):
    print("String:" + x)
    print("Object:" + y.toString())

printTwoItems(1, "Hello")       // ok, 1st definition is used
printTwoItems("Hello", 1)       // ok, 2nd definition is used
printTwoItems(1, 1)             // incompatible types
printTwoItems("Hello", "Hello") // ambiguous call
```

The first two procedure calls work fine. The first one calls the first procedure `Procedure<Object, String>`. The second call matches best with the procedure of type `Procedure<String, Object>`. The third call is not resolvable. The argument type `Int` is not compatible with the parameter type `String`, so that neither of the two procedures can be called. For the fourth procedure call there are two possible candidates with the same distance – this call is ambiguous and can not be resolved.
For parameters with generic types (see section 7.4), the distance is calculated for each type parameter and summed up for the whole type. For a procedure expecting a `Map<Number, Object>` and a call providing a `HashMap<Int, Float>` the distance would be calculated as follows (this assumes the existence of a class `HashMap` which directly inherits from `Map`):

$$\begin{aligned} total\ dist &= dist(Map, HashMap) \\ &\quad + dist(Number, Int) \\ &\quad + dist(Object, Float) \\ &= 1 + 1 + 3 \\ &= 5 \end{aligned}$$

Overloading Methods

The overload mechanism for methods works similar to overloading regular functions or procedures. Method calls are always done qualified by their reciver object. Therefore, methods and regular functions and procedures can not be mixed up during overload resolution.

Due to inheritance (see section 7.3) there is a special case, which has to be considered when searching for overloaded methods. Let A be a class and B be a subclass of A. Additionally let v be a variable of type A, which contains an object of type B (which is allowed due to the subtyping rules). If a method m is looked up at that variable, the overloading mechanism can only find methods defined on class A, since it is not possible to statically determine the dynamic type of v. This leads to the fact that a potential better match in class B is not found. If B *overrides* the method m (i.e. the implementation of m in B has the same signature as in A), the overridden implementation is called at runtime, due to dynamic dispatch. The following example illustrates this behavior:

```
class Person:
    // ...
class Student inherits Person:
    // ...

class Library:
    register(Person p):
        print("registered a new person!")
class UniversityLibrary inherits Library:
    register(Student p):
        print("registered a new student!")

Library lib := UniversityLibrary()
lib.register(Student())             // prints "registered a new person!"
```

6.6.2 Default Parameters and Overloading

Default parameters (as introduced in section 6.2) are syntactic sugar for overloading. If a parameter is initialized in the parameter list (i.e. it has a default value), this will generate multiple overloaded versions of the procedure, with and without the initialized parameter. The procedure `drawCircle` in the following example has one mandatory and one optional parameter:

```
drawCircle(Point center, Int radius := 1):
    algorithms.bresenhamCircle(center, radius)
```

If the procedure is called without a second argument, a circle with the radius 1 is drawn. The two implicitly generated procedures are shown in the example below:

```
drawCircle(Point center, Int radius):
    algorithms.bresenhamCircle(center, radius)

drawCircle(Point center):
    drawCircle(center, 1)
```

7 Classes

Every object is an instance of a *Class*. Classes represent the internal structure of an object, as well as its static behavior. Every class defines a type and every type is defined by a class.

7.1 The Structure of Classes

Features of a class consist of method declarations, as well as variable and constant declarations. All features of a class are *simultaneously visible* (see section 6.5). Thus, inside a class scope a feature does not necessarily need to be defined before it is used.

7.1.1 Access

Accessing a feature of an object can be achieved in two ways: Either by using the *regular access* (denoted with a '.') or by using the *silent dynamic access* (*"SDA"*, denoted with '->'). Classes may define more than one method of the same name that only differ in number and/or type of its formal parameters (also referred to as *overloading*, see section 6.6). To avoid ambiguities between class features, local declarations in methods and declarations from enclosing scopes, access to features must always be qualified. A qualified access is introduced by the name of the containing namespace (in the most cases the containing object's identifier), followed by the access symbol (either '.' or '->') and the name of the object accessed. By specifying the receiver object, the scope is set for the lookup of the feature. If a feature of a class is accessed from inside the class, the qualification has to be done by using the keyword **self**.
Depending on the context of the access, features may not be accessible because of visibility restrictions (see subsection 6.4.1). If the requested feature exists and is available, accessing it returns a reference to that object. Otherwise a compiler error will occur, or, in case of a dynamic access, an *AccessException* will be thrown at runtime.

Regular Access

In general, accessing an object's feature is done by using the '.' notation. Considering an object o with a feature f, accessing f has to be qualified by $o.f$.

Silent Dynamic Access

Monty allows a second way of accessing object features. Every object owns an attribute of type $Map < String, Object >$, which is used to implement *dynamic attributes*. Objects can be extended at runtime by defining such dynamic attributes, which are mappings from names to objects. They are defined and accessed by using the *silent dynamic access* `'->'` (also referred to as *SDA*). If the SDA occurs on the left side of an assignment, either a new dynamic attribute is defined, or an existing feature or dynamic attribute is overridden). Consider the following creation of a feature of name f and value v an object o:

```
o->f := v
```

As a result of that statement, the map of dynamic attributes is introduced a new mapping from key f to value v. The new dynamic feature can only be accessed by using SDA instead of the regular access: $o\text{->}f$.
Unless the dynamic attribute is undefined, a silent dynamic access will yield the associated value of the given name, located inside the receiver object's map of dynamic attributes. If there already is some regular feature with the same identifier defined on the same object, the silent dynamic access will return the regular one, since its priority is higher than any dynamic attribute. If the dynamic attribute is undefined, a read access will result in an *AccessException*.

7.2 Instantiation

Instantiating a class means to create an object, based on the structure and behavior defined by the class. The type of the object is the class itself. To interact with the object, its attributes and methods can be directly addressed, as long as their access is not restricted by an access modifier.
Instantiating a class is done by calling the class as a function. The return value is the new instance. The instantiation process consists of the two sub-processes *construction* and *initialization*.

7.2.1 Construction

Creating an object starts with allocating the memory that is needed to store the new object in. The address to the allocated memory area is then bound to some variable identifier as a reference to the new object. This process is called *construction*.

```
Cls c := Cls()
```

Calling a class in the above way triggers the construction of an object of type *Cls*. Memory for the new object will be allocated and the identifier c is bound to its address. After that, the initialization process starts.

7.2.2 Initialization

After an object has been constructed, its attributes need to be initialized. Initializing an object can be achieved by using *initializers*.
Initializers are special methods of a class. They are defined as procedures and can be overloaded. Usually initializers are not called explicitly, but as a part of the instantiation of a class. Every class implicitly includes a *default initializer*, which does not have any parameters. Default initializers evaluate all initializing declarations inside the body. The initialization process will always call the default initializer of a class first. After that, specific initializer calls may follow. Consider the following example class:

```
class InitializerDemo:
  ~ Int x := 13
  ~ Int y := 2
  + Int z

  + initialize():
    z := 013_2
```

The default initializer of the class will initialize the attributes x and y, binding the values 13 and 2 on their respective attributes. The custom initializer only needs to consider the initialization of attribute z, since x and y have already been initialized.
User-defined initializers may require parameters. The given parameter values can be assigned to attributes of the class or be used to determine the specific initialization value of some attribute. By providing parameterizable initializers, objects can be initialized in different ways, based on the number and type of arguments passed to the constructor call.
In order to support the co-existence of multiple initializers which all have the same name but differ in the number and type of their parameters, *Monty* heavily relies on *overloading*.

7.2.3 Abstract Classes

Abstract classes are specified with the keyword *abstract* in front of the *class* keyword of a classes declaration. They can not be instantiated directly, but may have initializers, which can be called by subclasses.
Methods of an abstract class may be themselves abstract. If a class contains at least one abstract method, it has to be defined as an abstract class. However, abstract classes may not contain any abstract methods at all and still be defined as abstract classes.

7.3 Inheritance

As introduced in section 6.4, a class declaration may include the *inherits* keyword, followed by a various number of *parent classes*. By the use of that kind of declaration, all features from each parent class become features of the new class. Private features are not visible within derived classes. Features of a class which have been derived from a parent class will be referred to as *inherited features*.
The default initializer of the inherited class will automatically be called during initialization. Initializers of parent classes may be called by other initializers. However, an inherited initializer can not be used to initialize a derived class. Instead, a new initializer has to be defined in the derived class, which calls the parent initializer.
If the parent class is abstract, all abstract inherited features must be implemented. Otherwise the new class is abstract itself and has to be denoted as such.

7.3.1 Parent Initializers

Accessing features from one or more parent classes can be achieved only by using the *parent*-keyword:

---

parentExpression ::= **`parent`** '(' *type* ')' '.' *expression*

Vorschlag: Parent raus, stattdessen `(self as ParentClass).feature`

Evaluating expressions of the above form by evaluating the qualification-part of the access, resulting in a reference on a parent class. Using this reference, the method defined by the method name can be located inside the set of methods, inherited from the identified parent class.
Consider a class C that inherits from parent classes $S_1, \ldots, S_k$ (in that order). Instantiating class C constructs C as expected. However, the initialization process calls the default initializer of class C. Since C inherits from super classes, it's default initializer will call the default initializers of $S_1, \ldots, S_k$ (in that order), before executing it's own code. After that, user-defined initializers may be called. Explicitly calling parent initializers is possible by using the parent-function inside an initializer. Since classes may inherit from more than one class, there may be more than one parent initializer to call.

7.3.2 Multiple Inheritance

Allowing a class to inherit from more than one parent class may lead to ambiguity in the determination of a feature's definition. Consider some class D that inherits from classes B and C that each inherit from a class A. Such an inheritance structure will create a diamond-like shape in a class diagram. Class A may contain some feature f which is inherited by B and C. The problem to

determine, whether class D inherits feature f from B, from C, or from both, is referred to as the *diamond problem*.
Consider a class C that inherits from parent classes $S_1, \ldots, S_n$ (in that order). The inheritance process of *Monty* can be summarized in the following steps:

1. Inherit all non-private features of the first parent class S_1.
2. Inherit all non-private features of S_k which have not already been inherited from $S_1, \ldots, S_{k-1}$.

By following the above rules, every feature is inherited once, preferring features from the left-most parent classes. Non-inherited features are not lost, though. They have to be accessed by using the `parent` keyword.

7.3.3 Subtyping

Every class defines a type. Whenever an expression of a certain type T is expected, also an expression of a *subtype* S of T can be used. Subtypes are defined according to the following rules:

1. Class C is a subtype of itself.
2. If a class C inherits from classes $S_1, \ldots, S_k$, then C is a subtype of $S_1, \ldots, S_k$. Every subtype of C is a subtype of $S_1, \ldots, S_k$ as well.

The concept of subtyping defines two different compatibilities of types. In general, every occurrence of a type C can be substituted by a subtype S_i of C.

7.3.4 Overriding

Overriding a method of some parent class means to redefine it inside a subclass. Calling an overridden method executes the block that is currently associated with the method's name. However, to successfully override a method, there are some preliminaries to fulfill:

- The number, type and order of the overridden and overriding parameter list have to be equal.
- When overriding functions, the original and new return types have to be equal.

Overriding a method in a class has an impact on all its subclasses. If a method is defined in a class A and overridden in a class B, all subclasses of B inherit the overridden method, whereas all other subclasses of A inherit the original method. Redefining attributes is *not* possible.

Late Binding

Consider a class C with a non-private method m and subclasses $S_1, \ldots, S_k$ that inherit from C with the classes $S_1, \ldots, S_l$ ($l < k$) redefining m. By the rules of subtyping any object of type $S_1, \ldots, S_k$ can be used wherever an object of type C is expected. If a program includes the access of method m on an object of type C, it is not possible to determine, whether the object is of type $C, S_1, \ldots, S_{k-1}$ or S_k.
Resolving the exact type is only possible at runtime. If the type is determined, its implementation of the method m will be bound to the identifier m. This process is referred to as *late binding*.

7.4 Generics

In *Monty*, the term "Generics" refers to *generic classes*. A class is called *generic*, if its declaration includes at least one type parameter (see section 6.4 for the syntax of class declarations).
The type parameter is a placeholder for a type. When instantiating a generic class, every generic parameter is replaced by a concrete type. Whether the instantiation of a generic class is possible with specific types can be checked at compile time. Such a check is necessary to validate that the methods used inside the class are compatible with the concrete types.
Consider the following declarations of array objects:

```
Array<Int> numbers := [1, 2, 3, 4]          // array of integers, size 4
Array<String> names := Array<String>(25, "") // array of strings, size 25
```

The generic parameter from the class *Array* is resolved to the concrete types *Int* and *String*. Two generics classes are considered compatible, if the following condition holds: Consider a class C and a generic type parameter P. If $C\langle P\rangle$ is a subtype of $S\langle Q\rangle$ if and only if C is a subtype of S and $P = Q$. This behavior is also called *invariance*, since the generic type parameters have to stay invariant in the inheritance graph.

7.5 Lifetime

When a class has been instantiated, an object will be stored in memory. The created object stays available in memory ("alive"), as long as any active reference on that specific instance exists. Keeping an object alive means, keeping all of its attributes alive, too.
Objects that have no further use inside a system will be *garbage collected*, meaning an instance's allocated memory will be released and all instance-related information will be overwritten.

8 Modules and Packages

Modules and *Packages* encapsulate classes. Both can be used as compilation units. While a module is a *.monty* file, a package is a container for modules.

8.1 Modules

A *Module* is a file. Anything contained in this file is part of that module. The name of the file (excluding the *.monty* file extension) is the name of the module and thus must be a valid identifier (see section 2.3). Modules may contain functions, procedures, classes and even single statements. They have to be imported before any other declaration or statement (see section 8.3 for more information on imports).

---

moduleDeclaration ::= { *importStatement* }
{ [*accessModifier*] *declaration* | *statement* }

Every module defines a scope. Elements in that scope can be accessed via the module's name. All statements inside a module, which are not part of a declaration are executed when the module is first imported (also referred to as *initialization code*). The declarations inside the module can be accessed like object features, using the '.' access (subsection 7.1.1). Module features also may have visibility specifiers (subsection 6.4.1), where private and protected basically have the same effect for module features since modules do not support inheritance.

8.2 Packages

A package is a directory which contains *Monty* files or other packages. All *.monty* files contained in that directory (i.e. the specific modules contained therein) are part of that particular package. The modules inside a package can be accessed like object features. Packages may not only contain modules but also other packages which then are features of their containing package, respectively. A package can also have initialization statements (in analogy to the *initialization code* of modules, see section 8.3).

8.3 Import Mechanisms

Modules may import other modules or packages, as well as their respective features using the import statement:

```
importStatement ::= 'import' Identifier { '.' Identifier }
                    [ '.'  (ClassIdentifier | ConstantIdentifier) ]
                    [ 'as' ( Identifier
                           | ClassIdentifier
                           | ConstantIdentifier) ]
```

The syntax for imports allows three different ways to import a module or package:

```
import guilib
import guilib as gui
import guilib.window as win
```

The first line imports *guilib*, which can be either a module or a package. It can be referenced by using the name *guilib*. The second line also imports *guilib*. Using the *as* keyword, it is available using the name *gui* instead of the module's original name *guilib*. The third line imports the feature (this might also be a module or package) *window* of the module *guilib* into the current module. It is accessable via the name *win*. This can be used to shorten the access expressions. All names of the import expression must reference either modules or packages except the last one. In this case, *window* could be either a module or a subpackage of the package *guilib*, or it could be a feature of a module *guilib*.

When a module or a package is imported, first it is searched inside the directory of the importing module. If no module or package with the required name is found inside that directory, the paths inside the environment variable *MONTY_PATH* are searched. If a package or a module with the same name exist inside the same directory, it is imported.

If a module contains *initialization code* (see section section 8.1), this code is executed when the module is imported the first time. This leads to the effect that the code of the importing module is executed last. Just like modules, packages may also have initialization code, which must be located inside the special module *_package.monty*. If no such file exists, all modules inside a package will be imported in alphabetical order and thus be available as features of the package. The only exception are modules and packages which start with an underscore. Those can only be used from inside their containing package. If there is a file *_package.monty* that module is the entry point for the package. All names which are visible inside this module are features of the package. In particular this applies for modules which are imported inside the package file. The initialization code inside the *_package.monty* file is executed when the package is imported.

Cyclic imports of modules and packages are possible. However, their initialization code may not mutually access the other module/package, since this would cause an endless recursion. The declarations, however, may use cycli-

cally imported declarations.

chicken.monty

```
import egg

greet():
    print("I am the chicken")
    print("I am first!")

egg.greet()
greet()
```

egg.monty

```
import chicken

greet():
    print("I am the egg")
    print("I am first!")

chicken.greet()
greet()
```

The above example won't work, because the initialization code of each module uses code from the other module, which again has initialization code which uses the own module. However, the example works if the modules would look as follows:

chicken.monty

```
import egg

greet():
    print("I am the chicken")
    print("I am first!")

greet()
```

egg.monty

```
import chicken

greet():
    print("I am the egg")
    print("I am first!")

greet()
```

In this case the output depends on which module is executed. If *chicken.monty* is executed, the first output is *"I am the egg..."*. If *egg.monty* is executed, the first output is *"I am the chicken..."*. This example works, because the initialization code of each module does not use the other one.

8.4 The Monty Core Library

The core library contains the basic data types (described in chapter 3), as well as some core functionality like the *print* procedure and basic exception types. This library is a built-in package which is located inside the directory of the monty installation. Every module implicitly contains the following import statement:

```
import corelib
```

Additionally the features of this package are directly accessible inside the importing module, so that all core library features are available without an explicit import statement and without a qualified package feature access.

9 Rationale

9.1 Syntax

The syntax is inspired by *Python*[10] and *Java*[3]. The use of those two languages as a source for inspiration was justified by their popularity and in case of *Python* by its conciseness.
One goal of *Monty* is to provide a compact and simple syntax, which allows to write structured and readable code. To fulfill this goal, we try to call every feature of *Monty* by its intention. For example, the keyword for handling exceptions is not *except* or *catch* like in *Python* or *Java*, but `handle` (subsection 5.8.2).

Indentation. The most apparent *Python* heritage is the use of indentation (section 5.2) to indicate blocks rather than curly braces or other symbols. We endeavor to increase code readability and enforce consistently structured code. The only difference between the indentation rules in *Monty* and *Python* is that *Monty* not only allows expressions to be line-wrapped, if enclosed in parenthesis, but also if the second operand for a binary operator is wrapped to a new line.

Identifiers. Although *Monty* supports Unicode characters in string literals, *identifiers* may only consist of ASCII characters (section 2.3) in order to increase the readability. The reasoning behind this restriction is the fact that Unicode identifiers will exclude certain user groups due to charset or font issues which may result in letters not being displayed correctly.

Identifier Styles. *Monty* requires special notations for class, constant and variable identifiers (see section 2.3). On the one hand this limits the freedom of naming, on the other hand it reduces the amount of keywords (such as *const* to indicate constants). Another consequence is a higher readability as the kind of an identifier is more obvious.

Integers (subsection 2.7.2). *Monty* supports integer literals in different numeric systems. By default, all numerical systems with a base ranging from 2 to 36 are available. Restricting the allowed bases to a maximum of 36 is motivated by the fact that there are only 26 letters (A-Z, where the letters are case insensitive) plus 10 digits (0-9) available.
Integer literals using bases other than 10 need a leading zero digit in order to distinguish between identifiers and numbers.

Float (subsection 2.7.3). Although other languages accept floating point literals with a leading decimal sign (like `.4`), *Monty* requires a leading `0`. Again, this is a decision in favor of readability.

Assignments. Languages like *C++* [9] or *Java* [3] use the equality sign ("=") for assignments. This, however, collides with the equality operator. Often the "==" symbol is used as a solution. This is often confusing. Therefore, *Monty* uses ":=" for assignments (see section 5.4). This is not only the common notation in mathematics, it also makes the "=" operator available to be used in equality checks (which is also the correct notation in mathematics).

Operator Symbols. Unlike other languages (such as *Ruby* [2]), *Monty* restricts operators symbols to a fixed set. This has the advantage that the precedence and semantics for each operator is clearly defined. For example, the semantics of a plus operator ("+") is obvious from the context, while the semantics of symbols like "★" is not generally known. Also, if other symbols were allowed, their syntax would not be obvious for the reader, especially in question of infix or prefix notation.

Visibility Specifiers. The visibility of an object's features is controlled by visibility specifiers. The syntax of those specifiers corresponds to the standard notation for UML class diagrams [8] (see subsection 6.4.1). This notation is commonly used as it enables programmers to get a quick overview over the class hierarchy but it may also be used as a means for documentation and specification.

No List Literal. There is no literal for *lists* in *Monty*. The literals for *Array* and *Map* (section 4.2) both base on the squared brackets. To create a literal for lists, we would have to introduce a new kind of brackets. To keep the amount of delimiters (section 2.3) small, we decided against it. Following the ideal of a unique way to achieve a goal, list objects can only be created using an array literal (e.g. `List<Int>([1,2,3])`).

`break` and `skip`. Although other languages like *C* call the same keywords `break` and `continue`, *Monty* tries to name everything by its purpose. Therefore the keyword, known as `continue`, which does not continue the current loop iteration, but instead skips the rest of it, is called **`skip`** (see section 5.6).

`parent` Instead of *super*. Since one goal of *Monty*'s syntax is to name everything by its purpose, the function to access the features of a parent-class, is called **`parent`** (see section 7.3) instead of *super*-function like in *Java*. To access the features of the parent-class, it is also possible to use the class-name instead of the parent-function. However, while designing the language we decided only to grant access to the parent-class by using the function. This is a decision in favor of readability.

No Multi Line Comments. In *Monty* only single line comments exist. We decided against multi line comments, in favor of readability. As with multi line comments a block in one line or in one expression can be commented out, the readability could be suffering.

Tabulator As One Character. Since *Monty* uses *indentation* to identify *blocks*, beside the space-character also the *Tab*-character is important. The sequence of *LayoutCharacter* (section 2.2), which defines the offset, is not only identified by its length, but also by its order. For example, even if the sequences [TAB, SPACE, TAB] and [TAB, TAB, SPACE] look the same in the editor, they are not the same.

9.2 Semantics

9.2.1 General

Expressions and Statements. *Monty* is not an expression-oriented language. This means that the language distinguishes between expressions and statements. While expressions evaluate to an object, statements do not. One issue of expression-oriented languages is that even control structures would need to have a return type. Most expression-oriented languages introduce types like *void* or *None* in order to deal with the arising problems. However, while designing the language we decided not to integrate such a type. In analogy, variables cannot have a *null* value contrary to *Java*[3]. This decision was made as we regard null values as dangerous which also confirms Tony Hoare in his statement about his *"Billion Dollar Mistake"*[4].

Garbage Collection. In a modern programming language it should not be a necessity to be concerned about allocating and freeing memory, particularly if the goal of the language is to fit the gap between scripting and programming languages. This is why *Monty* advocates the use of a *garbage collector* (see section 7.5) in its implementation.

Destructors. Destructors are used to free the memory allocated by an object when its lifetime ends. *Monty* works without explicit destructors. Due to the use of a *garbage collector* (see section 7.5) the moment of an object's destruction is unknown. Consequently, this also applies for the destructor call. For this reason, memory management is reserved for the garbage collector and there is no possibility of custom-defined destructors.

Overloading. Like most other programming languages *Monty* provides a *context-free overloading* method (see section 6.6). This means that the signatures of overloaded functions need to differ in type or count of their parameters. The return type is not considered (in contrast to *context-sensitive overloading*). We concluded that context-free overloading is less prone to ambiguity.

The Default Visibility. *Monty* supports the four visibility specifiers *public*, *protected*, *private* and *package*. When the visibility specifier is omitted, the visibility used is *package* which was nominated as the default visibility (see subsection 6.4.1). This allows to use the feature outside its class but also restricts the visibility to the code of the same package, wherefore it is the most suitable default visibility.

Module Headers. *Monty* uses the file-system to organize packages. Therefore the name of a *module* is also the filename (see section 8.1). Since the name of a module is bounded to its filename, we decided that a *module header* inside a *module* is unnecessary.

9.2.2 Data Model

Array and Sequence In *Monty* every data collection inherits from the abstract class *Collection*. Therefore, in many situations the programmer does not need to distinguish between an *Array* and other sequence types like *List*.
The disadvantage of treating array types as regular classes is that the runtime performance may suffer. On the other hand, it is more valuable to ensure that everything is an object and that all sequence types have a common interface.

Operator. In *Monty* everything is an object, so are operators (see section 4.4). Since operators are defined for particular classes (types), it is obvious to declare them as instance methods. This way, operators are only syntactic sugar for method calls. Handling operators as common methods has the advantage that the overloading of operators is already given by regular overloading.

Voiding. *Voiding* is a term from *ALGOL 68* [6, 5.1.7] which describes the functionality to allow unwanted results to be thrown away. For example using a function as a procedure call. *Monty* supports *voiding*. This is a consequence of the data model in combination with *subtyping* (see subsection 7.3.3). Since *function* inherits from *procedure* (see section 3.4), every *procedure call* can invoke a function.

Type-Erasure. *Monty* does not support *type-erasure*. This means, generic type parameters are not removed at runtime. Therefore the classes List<Int> is not the same as List<Char>. As a result that the *type system* gets stronger.

9.2.3 Multiple Inheritance

Motivation. Multiple inheritance is often discredited to cause more problems than it solves. A common argument is the *diamond problem* [5]. Besides that, it is the simplest and clearest inheritance strategy. Almost every language which is restricted to single inheritance uses different strategies to overcome the resulting limitations. Examples are *interfaces*, *mixins* and *traits* – all with

the same goal: inheriting properties and methods from more than one parent class. This is the motivation for *Monty* supporting multiple inheritance (see subsection 7.3.2).

Implicit Inheritance Priority. One goal of *Monty* is to provide a language for rapid development. By using an implicit strategy to handle name clashes in multiple inheritance cases, this goal is a little bit closer. Other languages like *C++* [9] forbid using the same feature names in two different parent classes by default. *Monty* does not. By using the order of inherited classes (see subsection 7.3.2), the name problem can definitely be solved at the time the class is created. Also every shadowed parent class feature is still available using the *parent* function.

Interfaces. Many object-oriented languages with single inheritance are using interfaces to overcome their inheritance limitations. Interfaces help classes to share the same signature but they do not reduce redundant code because functionality can not be inherited from interfaces. If a language supports multiple inheritance like *Monty* (see subsection 7.3.2), interfaces can be realized as abstract classes.

Traits and Mixins Many modern scripting languages (like *Ruby* [2] and *PHP* [7]) use traits or mixins to overcome the limitations of single inheritance. This allows them to compose functionality into classes, regardless of how the inheritance hierarchy is set up. It is a quick way to abstract logic into separate classes but it does not support clear object-orientation. Often the desired behavior should rather be achieved using aggregation.

Bibliography

[1] IEEE standard for floating-point arithmetic. *IEEE Std 754-2008*, pages 1–70, Aug 2008.

[2] David Flanagan and Yukihiro Matsumoto. *The Ruby Programming Language*. O'Reilly, first edition, 2008.

[3] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, The (3rd Edition)*. Addison-Wesley Professional, 2005.

[4] Tony Hoare. Null references: The billion dollar mistake. In *QCon*, 2009.

[5] Robert C. Martin. Java and C++: A critical comparison. 1997.

[6] A.D. McGettrick. *ALGOL 68: A First and Second Course*. Cambridge computer science texts. Cambridge University Press, 1978.

[7] PHP. Official PHP5 online documentation on traits. 2014.

[8] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language reference manual: [covers UML2.0]*. The Addison-Wesley object technology series. Addison-Wesley, Boston, 2. ed edition, 2005.

[9] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 2000.

[10] Guido van Rossum and Fred L. Drake. The Python Language Reference. Release 3.4.1, May 2014.

[11] David A. Watt. The Design of Monty: a programming/scripting language. *Electr. Notes Theor. Comput. Sci.*, 141(4):5–28, 2005.

A Syntax Specification

Comment ::= '//' {*EveryCharButNewline*} ↵

LayoutCharacter ::= *Space* | ↵
LayoutSequence ::= *LayoutCharacter* { *LayoutCharacter* }

Space ::= '\t' | ' '
↵::= ['\r'] '\n' { ['\r'] '\n' }

Identifier ::= { '_' } *LowercaseLetter* { *IdentifierSymbol* }

ConstantIdentifier ::= { '_' } *UppercaseLetter*
{ *UppercaseLetter* | *Digit* | '_' }

ClassIdentifier ::= *UppercaseLetter* { *IdentifierSymbol* }
LowercaseLetter { *IdentifierSymbol* }

IdentifierSymbol ::= *LowercaseLetter* | *UppercaseLetter* | *Digit* | '_'

Digit ::= '0' | ... | '9'
LowercaseLetter ::= 'a' | ... | 'z'
UppercaseLetter ::= 'A' | ... | 'Z'

literal ::= *BoolLiteral*
| *IntLiteral*
| *FloatLiteral*
| *CharacterLiteral*
| *StringLiteral*

BoolLiteral ::= **true** | **false**

IntExponent ::= ('e' | 'E') ['+'] *Digit* { *Digit* }

Base ::= ['0' | '1' | '2'] *Digit*
| '3' ('0' | ... | '6')

IntLetter ::= 'A' | ... | 'F'
| 'a' | ... | 'f'

IntLiteral ::= (*Digit* { *Digit* } [*IntExponent*])

```
            | ( '0' { IntLetter | Digit } '_' Base )

FloatingLiteral ::= Digit { Digit } '.' Digit { Digit }
                    [ FloatExponent ]
FloatExponent ::= ( 'e' | 'E' ) [ '−' | '+' ] Digit { Digit }

CharacterLiteral ::= '\'' Character '\''
Character ::= UnicodeCharacter | CharacterEscapeSequence

StringLiteral ::= '"'
                  { CharacterEscapeSequence | UnicodeStringCharacter }
                  '"'


expression ::= literal
            | '(' expression ')'
            | name
            | unaryOperator expression
            | expression binaryOperator expression
            | conditionalExpression
            | aggregate
            | functionCall

unaryOperator ::= not | '−'
binaryOperator ::= '+' | '−' | '*' | '/' | '%' | '^' | '=' |
                   '!=' | '<' | '>' | '<=' | '>=' | '−>' | in
operator ::= unaryOperator | binaryOperator
binaryOperatorLike ::= as | is | and | or

aggregate ::= map
            | array


map ::= '[' [ keyValuePair { ',' keyValuePair } ] ']'
keyValuePair ::= expression ':' expression

array ::= '[' [ expression { ',' expression } ] ']'
        | range

range ::= '[' expression '..' expression ']'

name ::= ClassIdentifier | variable

variable ::= variableName
            | expression '.' variableName      // regular access
            | expression '−>' variableName     // silent dynamic access
            | expression '[' expression ']'    // subscription
```

```
variableName := Identifier | ConstantIdentifier | self

conditionalExpression ::= expression 'if' expression 'else' expression

block ::= ':' ↵
          ⤳
             ( 'pass' ↵| blockContent { blockContent } )
          ⤺

blockContent ::= statement { ↵}
               | localDeclaration { ↵}

assignment ::= name ':=' expression ↵

compoundAssignment ::= name compoundSymbol expression ↵

compoundSymbol ::= '+=' | '−=' | '*=' | '/=' | '%=' | '^='

conditionalStatement ::= 'if' expression block
                         { 'elif' expression block }
                         [ 'else' block ]

loopStatement ::= 'while' expression block

procedureCall ::= name '(' [ expression { ',' expression } ] ')'

raiseStatement ::= 'raise' [ expression ]

handleStatement ::= 'try' block
                    handler { handler }

handler ::= 'handle' [ ClassIdentifier identifier {',' ClassIdentifier identifier } ]
            block

declaration ::= localDeclaration
              | classDeclaration

localDeclaration ::= variableDeclaration
                   | constantDeclaration
                   | procedureDeclaration
                   | functionDeclaration

variableDeclaration ::= type Identifier [ ':=' expression ] ↵
constantDeclaration ::= type ConstantIdentifier [ ':=' expression ] ↵
```

```
type ::= { name '.' } ClassIdentifier [ '<' type { ',' type } '>' ]

procedureDeclaration ::= Identifier '(' [ parameterList ] ')' block

parameterList ::= ( parameter { ',' parameter } { ',' defaultParameter } )
                | ( defaultParameter { ',' defaultParameter } )

parameter ::= type Identifier
defaultParameter ::= type Identifier ':=' expression

functionDeclaration ::= type Identifier '(' [ parameterList ] ')' block

classDeclaration ::= ['abstract'] 'class' ClassIdentifier
                     [ '<' ClassIdentifier { ',' ClassIdentifier } '>' ]
                     ['inherits' typeList]
                     classBody

typeList ::= { name '.' } ClassIdentifier
           { ',' { name '.' } ClassIdentifier }

classBody ::= ':' ↵
              ⇀
                  (memberDeclaration { memberDeclaration } | 'pass')
              ↽

memberDeclaration ::= [accessModifier] ( variableDeclaration
                                       | constantDeclaration
                                       | methodDeclaration )

accessModifier ::= '+' | '~' | '#' | '−'

methodDeclaration ::= functionDeclaration
                    | procedureDeclaration
                    | abstractMethodDeclaration

abstractMethodDeclaration ::= 'abstract' [ type ] Identifier
                              '(' [parameterList] ')' ↵

moduleDeclaration ::= { importStatement }
                      { [accessModifier] declaration | statement }

importStatement ::= 'import' Identifier { '.' Identifier }
                    [ '.' (ClassIdentifier | ConstantIdentifier) ]
                    [ 'as' (Identifier | ClassIdentifier | ConstantIdentifier) ]
```

B Predefined Classes