

Declarative Programming

Philipps University Marburg

April 21, 2020

based on the script by Prof. Dr. Klaus Ostermann, with contributions by Jonathan Brachthäuser, Yufei Cai, Yi Dai, Tillmann Rendel, Prof. Dr. Christoph Bockisch

Large parts of this script are based on the book "How To Design Programs" by M. Felleisen, R.B. Findler, M. Flatt and S. Krishnamurthi.

Contents

1 Programming with expressions

1.1 Programming with arithmetic expressions

Each of you knows how to add numbers, make coffee or assemble a cupboard at a Swedish furniture store. The sequence of steps you follow to do this is called *an algorithm*, and you know how to execute such an algorithm. In this course, we will reverse the roles: You will program the algorithm, and the computer will execute it. A formal language in which such algorithms can be formulated is called a programming *language*. The programming language that we will use initially is called *BSL*. BSL stands for "Beginning Student Language". We will use *DrRacket* to edit and execute the BSL programs. DrRacket can be downloaded from the URL <http://racket-lang.org/>. Please set the language to "How To Design Programs - Beginners". The best way to follow this text is to start Dr-Racket in parallel and always program along.

Many simple algorithms are already predefined in a programming language, e.g. arithmetic with numbers. We can set "tasks" by asking DrRacket a question to which DrRacket then gives us the answer in the output window. For example, we can ask the question

```
(+ 1 1)
```

in the *definition area* (the upper part of the DrRacket interface) - we receive a response in the *interaction area* (the area below the definition area) when this program is evaluated ("Start" button) 2. You write and edit your programs in the definition area. As soon as you change something here, the "Save" button appears, with which you can save the definitions in a file. The result of a program execution is displayed in the interaction area; expressions can also be entered here which are evaluated immediately but are not saved in the file.

We call the type of programs or questions such as `(+ 1 1)` *expressions*. In future, we will represent such question/answer interactions in such a way that we place the character in front of the question and the answer to the question in the next line:

```
> (+ 1 1)  
2
```

In the following, we call operations such as `+` *functions*. The operands such as `1` are called *arguments*. Here are some more examples of expressions with other functions:

```
> (+ 2 2)  
4  
> (* 3 3)  
9  
> (- 4 2)
```

```

2
> (/ 6 2)
3
> (sqr 3)
9
> (expt 2 3)
8
> (sin 0)
0

```

The arguments of these functions are always numbers, and the result is also a number. We can also use a number directly as an expression. For example:

```

> 5
5

```

DrRacket displays exactly the same number as the result. A number that is used directly in an expression is also called a number *literal*.

For some expressions, the computer cannot calculate the mathematically correct result. Instead, we get an approximation of the mathematically correct result. For example:

```

> (sqrt 2)
#i1.4142135623730951

> (cos
pi) #i-1.0

```

The *i* in the last result stands for "inexact". In BSL, you can see from this *i* whether a number is an exact result or only an approximate result.

Programs contain expressions. All the programs we have seen so far *are* expressions. Everyone of you knows expressions from mathematics. At this point, an expression in our programming language is either a number, or something that starts with a left parenthesis "(" and ends with a right parenthesis ")" ends with a right bracket ")". We refer to numbers as *atomic expressions* and expressions that start with a parenthesis as *compound expressions*. Later, other types of expressions can be added.

How can you add more than two numbers? There are two ways to do this: By nesting:

```

> (+ 2 (+ 3 4))
9

```

or by addition with more than two arguments:

Program
an expression that
is the sum of the
numbers 3, 5, 19,
and
32 calculated.

```
> (+ 2 3 4)
9
```

Whenever you want to use a function such as `+` or `sqrt` in BSL, write an opening bracket, followed by the name of the function, then a leading character (or line break) and then the arguments of the function, i.e. in un-

In this case, enter the numbers to which the function is to be applied.

You have seen from the example of nesting that compound expressions are also permitted as arguments. This nesting can be as deep as you like:

```
> (+ (* 5 5) (+ (* 3 (/ 12 4)) 4))
38
```

Program
an expression that
is the average of the
numbers 3, 5, 19
and 32 are calculated.

The result for such a nested expression is calculated in the same way as you would do it on a sheet of paper: If an argument is a compound expression, the result for this expression is calculated first. This sub-expression may itself be nested; in this case, this calculation rule is also applied to these sub-expressions (*recursive* application). If several arguments are compound expressions, they are evaluated in an unspecified order. The order is not fixed because the result does not depend on the order - more on this later.

In summary, programming at this point is the writing of mathematical expressions. Executing a program means calculating the value of the expressions it contains. Pressing "Start" causes the program to be executed in the definition area; the results of the execution are displayed in the action area.

Another practical tip: If you are reading this document with a web browser, all functions that appear in the example expressions should contain a hyperlink to their documentation. For example, the addition operator in the expression `(+ 5 7)` should contain such a hyperlink. Under these links you will also find an overview of the other operations that you can use.

1.2 Arithmetic with non-numerical values

If we could only write programs that process numbers, programming would be just as boring as mathematics ;-) Fortunately, there are many other types of values that we can calculate with in the same way as numbers, for example text, truth values, images, etc.

For each of these so-called *data types*, there are *constructors that can be* used to construct values of these data types, as well as *functions* that can be applied to values of this data type and that construct further values of the data type. Constructors for numerical values are, for example, `42` or `5.3` (i.e. the number *literals*; functions are, for example, `+` or `*`).

The constructors for text (also called *string* in the following) can be recognized by quotation marks. For example

"Concepts of programming languages"

is a string literal. A function on this data type is `string-append`, for example

```
> (string-append "Concepts of " "Programming  
languages") "Concepts of programming languages"
```

There are other functions on strings: to extract parts from a string, to reverse the order of letters, to convert to upper or lower case, etc. Together, these functions form the *arithmetic of the strings*.

The names of all these functions do not need to be memorized; if required, the available functions for numbers, strings and other data types can be looked up in the DrRacket help under: Help - How to Design Programs
Languages - Beginning Student - Pre-defined Functions

So far, we have only learned about functions where all arguments and the result must be of the same data type. For example, the `+` function only works with numbers, and the `string-append` function only works with strings. However, there are also functions that expect values of one data type as an argument, but return values of another data type as a result, for example the `string-length` function:

```
> (+ (string-length "Programming languages")  
5) 24
```

The result of `(string-length "programming languages")` is the number 19, which can be used as an argument for the `+` function as normal. You can therefore use functions that belong to different data types together in an expression. However, you must ensure that each function has arguments of the correct data type. There are even functions that expect arguments of different data types, for example

```
> (replicate 3  
"hi") "hihihi"
```

Finally, there are also functions that convert data types into each other, for example

```
> (number->string 42)  
"42"  
> (string->number "42")  
42
```

This example illustrates that 42 and "42", despite their similar appearance, are two very different expressions. To compare them, let's add two more expressions, namely

`(+ 21 21)` and `"(+ 21 21)"`.

Numbers	Strings
---------	---------

42	"42"
----	------

<code>(+ 21 21)</code>	<code>"(+ 21 21)"</code>
------------------------	--------------------------

Program
an expression that
generates the string
"The average is
...". Instead of
the three dots, the
average of the
numbers 3, 5, 19
and
32. Use the
expression that
calculates this
average as a sub-
expression.

The first expression, `42`, is a number; the evaluation of a number results in the number itself. ¹ The third expression, `(+ 21 21)`, is an expression which, when evaluated, also gives the value 42. Each occurrence of the expression `42` can be replaced in a program by the expression `(+ 21 21)` (and vice versa) without changing the meaning of the program.

The second expression, `"42"`, on the other hand, is a string, i.e. a sequence of digits that happens to correspond to the value 42 when interpreted in decimal notation. Accordingly, it is also pointless to add anything to `"42"`:

```
> (+ "42" 1)
+:: expects a number as 1st argument, given "42"
```

The last expression, `"(+ 21 21)"`, is also a sequence of characters, but it is not equivalent to `"42"`. So one cannot be replaced by the other without changing the meaning of the program, as this example illustrates:

```
> (string-length "42")
2

> (string-length "(+ 21 21)")
9
```

There are programming languages that support automatic conversions between numbers and strings that represent numbers. This does not change the fact that numbers and strings that can be read as numbers are very different things.

Now back to our introduction of the most important data types. Another important data type is truth values (Boolean values). The only constructors for this are the literals `#true` and `#false`. Functions on Boolean values are, for example, the propositional logic functions:

```
> (and #true
#true) #true
> (and #true
#false) #false
> (or #true
#false) #true
> (or #false
#false) #false
> (not
#false) #true
```

Boolean values are also frequently returned by comparison functions:

```
> (> 10 9)
#true
```

Do you know him? Question to the pregnant computer scientist: Will it be a boy or a girl? Answer: Yes!

¹ If we wanted to be very precise, we could still distinguish between the number literal `42` and the mathematical object 42; the former is only a notation (syntax) for the latter. However, we will define the meaning of programs purely syntactically, so this distinction is not relevant for us.

```
> (< -1 0)
#true
> (= 42 9)
#false
> (string=? "hello"
"world") #false
```

Of course, expressions can still be nested as desired, e.g. like this:

```
> (and (or (= (string-length "hello world") (string-
>number "11"))
        (string=? "hello world" "good morning")))
      (>= (+ (string-length "hello world") 60) 80))
#false
```

The last data type we will introduce today is images. In BSL, images are "normal" values, with associated arithmetic, i.e. functions on them. Existing images can be inserted directly into the program via copy&paste or via the "Insert - Image" menu. If you are viewing this document in the browser, you can copy and paste the



image of this rocket into your program. Just as the evaluation of a number produces the number itself, the evaluation of the image produces the image itself.





with other data types, a range of functions are also available for images. However, these functions must first be added to BSL by activating a "teach pack". Activate the HtDP/2e teach pack "image.ss" in DrRacket to experiment with the following examples yourself.

For example, the area of the image can be calculated using this expression:

```
> (* (image-width  (image-height ))
600
```

Instead of inserting existing images into the program, you can also construct new images:

```
> (circle 10 "solid" "red")

> (rectangle 30 20 "solid" "blue")

```

The arithmetic of images not only includes functions to construct images, but also to combine images in various ways:

Note in this example as the indentation of the text helps to understand which sub-expression is the argument of which function. In DrRacket, try out how the "Indent" or "Indent all" functions in the "Racket" menu change the indentation of your program.

Make sure to use the "image.ss" teachpack that belongs to HtDP/2e. It can be found in the DrRacket teachpack dialog in the middle column. Alternatively, you can add the instruction "(require 2htdp/image)" at the beginning of your file.

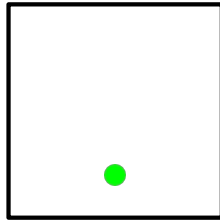
Program an expression that draws several circles around the same center point. Program an expression that draws a circle that is intersected by a diagonal line. Mark the intersections with small circles.


```
> (overlay (circle 5 "solid" "red")
         (rectangle 20 20 "solid" "blue"))
```



Use the BSL documentation (e.g. via the links in the browser version of this document) if you want to know what other functions are available on images and what parameters they expect. Two important functions you should know are `empty-scene` and `place-image`. The first creates a scene, a special rectangle in which images can be placed. The second function places an image in a scene:

```
> (place-image (circle 5 "solid" "green")
              50 80
              (empty-scene 100 100))
```



1.3 Appearance and dealing with mistakes

Different types of errors can occur when creating programs. It is important to know the classes and causes of these errors.

Syntax errors are an important type of error. An example of a program with a syntax error are the expressions `(+ 2 3(` or `(+ 2 3` or `(+ 2 (+ 3 4)`.

Syntax errors are checked and found by DrRacket before the program is executed; this check can also be initiated using the "Syntax check" button.

A syntax error occurs when a BSL program does not match the BSL grammar. Later we will define this grammar precisely; informally, a grammar is a set of rules about the structure of correct programs.

The grammar of the part of BSL you already know is very simple:

- A BSL program is a sequence of expressions.
- An expression is a number, an image, a Boolean value, a string or a function call.
- A function call has the form `(f a1 a2 ...)` where `f` is the name of a function and the arguments `a1, a2, ...` are expressions.

The following programs are all syntactically correct, but not all of them can be evaluated:

```

> (number->string "asdf")
number-string: expects a number, given "asdf"
> (string-length "asdf" "fdsa")
string-length: expects only 1 argument, but found 2
> (/ 1 0)
/: division by zero
> (string->number "asdf")
#false

```

Not every syntactically correct program has a meaning in BSL. *Meaning* in this case means that the program can be executed correctly and returns a value. The set of BSL programs that have a meaning is only a *subset* of the syntactically correct BSL programs.

The execution of the program `(number->string "asdf")` results in a *Runtime error*, an error that occurs while the program is running (in contrast to syntax errors, which are detected *before* the program is executed). If a runtime error occurs in BSL, program execution is aborted and an error message is output.

This error is an example of a *type error*: The function expects an argument to have a certain type, in this case 'number', but the argument actually has a different type, in this case 'string'.

Another error that can occur is that the number of specified arguments does not match the function. This error *occurs* in the program `(string-length "asdf" "fdsa")`.

Sometimes the data type of the argument is correct, but the argument does not 'fit' in some way. In the example `(/ 1 0)`, the division function expects numbers as arguments. The second argument is a number, but the execution results in an error message because division by zero is not defined.

Programming languages differ in the point in time at which such errors are found, for example before execution or during execution. In general, the earlier the better! - However, this additional security often comes at the cost of other restrictions, such as the fact that some correct programs can no longer be executed.

A different situation exists in the case `(string->number "asdf")`. The execution of the program results in the value `#false`. This value signals that the string passed does not represent a number. In this case, *no* runtime error occurs, but the execution is continued. The caller of `(string->number "asdf")` therefore has the option of querying whether the conversion was successful or not and continuing the program differently depending on this. The program is therefore well-defined from a BSL perspective. Alternatively, the `string->number` function could have been defined in such a way that it triggers a runtime error in this situation.

1.4 Comments

In addition to the actual program text, a program can also contain comments. Comments have no influence on the meaning of a program and only serve to make a program easier to read. Especially when programs become larger, comments can help to understand the program more quickly. Comments are introduced by a semicolon; everything in a line after a semicolon is a comment.

```
; Calculates the golden  
number (/ (+ 1 (sqrt 5)) 2)
```

It can also sometimes be helpful to "comment out" entire programs. Programs that are contained in comments are ignored by DrRacket and not evaluated.

```
(/ (+ 1 (sqrt 5)) 2)  
; (+ 42
```

In the example above, no result would be output as the calculations are in a comment. Incorrect parts of a program (such as the one in the second comment line) can be commented out so that the rest of the program can be executed in DrRacket.

We will say more about where, how and in what detail programs should be commented on later.

1.5 Meaning of BSL expressions

Let's summarize the current status: Programming is the writing of arithmetic expressions, where arithmetic includes numbers, strings, boolean values and images. Programs are syntactically correct if they have been constructed according to the rules from the previous section. Only syntactically correct programs (but not all) have a meaning in BSL. The meaning of an expression in BSL is a value, and this value is determined by the following evaluation rules:

- Numbers, strings, images and truth values are values. In the rest of this rule, we use variants of the letter *v* as placeholders for expressions that are values and variants of the letter *e* for any expressions (mnemonic: value = value, expression = *expression*).
- If the expression is already a value, its meaning is this value.
- If the expression has the form `(f e1 . . . en)`, where *f* is a function name and *e*₁, . . . , *e*_n are expressions, this expression is evaluated as follows:
 - If *e*₁, . . . , *e*_n are already values *v*₁, . . . , *v*_n and *f* is defined on the values *v*₁, . . . , *v*_n, then the value of the expression is the application of *f* to *v*₁, . . . , *v*_n

- If e_1, \dots, e_n are already values v_1, \dots, v_n but f is *not* defined on the values v_1, \dots, v_n , then the evaluation is aborted with an appropriate error message.
- If at least one of the arguments is not yet a value, the values of all arguments are determined by applying the same evaluation rules, so that the previous evaluation rule is then applicable.

We can understand these rules as instructions for the step-by-step *reduction* of programs. We write $e \rightarrow e'$ to say that e can be reduced to e' in one step. Values cannot be reduced. The reduction is like is defined as follows:


- If the expression has the form $(f \ v_1 \ \dots \ v_n)$ and the application of f to v_1, \dots, v_n gives the value v , then $(f \ v_1 \ \dots \ v_n) \rightarrow v$.
- If an expression e_1 contains a sub-expression e_2 in an *evaluation position* that can be reduced, i.e. $e_2 \rightarrow e_2'$, then $e_1 \rightarrow e_1'$ applies, whereby e_1' is created from e_1 by replacing e_2 with e_2' .

We call the last rule the *congruence rule*. In the part of the language that you have already become familiar with, *all* positions of sub-expressions are evaluation positions. Since so far only function calls have sub-expressions, the following special case of the congruence rule applies in this case: case $e_i \rightarrow e_i'$, then $(f \ e_1 \ \dots \ e_n) \rightarrow (f \ e_1 \ \dots \ e_{i-1} \ e_i' \ e_{i+1} \ \dots)$.

We use the following conventions:

- $e_1 \ e_2 \ e_3$ is a shorthand notation for $e_1 \ e_2$ and $e_2 \ e_3$
- If we write $e^* e'$, this means that there is an $n \geq 0$ and e_1, \dots, e_n so that $e \ e_1 \ \dots \ e_n \ e'$ is valid. In particular, $e^* e$. One says $*$ is the reflexive and transitive termination of.

Examples:

- $(+ \ 1 \ 1) \ 2$.
- $(+ \ (* \ 2 \ 3) \ (* \ 4 \ 5)) \ (+ \ 6 \ (* \ 4 \ 5)) \ (+ \ 6 \ 20) \ 26$.
- $(+ \ (* \ 2 \ 3) \ (* \ 4 \ 5)) \ (+ \ (* \ 2 \ 3) \ 20) \ (+ \ 6 \ 20) \ 26$.
- $(+ \ (* \ 2 \ (+ \ 1 \ 2)) \ (* \ 4 \ 5)) \ (+ \ (* \ 2 \ 3) \ (* \ 4 \ 5))$.
- $(+ \ (* \ 2 \ 3) \ (* \ 4 \ 5)) \ ^* 26$ but not $(+ \ (* \ 2 \ 3) \ (* \ 4 \ 5)) \ 26$.
- $(+ \ 1 \ 1) \ ^* (+ \ 1 \ 1)$ but not $(+ \ 1 \ 1) \ (+ \ 1 \ 1)$.
- `(overlay (circle 5 "solid" "red") (rectangle 20 20 "solid" "blue")) (overlay (rectangle 20 20 "solid" "blue") (overlay (circle 5 "solid" "red") (rectangle 20 20 "solid" "blue")))`



Experimentation
You in DrRacket with the "Stepper" button: Enter a complex expression in the definition area, press "Stepper" and then the "Step right" button and observe what happens.

In general, an expression can have several reducible sub-expressions, i.e. the congruence rules can be used in several places at the same time. In the examples above, for example, we have $(+ (* 2 3) (* 4 5))$ $(+ (* 2 3) 20)$ but also $(+ (* 2 3) (* 4 5))$ $(+ 6 (* 4 5))$. However, it is not difficult to see that whenever we have the situation $e_1 e_2$ and $e_1 e_3$, then there is an e_4 such that $e_2 * e_4$ and $e_3 * e_4$. This property is called *confluence*. Reductions that diverge can therefore always converge again.

The value you get at the end is clear in any case.

Based on these reduction rules, we can now define under which circumstances two programs are equivalent: Two expressions e_1 and e_2 are equivalent, written $e_1 e_2$, if there is an expression e such that $e_1 * e$ and $e_2 * e$.

Examples:

- $(+ 1 1) 2$.
- $(+ (* 2 3) 20) (+ 6 (* 4 5)) 26$.
- $(\text{overlay} (\text{rectangle } 20 \ 20 \ \text{"solid"} \ \text{"blue"}) (\text{overlay} (\text{circle } 5 \ \text{"solid"} \ \text{"red"}))$ .

The justification for this definition lies in the following important property: We can replace subexpressions within a large program with equivalent subexpressions without changing the meaning of the overall program. We can express this more formally as follows:

Let e_1 be a subexpression of a larger expression e_2 and $e_1 e_3$. Furthermore, let e_2' be a copy of e_2 in which occurrences of e_1 have been replaced by e_3 . Then: $e_2 e_2'$.

This property follows directly from the congruence rule and the definition of $*$. This concept of equivalence is identical to the one you know from school mathematics when you transform equations, for example when we transform $a + a - b$ to $2a - b$ because we know that $a + a = 2a$.

The use of $*$ to show that programs have the same meaning is also called *equational reasoning*.

2 Programmers design languages!

The programs you have written so far were essentially calculations that you could also perform on a calculator - with the big difference that BSL can handle the arithmetic of many types of values and not just numbers. For the functions that you can use in these expressions, you can choose from a fixed set of predefined functions. The *vocabulary* (set of functions) that you can use is therefore fixed.

A real programming language differs from a calculator in that you can define new functions based on existing functions and then use them in expressions and definitions of new functions. In general, you can define *names* and thus extend the vocabulary that is available to you (and others when your programs are published). So a program is more than a set of machine instructions - it also defines a *language* for the domain of the program. You will see that the ability to define new names is the key concept for dealing with the complexity of large software systems.

2.1 Function definitions

Here is the definition of a function that calculates the average of two numbers.

```
(define (average x y) (/ (+ x y) 2))
```

If this function definition is written to the definition area and then "Start" is pressed, nothing happens. A function definition is *not an expression*. The effect of this definition is that the name of the new function can now be used in expressions:

```
> (average 12 17)
14.5
> (average 100 200)
150
```

Of course, these values could also have been calculated without the previous function definition:

```
> (/ (+ 12 17) 2)
14.5
> (/ (+ 100 200) 2)
150
```

However, we can see that the second version is redundant: the algorithm for calculating the average has been replicated twice. It is also less abstract and less easy to understand, because we first have to understand that the algorithm calculates the average of two numbers, while in the first version we have given the algorithm a *name* that hides the details of the algorithm.

Good programmers generally try to avoid any kind of redundancy in programs. This is sometimes referred to as the DRY (Don't repeat yourself) principle. The reason for this is not only that it saves writing work, but also that redundant programs are more difficult to understand and maintain: If I want to change an algorithm later, I first have to find all copies of this algorithm in a redundant program and change each of them. Therefore, programming is never a monotonous repetitive activity, because recurring patterns can be encapsulated and reused in function definitions (and other forms of abstraction you will learn about).

In general, function definitions have this form:

```
(define (FunctionName InputName1 InputName2 ...) BodyExpression)
```

Function definitions are *not* expressions but a new category of programs. Function definitions may therefore not be used as arguments of functions, for example. A function definition starts with the keyword `define`. The sole purpose of this keyword is to distinguish function definitions from expressions. In particular, there must not be any functions called `define`. `FunctionName` is the name of the function. This is required in order to be able to use (or *call*) the function in expressions. `InputName1`, `InputName2` and so on are the *parameters* of the function. The parameters represent the input of the function, which only becomes known when the function is called. The `BodyExpression` is an expression that defines the output of the function. The parameters of the function are generally used within the `BodyExpression`. We call `BodyExpression` the *implementation* of the function or the *body* of the function.


Function calls have the form:

```
(FunctionName ArgumentExpression1 ArgumentExpression1 ...)
```



A function call to a (*user-defined*) function defined with `define` therefore looks exactly like the use of a built-in (*primitive*) function. This is no coincidence. The fact that you cannot see whether you are calling a primitive function or a user-defined function makes it easier to extend or change the programming language itself. For example, a primitive function can be turned into a user-defined function, or a programmer can define extensions that look as if the language has been extended by primitive functions.

2.2 Functions that produce images

Of course, BSL functions can receive not only numbers but any values as input or return them as output. In section §1.2 "Arithmetic with non-numeric values" you saw how to `place` an image in a scene using `place-image`. For example, the three expressions

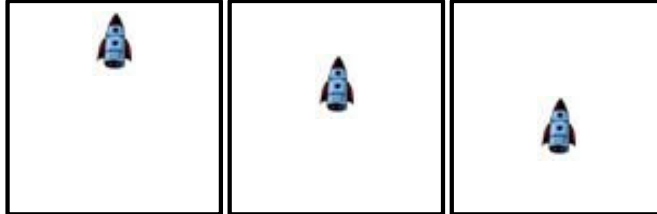
```
(place-image  50 20 (empty-scene 100 100))
```

```

(place-image  50 40 (empty-scene 100 100))
(place-image  50 60 (empty-scene 100 100))


```

the pictures



Obviously, these three expressions together are redundant, because they only differ in the parameter for the height of the rocket. With a function definition, we can express the pattern that these three expressions have in common:

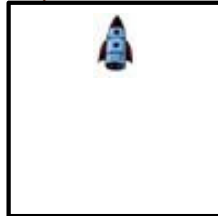
```

(define (create-rocket-scene height)
  (place-image  50 height (empty-scene 100 100)))

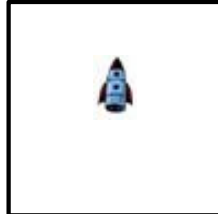
```

The three images can now be generated by calling the function.

```
> (create-rocket-scene 20)
```



```
> (create-rocket-scene 40)
```



```
> (create-rocket-scene 60)
```




You can also understand the height parameter as a time parameter; the function thus maps points in time to images. We can also understand such functions as a film or animation, because a film is characterized by the fact that there is an animation for every time there is a corresponding image.

In the Teachpack universe, there is a function that shows the movie corresponding to such a function on the screen. This function is called `animate`. The evaluation of the printout

```
(animate create-rocket-scene)
```

causes a new window to open in which an animation can be seen that shows how the rocket moves from top to bottom and finally disappears. When you close the window, a number is displayed in the interaction area; this number represents the current height of the rocket at the time the window was closed.

The `animate` function has the following effect: A stopwatch is set with the value 0 initially.

The counter value is incremented by one 28 times per second. Each time the counter is incremented by one, the `create-rocket-scene` function is evaluated and the resulting image is displayed in the window.

2.3 Meaning of function definitions

To define the meaning of function definitions, we need to say how function definitions and function calls are evaluated. With function definitions, expressions can no longer be evaluated in isolation (without considering the rest of the program); they are evaluated in the *context of* a set of function definitions. This context comprises the set of all function definitions that *precede* the expression to be evaluated in the program text. In order not to make our formal notation unnecessarily heavyweight, we will not explicitly add this context to the reduction relation; instead, we simply assume that there is a global context with a set of function definitions.

The evaluation rules from §1.5 "Meaning of BSL expressions" are now extended to include function definitions as follows:

- If the expression has the form $(f\ v_1 \dots v_n)$ and f is a primitive (built-in) function and the application of f to v_1, \dots, v_n yields the value v , then $(f\ v_1 \dots v_n) \rightarrow v$.

A teach pack is a library of functions that you can use in your program. You can activate a teach pack by You (require 2htdp/universe) to the beginning of your Add file.

How is the Expression $(\ast\ (-\ 1\ 1)\ (+\ 2\ 3))$ evaluated? How many steps are required? How is the Expression $(\text{and}\ (= 1\ 2)\ (= 3\ 3))$ evaluated? How many steps are required? Can you program a function `mul` that calculates the same as `*`, but (similar to and) requires fewer steps?

- If the expression has the form `(f v1 ... vn)` and `f` is *not* a primitive function and the context contains the function definition

```
(define (f x1 ... xn) BodyExpression)
```

then `(f v1 ... vn)` `NewBody`, where `NewBody` is created from `BodyExpression` by replacing all occurrences of `xi` with `vi` (for $i=1\dots n$).

The congruence rule from §1.5 "Meaning of BSL expressions" remains unchanged.

Example: Our program contains the following function definitions.

```
(define (g y z) (+ (f y) y (f z)))
(define (f x) (* x 2))
```

Then `(g (+ 2 3) 4)` `(g 5 4)` `(+ (f 5) 5 (f 4))` `(+ (* 5 2) 5`
`(f 4))` `(+ 10 5 (f 4))` `(+ 10 5 8)` `23`

Note that during the entire reduction the context contains both `f` and `g`, so it is not a problem that the function `f` is called in the body of `g`, although `f` is only defined *after* `g`. The evaluation of the program

```
(define (g y z) (+ (f y) y (f z)))
(g (+ 2 3) 4)
(define (f x) (* x 2))
```

fails, however, because the context in the evaluation of `(g (+ 2 3) 4)` is only `g` but does not contain `f`.

2.4 Conditional expressions

In the animation from the last section, the rocket simply disappears downwards out of the picture at some point. How can we get the rocket to "land" on the floor of the scene instead?

2.4.1 Motivation

Obviously, we need a case distinction in our program for this. You are familiar with case distinctions from numerous real-life examples. For example, the grading scheme for an exam, which assigns a grade to each number of points, is a function that distinguishes the different limits for the resulting grades. In BSL, we can define a grading scheme where you need at least 90 points for a 1 and every 10 points below that you go down a grade as follows:

```
(define (note points)
  (cond
    [(>= points 90) 1]
    [(>= points 80) 2]
    [(>= points 70) 3]
```

```

[(>= points 60) 4]
[(>= points 50) 5]
[(< points 50) 6]])

```

Some examples of how to use the notation scheme:

```

> (note 95)
1
> (note 73)
3
> (note 24)
6

```

2.4.2 Meaning of conditional expressions

In the general case, a conditional expression looks like this:

```

(cond
 [ConditionExpression1 ResultExpression1]
 [ConditionExpression2 ResultExpression2]
 ....
 [ConditionExpressionN ResultExpressionN])

```

A conditional expression therefore starts with an opening bracket and the keyword `cond`. This is followed by any number of lines, each of which contains two expressions. The left-hand expression is called the *condition* and the right-hand expression is called the *result*.

A `cond` expression is evaluated as follows. DrRacket first *evaluates* the first condition `ConditionExpression1`. If this evaluation results in the value `#true`, the value of the entire `cond` expression is the value of `ResultExpression1`. If, on the other hand, this evaluation results in the value `#false`, the process continues with the second line and proceeds in exactly the same way as with the first line. If there is no next line - i.e. all conditions have been evaluated to `#false` - the process is terminated with an error message. It is also an error if the evaluation of a condition does not result in `#true` or `#false`:

```

> (cond [(< 5 3) 77]
        [(> 2 9) 88])
cond: all question results were false
> (cond [(+ 2 3) 4])
cond: question result is not true or false: 5

```

We need to add the following two rules to the reduction rules for BSL to take conditional expressions into account:

```

(cond [#false e] [e2 e3] .... [en-1 en]) (cond [e2 e3] .... [en-1 en])
and

```

```
(cond [#true e] [e2 e3] . . . . [en-1 en]) e
```

In addition, we supplement the evaluation items that can be used in the congruence rule as follows: In an expression of the form

```
(cond [e0 e1]
      [e2 e3]
      . . . .
      [en-1 en])
```

is the expression e_0 in an evaluation position, but not e_1, \dots, e_n .



Example: Let's look at the call (note 83) in the example above. Then

```
(note 83) (cond [(>= 83 90) 1] [(>= 83 80) 2] ...) (cond
[#false 1] [(>= 83 80) 2] ...) (cond [(>= 83 80) 2] ...) (cond
[#true 2] ...) 2
```

2.4.3 Example

Back to our rocket. Obviously, we need to distinguish between two cases here. While the rocket is still above the bottom of the scene, it should sink as usual. However, if the rocket has already reached the ground, it should not sink any further.

Since the scene is 100 pixels high, we can distinguish the cases that the current height is less than or equal to 100 and that the current height is greater than 100.

```
(define (create-rocket-scene-v2 height)
  (cond
    [(<= height 100)
     (place-image  50 height (empty-scene 100 100))]
    [(> height 100)
     (place-image  50 100 (empty-scene 100 100))]))
```

For the variants the `create-rocket-scene` function, we use the naming convention of giving the variants the suffixes -v2, -v3, etc.

2.4.4 Some syntactic sugar...

Two special cases of conditional expressions are so common that BSL has its own syntax that is optimized for these special cases.

The first special case is when you want to have a branch of the condition that is always used when all other branches are not applicable. In this case, you can use the keyword `else` instead of the condition. We could therefore also formulate the example above like this:

```
(define (note points)
  (cond
    [(>= points 90) 1]
    [(>= points 80) 2]
```

```

[(>= points 70) 3]
[(>= points 60) 4]
[(>= points 50) 5]
[else 6]])

```

However, the `else` clause may only be used in the last branch of a `cond` expression:

```

> (cond [(> 3 2) 5]
        [else 7]
        [< 2 1) 13])
cond: found an else clause that isn't the last clause in its cond expression

```

The `else` two is equivalent to a branch with the always fulfilled condition `#true`, therefore in the general case the meaning of

```

(cond [e0 e1]
      [e2 e3]
      . . .
      [else en])

```

defined as the meaning of

```

(cond [e0 e1]
      [e2 e3]
      . . .
      [#true en])

```

So in this case, we are not specifying reduction rules for this language construct, but instead a transformation that transforms the meaning. If language constructs add "nothing new" but are merely an abbreviation for a particular use of existing language constructs, such language constructs are also called *syntactic sugar*.

Another special case of conditional expressions is that there is only one condition that is to be checked, and depending on whether this condition is true or false, a different expression is to be selected. For this case, there is the `if` construct.

Example:

```

(define (aggregate state temperature)
  (if (< temperature 0) "frozen" "liquid"))

```

```

> (aggregate state -5)
"frozen"

```

In general, an `if` expression has the following form:

```

(if CondExpression ThenExpression ElseExpression)

```

An `if` expression is syntactic sugar; the meaning is determined by the transformation into this expression:

```
(cond [CondExpression ThenExpression]
      [else ElseExpression])
```

In general, `if` is suitable for situations in which we want to say something like "either one or the other". The `cond` expressions are suitable if you want to distinguish between more than two situations.

Although it initially looks like `if` is a special case of `cond`, you can also replace any `cond` expression with a nested `if` expression. For example, the function from above can also be written like this:

```
(define (note points)
  (if (>= points 90)
      1
      (if (>= points 80) 2
          (if (>= points 70) 3
              (if (>= points 60) 4
                  (if (>= points 50) 5
                      6))))))
```

In such cases, the `cond` construct is obviously more suitable because no deeply nested expressions are required. Nevertheless, it can be said that `cond` and `if` are equally powerful because one can be transformed into the other in such a way that the meaning remains the same.

2.4.5 Evaluation of conditional expressions

In §2.4.2 "Meaning of conditional expressions" we defined that in a conditional expression

```
(cond [e0 e1]
      [e2 e3]
      . . . .
      [en-1 en])
```

only the expression e_0 is in an evaluation position, but not e_1, \dots, e_n . Why this restriction - why not also allow the evaluation of e_1, \dots, e_n ? Consider the following example:

```
(cond [(= 5 7) (/ 1 0)]
      [(= 3 3) 42]
      [(/ 1 0) 17])
```

According to our evaluation rules:

```
(cond [(= 5 7) (/ 1 0)]
      [(= 3 3) 42]
      [(/ 1 0) 17])
```

```
(cond [#false (/ 1 0)]
      [(= 3 3) 42]
      [(/ 1 0) 17])
```

```
(cond [(= 3 3) 42]
      [(/ 1 0) 17])
```

```
(cond [#true 42]
      [(/ 1 0) 17])
```

42

If it were allowed to evaluate on the other positions as well, according to our rules we would have to abort the calculation in the example with an error as soon as we evaluate one of the `(/ 1 0)` expressions. According to the terminology from §1.5 "Meaning of BSL expressions", we lose the confluence property and the value of an expression is no longer unique.

The example above is very artificial, but we will see later that conditional expressions are often used to ensure that a function terminates - i.e. the evaluation does not continue indefinitely. It is therefore essential that not all positions can be evaluated. Operators such as `cond`, where the evaluation of any arguments is not permitted, are also called *non-strict*. Normal functions, on the other hand, where all arguments are evaluated before the function is used, are called *strict*.

2.4.6 Brevity is the spice of life

From the meaning of conditional expressions, we can derive a number of refactorings that can (and should) be used to simplify conditional expressions. Unnecessarily long expressions are also a violation of the DRY principle, because they are also a form of redundancy. Regarding the refactorings that turn a conditional expression into a Boolean expression, we will see later (§8.9.4 "Meaning of Boolean expressions") that these refactorings also match in terms of strictness behavior.

Printout	Simplified expression	Condition
<code>(if e #true #false)</code>	<code>e</code>	<code>e #true or e #fa</code>
<code>(if e #false #true)</code>	<code>(not e)-</code>	
<code>(if e e #false)</code>	<code>e</code>	<code>e #true or e #fa</code>
<code>(if (not e-1) e-2 e-3)</code>	<code>(if e-1 e-3 e-2)-</code>	
<code>(if e-1 #true e-2)</code>	<code>(or e-1 e-2)</code>	<code>e-2 #true or e-2</code>
<code>(if e-1 e-2 #false)</code>	<code>(and e-1 e-2)</code>	<code>e-2 #true or e-2</code>
<code>(if e-1 (f ... e-2 ...) (f ... e-3 ...))</code>	<code>(f ... (if e-1 e-2 e-3) ...) -</code>	






We will discuss an example of the last refactoring in §2.6.2 "DRY Redux". Later, we will show how to prove the correctness of these refactorings. You can understand the equivalences informally by simply playing through the possible cases and replacing expressions with `#true` or `#false`.

2.5 Definition of constants

If we look at `(animate create-rocket-scene-v2)`, we realize that the animation is still not satisfactory, because the rocket half sinks into the ground. The reason for this is that `place-image` places the center of the image at the specified point. For the rocket to land cleanly on the ground, the center must therefore be above the ground. With a little thought, it quickly becomes clear that the rocket can only reach a height of

`(- 100 (/ (image-height ) 2))`

should drop. This means that we have to modify our `create-rocket-scene-v2` function as follows:

```
(define (create-rocket-scene-v3 height)
  (cond
    [(<= height (- 100 (/ (image-height  ) 2))]
    (place-image  50 height (empty-scene 100 100))]
    [(> height (- 100 (/ (image-height  ) 2))]
    (place-image  50 (- 100 (/ (image-height  ) 2))
      (empty-scene 100 100)))))
```

2.6 DRY: Don't Repeat Yourself!

A call to `(animate create-rocket-scene-v3)` illustrates that the rocket now lands as we want it to, but it is obvious that `create-`

`rocket-scene-v3` violates the principle mentioned in section §2.1 "Function definitions" that good programs do not contain redundancy. In programmer jargon, this principle is often referred to as the DRY principle - Don't Repeat Yourself. - called.

2.6.1 DRY through constant definitions






One type of redundancy that occurs in `create-rocket-scene-v3` is that the height and width of the scene is repeated very often. Imagine you want to have a 200 by 400 scene instead of a 100 by 100 scene. For this purpose, you have to find all occurrences of the old height and width, find out whether they stand for the width or the height or something else (which is why the problem cannot be solved with automatic text replacement), and replace them with the new value. The effort is still manageable with `create-rocket-scene-v3`, but if you look at programs with many thousands of lines of code, it quickly becomes clear that this is a big problem.

Ideally, the relationship between the requirements for a program and the program text should be *continuous*: A small change request to the requirements for a program should only require a small change to the program text. In our concrete example, we can solve this problem with `define`. With `define`, not only functions, but also *constants* can be defined.

can be defined. For example, we can include this definition in our program write in:

```
(define HEIGHT 100)
```

The meaning of such a definition is that in the rest of the program `HEIGHT` is a valid expression that has the value `100` when evaluated. If we replace all occurrences of `100` in the program that stand for the height with `HEIGHT` and do the same for `WIDTH`, we get this variant of `create-rocket-scene`:

```
(define WIDTH 100)
(define HEIGHT 100)
(define (create-rocket-scene-v4 height)
  (cond
    [(<= height (- HEIGHT (/ (image-height  ) 2)))
     (place-image  50 height (empty-scene WIDTH
HEIGHT))] [(> height (- HEIGHT (/ (image-height  )
2)))
     (place-image  50 (- HEIGHT (/
(image-height  ) 2))
```

Constant values such as `100` in program texts are often derogatorily referred to as *magic numbers* by programmers.

For the meaning of the program it does not matter that the name of the constant consists only of capital letters. This is merely a naming convention that allows programmers to easily recognize which names refer to constants.

```
(empty-scene WIDTH HEIGHT)))))
```

Test with `(animate create-rocket-scene-v4)` that the program still works. Experiment with other values for `WIDTH` and `HEIGHT` to see that this small change is really enough to change the size of the scene.

In programming jargon, program changes that alter the structure of the program without changing its behavior are called *refactorings*. Refactorings are often carried out to improve the maintainability, readability or extensibility of the program. In our case, we have improved both maintainability and readability through this refactoring. We have already illustrated the improved maintainability; the improved readability is due to the fact that we can read the meaning of the constants from names such as `WIDTH`, whereas with magic numbers such as `100` we first have to find out this meaning through precise analysis of the program (also known as *reverse engineering* in programming jargon).

Incidentally, it does not matter whether the definitions of the constants are above or below the `create-rocket-scene` definition. The constants are visible within the entire program file. The constants are said to have *global scope*. To avoid having to search for the definitions of the constants in the program text, it makes sense to always define them in the same place. In many programming languages there is a convention that constant definitions are always at the beginning of the program text, so we will also adhere to this convention.

However, `create-rocket-scene-v4` still violates the DRY-principle. For example, the expression

```
(- HEIGHT (/ (image-height  ) 2))
```

several times. This redundancy can also be eliminated with `define`, because the value assigned to a constant can be described by any complex expression. In general, constant definitions have the following form:

```
(define CONSTANTNAME CONSTATEXpression)
```

In the previous example, we can name the constant `ROCKET-CENTER- TO-BOTTOM`, for example. Notice how choosing good names makes the meaning of the program much more obvious. Without this name, every time we want to read and understand the complex expression above, we would have to find out again that the desired distance from the center of the rocket to the ground is calculated here.


We can do the same with the multiple expression `(empty-scene WIDTH HEIGHT)`. We give it the name `MTSCN`.

The number `50` in the program text is also a *magic number*, but it has a different quality than `WIDTH` and `HEIGHT`: it is dependent on the value of other constants, in this case `WIDTH`. As this constant stands for the horizontal center, we define it as `(define MIDDLE (/ WIDTH 2))`.

In the tradition of the family of programming languages from which BSL originates, it is common to use the English pronunciation of the letters of the alphabet to abbreviate names. `MTSCN` is spoken hence "empty scene".

The last type of redundancy that still occurs is that the rocket itself appears several times in the program text. Although the rocket is not a number literal and therefore not a *magic number*, it is a *magic image* - with exactly the same disadvantages as *magic numbers*. We therefore also define a constant `ROCKET` for the image. The program, which contains all these *refactorings*, now looks like this:

```
(define WIDTH 100)
(define HEIGHT 100)
(define MIDDLE (/ WIDTH
2))
(define MTSCN (empty-scene WIDTH HEIGHT))


(define ROCKET )
(define ROCKET-CENTER-TO-BOTTOM (- HEIGHT (/ (image-
height ROCKET) 2)))
(define (create-rocket-scene-v5 height)
  (cond
    [(<= height ROCKET-CENTER-TO-BOTTOM)
     (place-image ROCKET MIDDLE height MTSCN)]
    [(> height ROCKET-CENTER-TO-BOTTOM)
     (place-image ROCKET MIDDLE ROCKET-CENTER-TO-
BOTTOM MTSCN)]))
```

2.6.2 DRY Redux

Stop! Even `create-rocket-scene-v5` still violates the DRY principle. However, we will not eliminate the remaining redundancies by defining functions or constants.

One redundancy is that the condition `(> height ROCKET-CENTER-TO-BOTTOM)` is true if `(<= height ROCKET-CENTER-TO-BOTTOM)` is false. However, this information is not directly in the program text; instead, the condition is repeated and negated. One possibility would be to write a function that calculates this condition depending on the `height` parameter and then call this function in both branches of the condition (and negate it once). In this case, however, there is a simpler solution, namely to use `if` instead of `cond`. This allows us to eliminate this redundancy:

```
(define WIDTH 100)
(define HEIGHT 100)
(define MIDDLE (/ WIDTH
2))
(define MTSCN (empty-scene WIDTH HEIGHT))

(define ROCKET )
(define ROCKET-CENTER-TO-BOTTOM (- HEIGHT (/ (image-
height ROCKET) 2)))
(define (create-rocket-scene-v6 height)
  (if
```

```


    (<= height ROCKET-CENTER-TO-BOTTOM)
    (place-image ROCKET MIDDLE height MTSCN)
    (place-image ROCKET MIDDLE ROCKET-CENTER-
    TO-
    BOTTOM MTSCN)))

```

The last redundancy we want to eliminate in `create-rocket-scene-v6` is that the two calls to `place-image` are identical except for one parameter. If in a conditional expression the bodies of all branches are identical except for one sub-expression, we can *pull* the condition into the expression like this:

```

(define WIDTH 100)
(define HEIGHT 100)
(define MIDDLE (/ WIDTH
2))
(define MTSCN (empty-scene WIDTH HEIGHT))

(define ROCKET )
(define ROCKET-CENTER-TO-BOTTOM (- HEIGHT (/ (image-
height ROCKET) 2)))
(define (create-rocket-scene-v7 height)
  (place-image
   ROCKET
   MIDDLE
   (if (<= height ROCKET-CENTER-TO-BOTTOM)
       height
       ROCKET-CENTER-TO-BOTTOM) MTSCN))

```

2.7 Meaning of function and constant definitions

We said above that it does not matter whether the constants are defined above or below the function definition. However, the order in which these constants are defined does matter. As you can see, some of the constant definitions use other constants. For example, the definition of `MTSCN` uses `WIDTH`. This also makes sense, because otherwise you would still have the redundancy that you actually wanted to eliminate.

DrRacket evaluates a program from top to bottom. If it encounters a constant definition, the value of the expression to which the name is to be bound (the `CONSTANT expression`) is calculated immediately. If this expression contains a constant that DrRacket does not yet know, an error occurs:

```

> (define A (+ B 1))
B: this variable is not defined
> (define B 42)

```

Therefore, only those constants (and functions) that have already been defined above the definition may be used in constant definitions.

Does this problem also occur with functions? Here is an experiment:

```
(define (add6 x) (add3 (add3 x)))
(define (add3 x) (+ x 3))
```

```
> (add6 5)
11
```


The reason why the order of function definitions is not important is that DrRacket only registers that there is a new function with the specified name when evaluating a function definition, but in contrast to constant definitions, it does not evaluate the `BodyExpression` of the function.

More formally, we can define the meaning of programs with function and constant definitions as follows:

- A program is a sequence of expressions, constant definitions and function definitions. These can occur in any order.
- A context is a set of function and constant definitions. The context is empty at the start of program execution.
- A program is evaluated from left to right (or top to bottom). There are three different cases here.
 - If the next program element is an expression, it is evaluated to a value according to the known reduction rules in the current context. For the evaluation of constants, `x v` applies if the context contains the definition `(define x v)`.
 - If the next program element is a function definition, this function definition is added to the current context.
 - If the next program element is a constant definition `(define CONSTANTNAME CONSTATEXpression)`, `CONSTATEXpression` is evaluated to a value `v` in the current context and the definition `(define CONSTANTNAME v)` is added to the context.

The current context is displayed in the DrRacket stepper as the set of function and constant definitions above the expression currently being reduced. Please use the stepper to visualize the reduction of the following program. It is best to first try to predict on a piece of paper what the reduction steps will be and then check with the stepper.

```
(define WIDTH 100)
(define HEIGHT 100)
(define MIDDLE (/ WIDTH
2))
(define MTSCN (empty-scene WIDTH HEIGHT))

(define ROCKET )
```

```

(define ROCKET-CENTER-TO-BOTTOM (- HEIGHT (/ (image-
height ROCKET) 2)))
(define (create-rocket-scene-v7 height)
  (place-image
   ROCKET
   MIDDLE
   (if (<= height ROCKET-CENTER-TO-BOTTOM)
       height
       ROCKET-CENTER-TO-BOTTOM) MTSCN))
(create-rocket-scene-v7 42)

```

Side note: Count the number of additional reduction steps you need per call of `create-rocket-scene-v7` (i.e. if you add more calls to the program). How many additional steps do you need if you use `create-rocket-scene-v2` instead? How do the differences come about and what do they mean?

2.8 Programming is more than just understanding rules!

A good chess player must understand the rules of chess. But not everyone who understands the rules of chess is also a good chess player. The rules of chess tell you nothing about how to play a good game of chess. Understanding the rules is only the first small step on the way to achieving this.

Every programmer must master the "mechanics" of the programming language: What kind of constructs are there in the programming language and what do they mean? What predefined functions and libraries are there?

Nevertheless, this does not make you a good programmer. Many beginner's books (and unfortunately also many beginner's courses at universities) for programming are designed in such a way that they *only* focus on these mechanical aspects of programming. Even worse, they don't even teach you what exactly their programs mean, they essentially just teach you the syntax of a programming language and some of its libraries.

This is because some programming languages used in beginner courses are so complicated that you spend most of the semester just learning the syntax of the language. Our language, BSL, is so simple that you already understand the mechanics of the language, which is one reason why we chose it. We will introduce a few more language features, but most of this lecture is about the more interesting part of programming: How do you get from a problem description to a good program? What types of abstraction are there and how can they be used? How do I deal with errors? How do I structure my program so that it is readable, maintainable and reusable? How can I master the complexity of very large programs?

The answers we give to these questions will be useful to you in *all* the programming languages you will use. That's what this course will be about.

3 Systematic program design

No new language features are presented in this chapter. Instead, this chapter deals with the *methodology* of programming: How can a problem description be systematically transformed into a high-quality program?

3.1 Functional decomposition

Programs rarely consist of a single function. Typically, programs consist of many function definitions, some of which are interdependent in that they call each other. Consider the following program for creating somewhat clumsy "Nigerian-Scam" letters:

```
(define (letter fst lst signature-
  name) (string-append
    (opening lst)
    "\n"
    (body fst lst)
    "\n"
    (closing signature-name)))

(define (opening lst)
  (string-append "Dear Mr./Mrs. " lst ","))

(define (body fst
  lst) (string-append
  "After the last annual calculations of your GNB "
  "account activity we have determined that you, "
  fst lst
  ", are eligible to receive a tax refund of $479.30.\n"
  "Please submit the tax refund request (http://www...)
  " "and allow us 2-6 days in order to process it.))

(define (closing signature-
  name) (string-append
  "With best
  regards,\n"
  signature-name))
```

We can now use these definitions to create many such letters with little effort:

```
> (letter "Tillmann" "Rendel" "Klaus Ostermann")
"Dear Mr./Mrs. Rendel,\nAfter the last annual calculations
of your GNB account activity we have determined that you,
TillmannRendel, are eligible to receive a tax refund of
$479.30.\nPlease submit the tax refund request (http://www...)"
```

This part of the script is based on [HTDP/2e] Chapter I and the article "On Teaching How to Design Programs" by Norman Ramsey.

and allow us 2-6 days in order to process it.\nWith best regards,\nKlaus Ostermann"

The result is a long string. The `\n` in the string stands for a line break. As soon as this string is written to a file, for example, the `\n` becomes a real line break.

In general, a program should be designed so that there is one function per task that the program should perform. Tasks should be arranged hierarchically: At the top level there is the overall task of the program (such as creating a form letter); this large task is broken down into smaller tasks such as creating the opening of a form letter, these can be broken down again into further small tasks if necessary.

The structure of the functions should follow the hierarchical structure of the tasks. For each task there is a function that calls the functions that correspond to the subtasks in its implementation. Accordingly, the functions can be arranged as a tree (or acyclic graph) with the main function at the root and the functions of the lowest level (which only call primitive functions) at the leaves.

A program in which each function has a clear task is easy to understand and easier to maintain, because if there are change requests, these typically relate to a specific task of the system. If the implementation of this task is localized in the form of a function definition, only this one function needs to be modified.

If you have a large number of functions to program, the question arises as to the order in which you program these functions. Two important variants are "top-down" and "bottom-up". "Top-down" means that you start with the main function, then program all the functions that are called in the main function, then all the functions that are in turn called by these functions, and so on. "Bottom-up" is exactly the opposite: you first program the simplest functions, which in turn only call primitive functions, then the functions that call the functions you have just programmed. This is continued until the main function is programmed at the very end. Estimating which approach is the right one in which situation is an important topic in the methodology of software construction.

If the call structure of the functions is an acyclic graph, this automatically results in a layer structure in which all functions in a layer only call functions from layers below it. An even stronger restriction is that functions may only call functions from the layer directly below them. However, this restriction makes sense because, with a clever choice of functions, it makes it possible to understand a layer without having to understand all the layers below it. This situation is called *hierarchical abstraction*, and it is the key to dealing with the complexity of large software systems.

The basic idea behind hierarchical abstraction through functions is that functions should be designed in such a way that a programmer is able to effectively define a function based on the name and documentation (the *specification*) of a function.

tively in a program - i.e. it is not necessary to understand the program text of the function and (transitively) of all functions that this function calls. We will see later that the specification of a function can be at least partially formalized in the form of test cases. For informal specifications, comments are usually used above the function definition.

In our example above, it is not necessary to understand the details of the `opening` function in order to understand the `letter` function; it is enough to know that this function returns the opening of the letter - whatever the details may be. will look like. In this case, the name of the function is sufficient to be able to use `opening` effectively; in other cases, further documentation is required. In any case, however, you should try to choose function names in such a way that as little further documentation as possible is required, because when a program evolves, it often happens that only the program text is "maintained" but not the comments and documentation is therefore often obsolete.

3.2 From problem to program

Consider the following problem for a program to be created:

The owner of a movie theater has complete freedom in setting ticket prices. The more he charges, the fewer the people who can afford tickets. In a recent experiment the owner determined a precise relationship between the price of a ticket and average attendance. At a price of \$5.00 per ticket, 120 people attend a performance. Decreasing the price by a dime (\$.10) increases attendance by 15. Unfortunately, the increased attendance also comes at an increased cost. Each performance costs the owner \$180. Each attendee costs another four cents (\$0.04). The owner would like to know the exact relationship between profit and ticket price so that he can determine the price at which he can make the highest profit.

The task is relatively clear, but how to turn it into a program is not. A good way to approach this task is to look at the quantities and their dependencies on each other and define them one after the other in the form of a function:

The problem states how the number of spectators depends on the ticket price. This is a clearly defined subtask, so it deserves its own function:

```
(define (attendees ticket-price)
  (+ 120 (* (/ 15 0.1) (- 5.0 ticket-price))))
```

Turnover depends on the sale of tickets: It is the product of the admission price and the number of spectators.

```
(define (revenue ticket-price)
  (* (attendees ticket-price) ticket-price))
```

The costs are made up of two parts: A fixed part (\$180) and a variable part that depends on the number of spectators. Since the number of spectators in turn depends on the ticket price, this function must also include the

ticket price as an input parameter and uses the already defined `attendees` function:

```
(define (cost ticket-price)
  (+ 180 (* 0.04 (attendees ticket-price))))
```

The profit is ultimately the difference between turnover and costs. As we already have functions for calculating turnover and costs, this function must receive all the values that these functions require as input - in this case, this is the admission price:

```
(define (profit ticket-
  price) (- (revenue ticket-
  price)
  (cost ticket-price)))
```

We can now use these functions to calculate the profit at a certain entry price. Try out at which price the profit is maximized!

Here is an alternative version of the same program, which consists of only one function:

```
(define (profit
  price) (- (* (+ 120
    (* (/ 15 0.1)
      (- 5.0 price)))
    price)
  (+ 180
    (* 0.04
      (+ 120
        (* (/ 15 0.1)
          (- 5.0 price)))))))
```

Check that this definition actually produces the same results as the program above. So, in principle, we only need one function for this program; however, it is obvious that the first version above is much more readable. It is also more maintainable: for example, consider what changes are needed in the program text if the fixed cost is dropped and a cost of \$1.50 per viewer is incurred instead. Try implementing this change in both versions. Compare.

3.3 Systematic design with design recipes

Most of what they have learned about programming so far is about how the programming language they are using works and what its constructs mean. We have learned about some important language constructs (function definitions, constant definitions, expressions) and gained some experience with how to use these constructs.

This program contains various *magic numbers*. Eliminate these by using appropriate constant definitions!

However, this knowledge is not sufficient to systematically construct a program from a problem description. To do this, we need to learn what is relevant in a problem description and what is not. We need to understand which data the program consumes and which it must produce depending on the input data. We have to find out whether a required function already exists in a library or whether we have to program this function ourselves. Once we have a program, we have to make sure that it actually behaves as desired. All kinds of errors can occur here, which we need to understand and rectify.

Good programs have a short description of what they do, what input they expect and what output they produce. It is best to document at the same time that the program actually works. Programs should also be structured in such a way that a small change in the problem description only means a small change in the program.

All this work is necessary because programmers rarely write programs for themselves. Programmers write programs that other programmers have to understand and develop further. Large programs are developed by large teams over long periods of time. New programmers join during this period, others leave. Customers are constantly changing their requirements for the program. Large programs almost always contain bugs, and often those who have to fix the bug are not the same as those who built the bug.

Moreover, a real program in use is never "finished". In most cases, it must be constantly developed further or at least maintained or adapted to new technology. It is not uncommon for programs to have a lifespan of many decades and therefore the original programmers of a program that is to be further developed or maintained may already be retired.

or are deceased.

Incidentally, the same problems occur even if there is only one programmer, because even this programmer forgets after some time what he once thought of a program and then benefits from a meaningful design just as much as if another programmer were to read his code.

Do your research,
what the "year
2000 problem" is.

For these reasons, we will provide you with systematic instructions that can be used to solve various design tasks step by step. We call these instructions *design recipes*.

3.3.1 Testing

An important part of the methodology presented will be to *test* programs systematically and automatically. When testing, you execute a program or a program part (such as a function) with test data and check whether the result is what you expect. Of course, you can test "by hand", for example by evaluating expressions in the interaction area and checking the results. However, it quickly becomes boring if you write down the same tests again and again to check that you have not "broken" anything when you change the program.

When, where, why and how much you should test will be discussed later. Here we only describe *how* to test automatically in DrRacket. There is a special function in BSL for this, `check-expect`. This expects two parameters, the first of which is an expression to be tested and the second an expression describing the desired result. Example: `(check-expect (+ 2 3) 5)` checks whether the result of the evaluation of `(+ 2 3)` is 5.

Here is an example of how `check-expect` can be used to create a function to test the conversion between Fahrenheit and degrees Celsius:

```
(check-expect (f2c -40) -40)
(check-expect (f2c 32) 0)
(check-expect (f2c 212) 100)

(define (f2c f)
  (* 5/9 (- f 32)))
```

All `check-expect` tests are executed each time the "Start" button is pressed. If all tests are successful, this is confirmed by a short message. If at least one test fails, this is indicated by an error message with more detailed information.

There are several variants of `check-expect`, such as `check-within` and `check-range`. Get an overview with the help of the documentation.

To support the programmer, DrRacket shows which parts of your program were run during the execution of the tests by coloring the code. Try out this behavior for yourself!

Of course, tests do not only work with numbers but with all types of data, including images. For example, we can use this function

```
(define (ring innerradius outerradius color)
  (overlay (circle innerradius "solid" "white")
    (circle outerradius "solid" color)))
```

test like this:

```
(check-expect (ring 5 10 "red") )
```

Since `check-expect` expects any expressions as parameters, *properties* of results can also be checked instead of the result itself, for example like this:

```
(check-expect (image-width (ring 5 10 "red")) 20)
```

We will see later that this is important in order not to link tests too strongly to the implementation of a function.

3.3.2 Information and data

A program describes a calculation that processes and produces *information* from the domain of the program. For example, information is something like "the car is 5m long" or "the employee's name is 'Müller'".

However, a program cannot process such information directly. Instead, we have to represent information as *data*. This data can in turn be *interpreted* as information. For example, we could represent the first piece of information above as a number with the value 5 and the second piece of information as a string with the value "Miller".

A date such as the number 5 can in turn be interpreted in many different ways the. For example, as in the example above, it can mean the length of a car in meters. However, it can also be interpreted as the temperature in degrees Celsius, as an amount of money in euros, or as the final grade of your exam in this course (hopefully not :-).

Because this relationship between information and data is so important, from now on we will write it down in the form of special comments, which we call *data definitions*. A data definition describes a class of data by a meaningful name that indicates the interpretation of the data.

Here are some examples of data definitions:

```
; Distance is a Number.  
; interp. the number of pixels from the top margin of a canvas  
  
; Speed is a Number.  
; interp. the number of pixels moved per clock tick  
  
; Temperature is a Number.  
; interp. degrees Celsius  
  
Length is a Number.  
; interp. the length in centimeters  
  
Count is a Number.  
; interp. the number of characters in a string.  
...
```

At this stage, you only know a few forms of data (numbers, strings, images, truth values), so you need to represent all information with these data types. Later we will get to know other data types where it will be much more challenging to choose a suitable representation for your information.

It is often helpful to provide data examples for a data definition.

```
; Temperature is a Number.  
; interp. degrees Celsius  
Examples:
```

```
(define sunny-weather 25)
(define bloody-cold -5)
```

The definition of data examples has two advantages: 1) They help to understand a data definition. For example, it could be that you have inadvertently designed your data definition in such a way that there are no data examples at all. 2) You can use the examples in tests of functions that consume or produce such data.

3.3.3 Design recipe for function definition

Based on the separation between information and data just discussed, we can now describe the design of individual functions as a sequence of steps.

1. Define how you represent the information relevant to the function (input and output) as data. Formulate corresponding data definitions (if not already available). For non-trivial data definitions, give some interesting examples for the data definition.
2. Write a signature, a task description, and a function header. A *signature* is a BSL comment that tells the reader how many and which inputs the function consumes and what output it produces. Here are two examples:

- For a function that consumes a string and produces a number:
; String - Number
- For a function that consumes a temperature and a Boolean value and produces a string:
Temperature Boolean - String

Note that we have used the previous data definition for *Temperature*.

A task *description* is a BSL comment that summarizes the purpose of the function **in one line**. The task description is the shortest possible answer to the question: *What does the function calculate?* Every reader of your program should understand what a function calculates without having to read the function definition itself.

A *function header*, sometimes also called a *header* or *stub*, is a function definition that matches the signature but in which the body of the function is only a dummy value, for example `0` if a number is to be returned or `(empty-scene 100 100)` if an image is to be returned. When designing the function header, important decisions must still be made, namely the names of the function and the input parameters must be determined. Typically, the parameter names should contain a

The names of the parameters can often be used to give an indication of what information or purpose the parameters represent. The names of the parameters can often be used sensibly in the task description.

Here is a complete example of a function definition after this step:

```
; Number String Image -> Image
; add s to img, y pixels from top, 10 pixels to the left
(define (add-image y s img)
  (empty-scene 100 100))
```

At this point, you can already press the "Start" button and use the function - but of course only the dummy value is returned and not the desired result.

3. Write *tests* between the task description and the function header that document what the function does using examples. On the one hand, these tests are part of the documentation of the function, on the other hand, these tests are executed automatically and thus protect it from errors in the function body. Here is an example of what a function looks like after this step:

```
Number -> Number
; compute the area of a square whose side is
len (check-expect (area-of-square 2) 4)
(check-expect (area-of-square 7) 49)

(define (area-of-square len) 0)
```

Now run the tests once and check that all tests (for which the dummy implementation does not happen to return the correct result) fail. This step is important to find errors in the formulation of the tests that could cause the test to always be successful.

4. In this step, you consider which of the available input data and any auxiliary functions and variables you need for the calculation. You replace the dummy value from the second step with *a* template in which the input data/functions/variables from above appear. At the moment, this template looks like this: the input data/functions/variables are simply separated by ... separated from each other unsorted in the function body. We will get to know more interesting templates later.

In our last example, the function after this step could look like this:

```
Number -> Number
; compute the area of a square whose side is
len (check-expect (area-of-square 2) 4)
(check-expect (area-of-square 7) 49)

(define (area-of-square len) (... len ...))
```

5. Now it is time to implement the function body, i.e. to gradually replace the template with an expression that fulfills the specification (signature, task description, tests). Our `area-of-square` function could now look like this:

```
Number -> Number
; compute the area of a square whose side is
len (check-expect (area-of-square 2) 4)
(check-expect (area-of-square 7) 49)

(define (area-of-square len) (* len len))
```

The `add-image` function could look like this after this step:


```
; Number String Image -> Image
; add s to img, y pixels from top, 10 pixels to the left
(check-expect (add-image 5 "hello" (empty-scene 100
100))
              (place-image (text "hello" 10 "red") 10 5 (empty-
scene 100 100)))
(define (add-image y s img)
  (place-image (text s 10 "red") 10 y img))
```

It is important to understand that the test *does not* state that `add-image` was implemented using `place-image`. The `check-expect` function compares the images resulting from the expressions and not the expressions themselves. For example, we could also replace the second parameter with an image literal (i.e. an image in the program text). Therefore, the test does not "reveal" the implementation of the function. This is important because it allows us to change the implementation of the function without affecting clients of the function (i.e. callers of the function). This concept is also known as *information hiding*.

6. Now it's time to test the function. As our tests are automated, all you need to do is click on "Start". If a test fails, either the test is incorrect or the function definition contains an error (or both at the same time). In this case, you should first check whether the observed behavior was really faulty or only your test is faulty. Depending on this, repair the test or the function definition until the test is executed without errors.
7. The final step consists of post-processing and refactoring the newly defined function and, if necessary, other parts of the program. Activities in this step include:
- Verification of the correctness of the signature and task description and compliance with the implementation of the function (e.g. number and task of the parameters).

- Checking the test coverage: Is the code completely covered by test cases? Is there an example for each interesting case of the input parameters ("corner cases")? Is there an example for each interesting case of the output?
- Check whether the function definition corresponds to the proposed template of the design recipe.
- Check whether there are functions or constants that are no longer required and can therefore be deleted.
- Search for redundant code, i.e. code that occurs identically or similarly in several places in the program. If necessary, identify how you can eliminate redundancy by defining constants or functions.
- Check whether there are functions with similar task descriptions and/or similar inputs/outputs. Identify and eliminate redundancy if necessary.
- Simplify conditional expressions in which different cases can be summarized.
- Test the program immediately after each refactoring. Refactoring should never change the behavior of the program. Never mix refactoring and extension/modification of the program.

Here is a graphical representation of the design recipe. The straight arrows describe the flow of information or the sequence in the initial design; the squiggly arrows describe the feedback through post-processing and the refactoring of the program.

 designrecipe.png

This mapping comes from the article "On Teaching How to Design Programs" by Norman Ramsey.

3.3.4 Programs with many functions

Most programs consist of not one but many functions. These functions are often interdependent, as one function can call another function.

You have seen a design recipe for the design of individual functions above. You should use this when designing each individual function. If you have defined many functions and global constants (variables), you should list the functions and constants in the function template that you believe will be required in the final function body.

As they cannot program all functions at once, the question arises as to the order in which they proceed. A common approach is the *top-down design*, in which the main function(s) of the application are programmed first. These functions are conveniently broken down into further auxiliary functions that do not yet exist when the function is programmed. It is therefore a good idea to always have a "wish list" of functions that still need to be programmed.

must be kept. An entry on the wish list consists of a meaningful function name, a signature and a task description. At the beginning, this list only contains the main function. If you realize that you need a new function when programming a function, add a corresponding entry to the wish list. Once you have finished with a function, look for the next function from the list that you want to implement. If the list is empty, you are done.

One advantage of top-down design is that you can break down your large design problem step by step into smaller and smaller problems until the problems become so small that you can solve them directly (in the case of function definitions, these are functions that only use built-in functions or library functions).

An important disadvantage is that you only program the details of the auxiliary functions at a relatively late stage. If you have made a mistake and the auxiliary functions cannot be implemented at all, you may have to throw a large part of your work back into the virtual wastebasket. Another important disadvantage is that you can only test your functions at a very late stage, i.e. only once all auxiliary functions have been fully implemented. One way to get around this problem is to replace an auxiliary function with a *test stub* first. A test stub is a dummy function definition that contains a predefined response.

as it is expected in the context of a test. For example, suppose you want to define an `area-of-cube` function that uses an `area-of-square` function that has not yet been programmed. To be able to test `area-of-cube` anyway, you can implement `area-of-square` provisionally using a test stub as in this example. If you later decide to implement this function, replace the test stub with the correct function definition.

```
Number -> Number
; computes the area of a cube with side length
len (check-expect (area-of-cube 3) 54)
(define (area-of-cube len) (* 6 (area-of-square len)))

Number -> Number
; computes the area of a square with side length
len (check-expect (area-of-square 3) 9)
(define (area-of-square len) (if (= len 3) 9 (error "not yet
implemented")))
```

The test stub for `area-of-square` uses the `error` function. This is well suited for documenting that a function has not yet been fully implemented. This is especially better than silently returning an incorrect result, because then you may only realize very late that you still have to implement this part.

Research
what the
abbreviations FIFO
and LIFO mean.
Discuss whether
FIFO or LIFO are
suitable for the
wish list and what
the consequences
are.

3.4 Information Hiding

The name, the signature, the task description and the tests together form the *specification* of a function. The specification should contain enough information to be able to use the function - a caller should not first have to study the implementation of the function (and perhaps even recursively the implementations of all the functions that are called in it) in order to be able to use the function.

One of the most important principles for dealing with the complexity of large programs in programming is *called information hiding*. In terms of functions, this principle states that a program is more readable, understandable and maintainable if all callers of functions rely only on the specifications of the function, but do not depend on implementation details. Furthermore, this principle states that there should be a difference between specification and implementation in such a way that there are many possible implementations of the same specification. If everyone adheres to this rule, it is guaranteed that the implementation of each function can be modified at will - as long as the specification is still adhered to, this principle ensures that the program will continue to work.

As an example, consider the `body` function from the spam mail generator of above. Here is the definition with a possible specification:

```
String String -> String

; generates the pretense for money transfer for the victim fst
last

(check-range (string-length (body "Tillman" "Rendel"))) 50 300)

(define (body fst lst)
  (string-append
    "After the last annual calculations of your GNB "
    "account activity we have determined that you, "
    fst lst
    ", are eligible to receive a tax refund of "
    "$479.30.\n" "Please submit the tax refund request "
    "(http://www...)" "and allow us 2-6 days in order to "
    "process it."))
```

An example of a caller that does not adhere to the specification and is impermissibly linked to the implementation of the would be one that defines follow-up text for the letter that refers to details of the text such as names and locations that are not mentioned in the specification.

However, if all callers adhere to the secret principle, it is ensured that the implementation of `body` can be changed to a large extent as long as it is a plausible text according to the task description that is between 50 and 300 characters long. This example also illustrates why it often makes sense to only test certain properties of the result in tests, but not the result exactly

must be specified. In the example, it is only tested that the length of the generated string is between 50 and 300 - this thus becomes part of the specification that callers can rely on. If, on the other hand, a fixed output string were tested, the test would reveal too many details about the implementation, rendering the information hiding principle useless.

4 Batch programs and interactive programs

A program, as we know it so far, always consists of expressions, function definitions and variable definitions. From the perspective of using a program, however, we can distinguish several subtypes of these programs. Two important subtypes are *batch programs* and *interactive programs*.

4.1 Batch programs

Batch programs consist of a main function. The main function uses auxiliary functions, which in turn can use other auxiliary functions. Calling a batch program means that the main function is called with the input of the program and returns with a result after the main function has finished. While the main function is being evaluated, there is no further interaction with the user.

Batch programs are often given their input in the form of command line parameters or standard streams. Further input can come from files whose names have been passed as command line parameters, for example. The output is often output via the standard output stream (stdout). A good example of batch programs are the command line tools of the Unix world, such as *ls*, *grep* or *find*. There is also a batch version of DrRacket.

program, *raco*, see <http://docs.racket-lang.org/raco/index.html>.

In a sense, batch programs behave like the functions we define: If you give them an input, they calculate independently for a while and then return the result. You already know that functions can be linked and combined very flexibly. This is also the case with batch programs, which are often combined with each other in scripts or on the command line. For example, if you want to know the list of all files in the current directory that were modified in March in a Unix shell and then sort them by size, you can achieve this by combining simple batch programs such as *ls*, *grep* and *sort*:

```
$ ls -l | grep "Mar" | sort +4n
-rwxr-xr-x 1 klaus Admin 193 Mar 7 19:50 sync.bat
-rw-r--r-- 1 klaus Admin 317 Mar 30 12:53 10-modules.bak
-rw-r--r-- 1 klaus Admin 348 Mar 8 12:30 arrow-big.png
-rw-r--r-- 1 klaus Admin 550 Mar 30 13:02 10-
modules.scrbl
-rw-r--r-- 1 klaus Admin 3611 Mar 8 12:35 arrow.png
-rw-r--r-- 1 klaus Admin 4083 Mar 27 15:16 marburg-
utils.rkt
-rw-r--r-- 1 klaus Admin 4267 Mar 27 15:15 marburg-
utils.bak
-rw-r--r-- 1 klaus Admin 7782 Mar 30 13:02 10-
modules.html
-rw-r--r-- 1 klaus Admin 14952 Mar 2 13:49 rocket-s.jpg
```

Research,
which are *standard
streams* like *stdin*
and *stdout* in Unix.

A good example of a batch program in BSL is the `letter` function from section §3.1 "Functional decomposition". We call it in the interaction area with the desired input and then receive the output. It is also possible to use this program as a batch program outside of DrRacket. However, in this case the input must be passed differently, namely in the form of command line parameters. The following program shows how this can be done. In order to access the command line parameters, some language structures are required that you are not yet familiar with. Therefore, do not try to understand the details of the program below; it is only intended to illustrate that Racket programs can be executed as batch programs without DrRacket.

```
(require racket/base)

(define (letter fst lst signature-
  name) (string-append
  (opening
    lst) "\n"
  (body fst lst)
  "\n"
  (closing signature-name)))

(define (opening lst)
  (string-append "Dear Mr./Mrs. " lst ","))

(define (body fst
  lst) (string-append
  "After the last annual calculations of your GNB "
  "account activity we have determined that you, "
  fst lst
  ", are eligible to receive a tax refund of $479.30.\n"
  "Please submit the tax refund request (http://www...) "
  "and allow us 2-6 days in order to process it.))

(define (closing signature-
  name) (string-append
  "With best
  regards,\n"
  signature-name))

(define args (current-command-line-arguments))

(if (= (vector-length args) 3)
  (display (letter (vector-ref args 0)
    (vector-ref args 1)
    (vector-ref args 2)))
  (error "Please pass exactly three parameters"))
```

If this program is saved in the `letter.rkt` file, you can use

call this in a Unix or DOS shell, for example:
\$ racket letter.rkt Tillmann Rendel Klaus
and you will receive the following output:
Dear Mr./Mrs. Rendel,
After the last annual calculations of your GNB account
activity we
have determined that you, TillmannRendel, are eligible to
receive a tax refund of \$479.30.
Please submit the tax refund request (<http://www...>) and allow
us 2-6 days in order to process it.
With best regards,
Klaus

You can now link this batch program to other command line programs; for example, you can link it to the `wc` program to determine the number of words in the generated text:

```
$ racket letter.rkt Tillmann Rendel Klaus | wc -w  
35
```

The program `racket`, which is called in this command, is the command-line version of DrRacket. You can, of course, also run Racket programs without the Racket environment. For example, you can use the Racket - Program file menu item to create an executable file that can be executed independently of DrRacket on any computer with the appropriate operating system.

If you want to read or write files in your BSL programs, you can use the [2http/batch-io](http://batch-io) teach pack.

In summary, it can be said that the strength of batch programs is that they can be easily combined with other batch programs in a variety of ways, either on the command line or automatically within batch files or shell scripts. By definition, batch programs are not suitable for interacting with the user. However, they are very often building blocks for interactive programs.

4.2 Interactive programs

Interactive programs consist of several main functions and a printout that informs the computer which of these functions processes which type of input and which of the functions produces output. Each of these main functions can of course use auxiliary functions. We will use the Universe Teachpack to construct interactive programs.

4.2.1 The Universe Teachpack

The "universe" Teachpack supports the construction of interactive programs: Programs that react to time signals, mouse clicks, keyboard input or network traffic and generate graphical output. We will demonstrate the functionality of this

Teachpacks using the following example. Please try out what this program does before reading on.

```
; WorldState is a number
; interp. the countdown when the bomb explodes

WorldState -> WorldState
; reduces countdown by one whenever a clock tick
occurs (check-expect (on-tick-event 42) 41)

(define (on-tick-event world) (- world 1))

WorldState Number Number MouseEvent -> WorldState
decreases countdown by 100 when mouse button is pressed
(check-expect (on-mouse-event 300 12 27 "button-down")
200)
(check-expect (on-mouse-event 300 12 27 "button-up") 300)

(define (on-mouse-event world mouse-x mouse-y mouse-event)
  (if (string=? mouse-event "button-down")
      (- world 100)
      world))

WorldState -> Image
; renders the current countdown t as an
image (check-expect (image? (render 100))
true) (check-expect (image? (render 0))
true) (define (render world)
  (if (> world 0)
      (above (text (string-append "Countdown for the bomb: "
                                   (number->string world))
                    30 "red")
              (text "Click to disarm!" 30
                    "red"))) (text "Boooom!!" 60 "red")))

WorldState -> Boolean
; the program is over when the countdown is <=
0 (check-expect (end-of-the-world 42) false)
(check-expect (end-of-the-world 0) true)
(define (end-of-the-world world) (<= world 0))

install all event handlers; initialize world state to 500
(big-bang 500
  (on-tick on-tick-event 0.1)
```



```
(on-mouse on-mouse-event)
(to-draw render)
(stop-when-end-of-the-world))
```

A central concept in interactive programs is that of the *world state*. An interactive application is typically in a certain *state*, which is called the we call WorldState.

The state in a Pacman game, for example, includes the current position of all the game pieces and the current score. In the program above, the current state only consists of a number that contains the current countdown to the bomb explosion.

The state of an interactive program changes when certain *events* occur. Events can be, for example, the pressing of keys on the keyboard, actions with the mouse or the expiry of certain time intervals (*timers*). Events are reacted to in the form of *event handlers*. An event handler is a function that receives the current WorldState and other information about the event that has occurred as input and produces a new WorldState as output.

In the example above, two event handlers are defined, `on-mouse-event` for mouse events and `on-tick-event` for timer events. The interval between two Timer events will be set to 0.1 seconds later. In the application above, we have decided to reduce the current countdown by one each time a timer event occurs.

The "`on-mouse-event`" function receives the following as input in addition to the current WorldState also requires the position of the mouse and the exact type of mouse event as input. In our example, the countdown is decreased by 100 each time the mouse is clicked.

The `render` function produces an image from the current WorldState, which graphically displays the current state of the program. This function is called each time the current WorldState has changed.

The last function, `end-of-world`, is used to check whether the application application is to be terminated. To do this, it checks the WorldState and produces a truth value as a result. In our example, we want to end the application after the bomb has exploded.

The last `big-bang` expression is used to initialize the WorldState and install all event handlers. The `big-bang` operator is a special form, i.e. not a normal BSL function. The first argument of `big-bang` is the initial WorldState. Since we have decided that in this application the WorldState is a number that represents the current countdown, we choose an initial countdown, `500`.

The sub-expressions such as `(on-tick on-tick-event)` are not function calls but special clauses of the `big-bang` operator in which the event handler functions and the functions for drawing (`to-draw`) and terminating (`stop-when`) the program are specified. Of course, we could have given the event handler functions other names - the `big-bang` is the place where you define which event type is linked to which event handler function.

If you use Pacman please search for an online version of Pacman on the Internet immediately and play a round!

will.

Some of these clauses expect further arguments in addition to the name of the event handler function. For example, the number in `(on-tick on-tick-event 0.1)` indicates that a timer event should be triggered every 0.1 seconds. You can also omit this specification; the default value is then assumed to be 1/28 second. There are a number of other clauses and variants of the above clauses; for more information, see the documentation of the universe teachpack. With the exception of the `to-draw` clauses, all of these clauses are optional; if no event handler is installed for an event type, these events are ignored.

If an interactive program (hereinafter also referred to as a *world program*) is ended, the `big-bang` expression returns the last `WorldState`.

Let us assume, for example, that the following events occur in the following order:

1. a timer event
2. a timer event
3. a mouse event (the mouse was moved to the position (123,456))
4. a mouse event (a mouse button was pressed at the position (123,456))

Let's assume that the state before the occurrence of these events is `500` and the current state after the occurrence of the respective event:

1. `499`, i.e. `(on-tick-event 500)`
2. `498`, thus `(on-tick-event 499)`
3. `498`, thus `(on-mouse-event 498 123 456 "move")`
4. `398`, thus `(on-mouse-event 498 123 456 "button-down")`

Since event handlers are only functions, the final result can also be calculated using the *function composition* of the event handlers:

```
(on-mouse-event
  (on-mouse-event
    (on-tick-event
      (on-tick-event 500))
    123 456 "move")
  123 456 "button-down")
```

The sequence of events therefore determines the order in which the event handler functions are connected one after the other. If you want to simulate the occurrence of a certain sequence of events in tests, for example, you can do this simply by composing the corresponding event handlers. This is important because many interesting situations (that you want to test) only arise in the composition of different events.

5 Data definition by alternatives: Sum types

The data types we have learned about and used so far include numbers, strings, truth values and data types defined in informal data definitions such as `temperature`.

In order to program realistic applications, it is helpful to have a larger and flexibly expandable repertoire of data types. In this chapter, we look at how data types can be used to differentiate between different situations and how these data types influence the design of programs.

5.1 Enumeration types

It is often the case that a data type only contains a finite enumerable set of values. We call such data types *enumeration types* or *enumeration types*. Here is an example of how an enumeration type is defined:

```
A TrafficLight shows one of three colors:
; - "red"
; - "green"
; - "yellow"
; interp. each element of TrafficLight represents which col-
ored
; bulb is currently turned on
```

Programming with enumeration types is very simple. If a function has a parameter that has an enumeration type, then the program typically contains a case distinction in which all alternatives of the enumeration type are treated separately from each other. Here is an example:

```
; TrafficLight -> TrafficLight
; given state s, determine the next state of the traffic light

(check-expect (traffic-light-next "red")

"green") (define (traffic-light-next s)
  (cond
    [(string=? "red" s) "green"]
    [(string=? "green" s) "yellow"]
    [(string=? "yellow" s) "red"])))
```

Sometimes a function is only interested in a subset of the possible values of an enumeration type. In this case, only the interesting cases are queried and the rest are ignored. An example of this is the `on-mouse-event` function from §4.2.1 "The Universe Teachpack", which receives an input from the `MouseEvent` enumeration type. `MouseEvent` is defined as follows:

```
A MouseEvt is one of these strings:
```

```
; - "button-down"
; - "button-up"
; - "drag"
; - "move"
; - "enter"
; - "leave"
```

In the `on-mouse-event` function, we are only interested in the `"button-down"` alternative. For all other alternatives of `MouseEvent`, we leave the status unchanged.

It is also possible that a function parameter has an enumeration type, but the function still does not differentiate between cases because an auxiliary function is called for this purpose. Nevertheless, it is a good heuristic to start in step 3 of the design recipe from section §3.3.3 "Design recipe for function definition" with a function template that distinguishes all cases of the parameter. For example, the template for the `traffic-light-next` function from above would look like this:

```
(define (traffic-light-next
  s) (cond
      [(string=? "red" s) ...]
      [(string=? "green" s) ...]
      [(string=? "yellow" s) ...]))
```

As you can see, you can generate a large part of a function definition quasi mechanically by systematically following the steps from the design recipe.

5.2 Interval types

Consider a program that should show a UFO landing. A program for this could look like the following:

```
; constants:
(define WIDTH 300)
(define HEIGHT 100)

; visual constants:
(define MT (empty-scene WIDTH HEIGHT))
(define UFO
  (overlay (circle 10 "solid" "green")
    (rectangle 40 2 "solid" "green")))

(define BOTTOM (- HEIGHT (/ (image-height UFO) 2)))
```

```
A WorldState is a number.
; interp. height of UFO (from top)
```

```
WorldState -> WorldState
```

```

; compute next location of UFO
(define (nxt y)
  (+ y 1))

WorldState -> Image
; place UFO at given height into the center of MT
(define (render y)
  (place-image UFO (/ WIDTH 2) y MT))

WorldState -> Boolean
; returns #true when UFO has reached the bottom, i.e. y > BOT-
TOM
(define (end-of-the-world y) (> y BOTTOM))

; wire everything together; start descend at position 0
(big-bang 0 (on-tick nxt) (to-draw render) (stop-when end-
of- the-world))

```

Now consider the following extension of the problem:

The status line should say "descending" when the UFO's height is above one third of the height of the canvas. It should switch to "closing in" below that. And finally, when the UFO has reached the bottom of the canvas, the status should notify the player that the UFO has "landed."

The data type hidden in this problem is not an enumeration type, because there are an infinite number of different heights (or, if we only count the integers, so many different heights that it would be impractical to define an enumeration type for this).

Therefore, we use *intervals* to define additional structure on ordered data types such as numbers or strings. In the following, we focus on (integer or non-integer) numbers. An interval is defined by its *boundaries*. An interval either has a lower and upper bound, or it has only one of these bounds and is open to the other side.

In our example, we can express the data definition for the WorldState as an interval to have a data definition that better captures the intent of the problem.

```

; constants:
(define CLOSE (* 2 (/ HEIGHT 3)))

```

```

A WorldState is a number. It falls into one of three
intervals:
; - between 0 and CLOSE
; - between CLOSE and BOTTOM
; - at BOTTOM
; interp. height of UFO (from top)

```

Not all functions that receive an interval type as an argument use

this additional structure. For example, the `render` function from above can remain as it is because the interval is only relevant for the status display but not for the Ufo.

However, functions that require the additional structure of the interval typically contain a `cond` expression that distinguishes the different intervals.

```
WorldState -> Image
add a status line to the scene create by render
(define (render/status y)
  (cond
    [(<= 0 y CLOSE) (above (text "descending" 12 "black") (render y))]
    [(< CLOSE y BOTTOM) (above (text "closing
in" 12 "black") (render y))]
    [(= y BOTTOM) (above (text "landed" 12 "black") (render y))]))
```

It is advisable to include this `cond` expression directly in the template as part of the template construction from the design recipe. However, the case differentiation does not necessarily take place first, but can also take place deeper in the function body. This is useful in our example, because we have violated the DRY principle (if we want to change the color of the text to "red", for example, we would have to change three lines). It is therefore advantageous to pull the conditional expression inwards:

```
WorldState -> Image
add a status line to the scene create by render
(define (render/status y)
  (above
    (text
      (cond
        [(<= 0 y CLOSE) "descending"]
        [(< CLOSE y BOTTOM) "closing
in"] [(= y BOTTOM) "landed"])
      12
      "black")
    (render y)))
```

Now only the `big-bang` expression needs to be adapted so that `render/status` and no longer `render` is used for drawing.

```
; wire everything together; start descend at position 0
(big-bang 0 (on-tick nxt) (to-draw render/status)
(stop- when end-of-the-world))
```

In addition to new function templates, intervals also provide us with stops for testing: Typically, you want to have a test case for each interval and especially for the interval limits.

5.3 Sum types

Interval types distinguish between different subsets of numbers (or other ordered values). Enumeration types enumerate the different elements of the type value by value.

Sum types generalize interval types and enumeration types. With sum types, existing data types and individual values can be combined with each other as required. A sum type specifies various alternatives, of which each value of this type fulfills exactly one alternative.

As an example, consider the `string->number` function, which converts a string to a number if this is possible and returns `#false` otherwise.

Here is the definition of a sum type that describes the behavior of this function:

```
A MaybeNumber is one of:  
; - #false  
; - a Number  
; interp. a number if successful, else false.
```

This allows us to define the following signature for `string->number`:

```
String -> MaybeNumber  
; converts the given string into a number;  
produces #false if impossible  
(define (string->number str) ...)
```

Sum types are sometimes also called union types.
In HTDP/2e they are called *itemizations*.

In the documentation of the HTDP languages, the signature of `string->number` so indicated:

```
String -> (union Number #false)
```

The operator `union` stands for the "on-the-fly" construction of an anonymous sum type with the same meaning as our `MaybeNumber` above.

What do you do with a value that has a sum type? As with enumeration and interval types, the only meaningful operation is typically the one that separates the different alternatives, for example in the context of a `cond` expression. Here is an example:

```
MaybeNumber -> MaybeNumber  
; adds 3 to a if it is a number; returns #false  
otherwise (check-expect (add3 5) 8)  
(check-expect (add3 #false)  
#false) (define (add3 a)  
  (cond [(number? a) (+ a  
    3)] [else #false]))
```

Functions with sum types differ somewhat in their design methodology from the functions we have learned about so far. We will therefore go through our design recipe again (§3.3.3 "Design recipe for function definition") and describe how the design of functions with sum types differs from the general design recipe.

The tax on an item is either an absolute tax of 5 or 10 currency units, or a linear tax. Design a function that computes the price of an item after applying its tax.

5.3.1 Design with sum types

We supplement the design recipe for the design of functions as follows:

1. If the problem divides values into different classes, these classes should be explicitly defined by a data definition.

Example: The problem definition above distinguishes between three different types of tax. This motivates the following data definition:

```
A Tax is one of
; - "absolute5"
; - "absolute10"
; - a Number representing a linear tax rate in percent
```

2. Nothing changes in the second step, except that the defined data type is typically used in the signature.

Example:

```
Number Tax -> Number
; computes price of an item after applying tax
(define (total-price itemprice tax) 0)
```

3. With regard to the tests, it is advisable to have at least one example for each alternative of the data type. For intervals, the limits of the intervals should be tested. If there are several parameters with sum types, all combinations of the alternatives should be tested.

Example:

```
(check-expect (total-price 10 "absolute5") 15)
(check-expect (total-price 10 "absolute10") 20)
(check-expect (total-price 10 25) 12.5)
```

4. The biggest innovation is the template for the function. In general, *the structure of the functions follows from the structure of the data*. This means that in most cases the function body starts with a `cond` expression that
Why is the order of the branches of the `cond` expression important in this example?


```
(define (total-price itemprice tax)
  (cond [(number? tax) ...]
        [(string=? tax "absolute5") ...]
        [(string=? tax "absolute10") ...]))
```

5. In the fifth step, the ... placeholders are replaced with the correct code. Here it makes sense to go through and implement each branch of the `cond` expression individually and one after the other. The advantage of this type of implementation is that you only ever have to worry about one of the cases and can ignore all other cases.

This distinction of cases is an example of a more general design concept for complex systems called *separation of concerns*.

You may realize during this step that it makes sense to pull the `cond` expression inside (similar to `create-rocket-scene-v6` in section §2.6.2 "DRY Redux"), or maybe you don't need to distinguish all cases at all, or maybe you want to outsource the distinction to a helper function - nevertheless, it usually makes sense to start with this template, even if its function looks different in the end.

Example:

```
(define (total-price itemprice tax)
  (cond [(number? tax) (* itemprice (+ 1 (/ tax 100)))]
        [(string=? tax "absolute5") (+ itemprice 5)]
        [(string=? tax "absolute10") (+ itemprice 10)]))
```

6. Nothing changes in the last step, but check that you have defined test cases for all alternatives.

5.4 Distinguishability of the alternatives

What if we had a continuous spectrum instead of two absolute tax rates in our last example? We could change our data definition as follows:

```
A Tax is one of
; - a Number representing an absolute tax in currency units
; - a Number representing a linear tax rate in percent
```

So far so good - but how can we distinguish between these cases in a `cond` expression? We have the predicate `number?`, but this does not allow us to distinguish between these two cases.

In our sum types, it is important to be able to clearly distinguish which alternative has been selected in a value that has a sum type. Therefore the sets of possible values for each alternative must be disjoint.

If they are not disjoint, additional information must be stored for each value to indicate which alternative this value belongs to: a so-called *tag*. With the means we have learned so far, we cannot express the above desired data type in a meaningful way. In the next chapter, however, we will get to know a language construct that can be used to define such *tags* and thus also such data types in a structured way.

The variant of the Sum types that we use are therefore also referred to as *untagged unions*.

6 Data definition through decomposition: Product types

Suppose you want to use the "universe" teachpack to write a program that simulates a ball bouncing back and forth between four walls. For the sake of simplicity, let's assume that the ball moves constantly at two pixels per time unit.

If you follow the design recipe, your first task is to define a data representation for all the things that change. For our constant velocity ball, these are two properties: The current position and the direction in which the ball is moving.

However, the `WorldState` in the "universe" teachpack is only a single value. The values that we know so far are always individual numbers, images, strings or truth values - even sum types do not change this. We must therefore somehow be able to combine several values in such a way that they can be treated as a single value.

It would be conceivable, for example, to encode two numbers into a string and decode them again if necessary (in theoretical computer science, such techniques are known as *Gödelization*), but these techniques are unsuitable for practical programming.

Every higher programming language has mechanisms for combining several pieces of data into one datum and, if necessary, breaking them down again into their constituent parts. In BSL, there are *structures* (*structs*) for this purpose. Structure definitions are often called *products* because they correspond to the cross product of sets in mathematics. This analogy also gives rise to the name 'sum type' from the last chapter, as the union of sets is also referred to as the sum of sets. In some languages, structures are also called *records*.

6.1 The `posn` structure

A position in an image is clearly identified by two numbers: The distance from the left edge and the distance from the top edge. The first number is called the *x-coordinate* and the second the *y-coordinate*.

In BSL, such positions are represented using the `posn` structure. A `posn` (pronunciation: position) is therefore a value that contains *two* values.

We can create an *instance* of the `posn` structure with the function `make-posn`. The signature of this function is *Number Number -> Posn*.

Example: These three expressions each create an instance of the `posn` structure.

```
(make-posn 3 4)
(make-posn 8 6)
(make-posn 5 12)
```

A `posn` has the same status as numbers or strings in the sense that functions can consume or produce instances of structures.

This part of the script is based on [HTDP/2e] chapter 5

Would it make sense, variants of the "universe" teachpack, in which the `WorldState` is represented by two values, for example?

In German this would be it is actually correct to speak of an *instance* of a structure. However, it has become customary to speak of an instance, analogous to the word *instance* in English, which is why we will also use this grammatically questionable terminology.

Now consider a function that calculates the distance of a position from the upper left corner of the image. Here is the signature, task description and header of such a function:

```
Posn -> Number
; to compute the distance of a-posn to the origin
(define (distance-to-0 a-posn) 0)
```

What is new about this function is that it only has one parameter, `a-posn`, in which both coordinates are passed. Here are some tests that illustrate what `distance-to-0` is supposed to calculate. The distance can of course be calculated using the well-known Pythagoras formula.

```
(check-expect (distance-to-0 (make-posn 0 5)) 5)
(check-expect (distance-to-0 (make-posn 7 0)) 7)
(check-expect (distance-to-0 (make-posn 3 4)) 5)
(check-expect (distance-to-0 (make-posn 8 6)) 10)
```

What does the function body of `distance-to-0` look like? Obviously, we need to extract the x and y coordinates from the parameter `a-posn` for this purpose. There are two functions `posn-x` and `posn-y` for this purpose. The first function extracts the x-coordinate, the second extracts the y-coordinate. Here are two examples that illustrate these functions:

```
> (posn-x (make-posn 3 4))
3
> (posn-y (make-posn 3 4))
4
```

We now know enough to implement `distance-to-0`. As an intermediate step, we define a template for `distance-to-0` in which the expressions for extracting the x and y coordinates are predefined.

```
(define (distance-to-0 a-
  posn) (... (posn-x a-posn)
  ...
  ... (posn-y a-posn) ...))
```

Based on this template, it is now easy to complete the function definition:

```
(define (distance-to-0 a-
  posn) (sqrt
  (+ (sqrt (posn-x a-posn))
    (sqrt (posn-y a-posn)))))
```

6.2 Structure definitions

Structures such as `posn` are not normally built into a programming language. Instead, the language only provides a mechanism for

to define your own structures. The number of available structures can therefore be extended by any programmer.

A structure is created by a special structure definition. Here is the definition of the `posn` structure in BSL:

```
(define-struct posn (x y))
```

In general, a structure definition has this form:

```
(define-struct StructureName (FieldName ... FieldName))
```

The keyword `define-struct` means that a structure is defined here. This is followed by the name of the structure. This is followed by the names of the structure's *fields*, enclosed in brackets.

In contrast to a normal function definition, a structure definition defines an entire set of functions as follows:

- A *constructor* - a function that has as many parameters as the structure has fields and returns an instance of the structure. In the case of `posn`, this function is called `make-posn`; in the general case, it is called `make-StructureName`, where `StructureName` is the name of the structure.
- One *selector* per field of the structure - a function that reads the value of a field from an instance of the structure. In the case of `posn`, these functions are called `posn-x` and `posn-y`; in general, they are called `StructureName-FieldName`, where `StructureName` is the name of the structure and `FieldName` is the name of the field.
- A *structure predicate* - a boolean function that calculates whether a value is an instance of this structure. In the case of `posn`, this predicate is called `posn?`; in general, it is called `StructureName?`, where `StructureName` is the name of the structure.

The rest of the program can use these functions as if they were primitive functions.

6.3 Nested structures

A ball moving at constant speed in two-dimensional space can be described by two properties: Its location and the speed and direction in which it is moving. We already know how to describe the location of an object in two-dimensional space: Using the `posn` structure. There are different ways to represent the speed and direction of an object. One of these ways is to specify a motion vector, for example in the form of a structure definition like this one:

```
(define-struct vel (deltax deltay))
```

The name `vel` stands for *velocity*; `deltax` and `deltay` describe how many points on the x- and y-axis the object moves per time unit.

Based on these definitions, we can, for example, calculate how the location of an object changes:

```
; posn vel -> posn
; computes position of loc after applying v
(check-expect (move (make-posn 5 6) (make-vel 1 2)) (make-
posn 6 8))
(define (move loc
  v) (make-posn
    (+ (posn-x loc) (vel-deltax v))
    (+ (posn-y loc) (vel-deltay
v))))
```

The degree of change of units is often referred to as *delta* in computer science (and elsewhere), hence the name of the fields.

How can we now represent a moving ball itself? We have seen that it is described by its location and its motion vector. One way of representing it would be this:

```
(define-struct ball (x y deltax deltax))
```

Here is an example of a ball in this representation:

```
(define some-ball (make-ball 5 6 1 2))
```

However, the relationship between the fields is lost in this representation: the first two fields represent the location, the other two fields the movement. A practical consequence is that it is also cumbersome to call functions that do something with velocities and/or movements but know nothing about balls, such as `move` above, because we always have to pack the data manually into the correct structures first. To move `some-ball` with the help of `move`, we would have to write expressions like this one:

```
(move (make-posn (ball-x some-ball) (ball-y some-ball))
      (make-vel (ball-deltax some-ball) (ball-deltay
some-
ball)))
```

A better representation *nests* structures within each other:

```
(define-struct ball (loc vel))
```

This definition is not yet nested - we will make this clear shortly with data definitions for structures. We can see the nesting when we create instances of this structure. The example ball `some-ball` is constructed by nesting the constructors inside each other:

```
(define some-ball (make-ball (make-posn 5 6) (make-vel 1 2)))
```

In this representation, the logical grouping of the data remains intact. It is now also easier to call `move`:

```
(move (ball-loc some-ball) (ball-vel some-ball))
```

In general, by nesting structures, data can be represented here in the form of a tree.

6.4 Data definitions for structures

The purpose of a data definition for structures is to describe what kind of data each field may contain. For some structures, the associated data definition is quite obvious:

```
(define-struct posn (x y))  
A Posn is a structure: (make-posn Number Number)  
; interp. the number of pixels from left and from top
```

Here are two plausible data definitions for `vel` and `ball`:

```
(define-struct vel (deltax deltax))  
; a Vel is a structure: (make-vel Number Number)  
; interp. the velocity vector of a moving object  
  
(define-struct ball (loc vel))  
; a Ball is a structure: (make-ball Posn Vel)  
; interp. the position and velocity of a ball
```

However, a structure does not necessarily have exactly one associated data definition. For example, we can view balls not only in two-dimensional space, but also in one- or three-dimensional space. In one-dimensional space, position and velocity can each be represented by a number. We can therefore define

```
; a Ball1d is a structure: (make-ball Number Number)  
; interp. the position and velocity of a 1D ball
```

Structures can therefore be "reused". Should you always do this if possible?

In principle, we would only need a single structure with two fields and could thus encode all product types with two components. For example, we could also dispense with `vel` and `ball` and just use `posn` instead:

```
; a Vel is a structure: (make-posn Number Number)  
; interp. the velocity vector of a moving object  
  
(define-struct ball (loc vel))  
; a Ball is a structure: (make-posn Posn Vel)  
; interp. the position and velocity of a ball
```

In principle, we can even express *all* product types, even those with more than two fields, with the help of `posn` by nesting `posn`.

Example:

```
; a 3DPosn is a structure: (make-posn Number (make-posn Number
Number))
; interp. the x/y/z coordinates of a point in 3D space
```

The LISP language was based on this principle: it had only one universal data structure, the so-called *cons cell*, which was used to represent all types of products. The cons cell corresponds to the following structure definition in BSL:

```
(define-struct cons-cell (car cdr))
```

Is this reuse of structure definitions a good idea? The price you pay for this is that you can no longer differentiate between the different data because there is only one predicate per structure definition. Our recommendation is therefore only to use structure definitions in several data definitions if the different data have a common semantic concept. In the example above, there is the common concept of the ball in n-dimensional space for `ball` and `ball1d`. However, there is no meaningful common concept for `Ball` and `Posn`; therefore, it does not make sense to use a common structure definition.

Another useful criterion for deciding on reuse is the question of whether it is important to be able to distinguish the different data with a predicate - if so, you should use separate structures in each case.

The names `cons`, `car` and `cdr` have a history, but this is not relevant for us. `car` is simply the name for the first component and `cdr` for the second component of the pair.

6.5 Case study: A ball in motion

Try out what the following program does. Understand how structures and data definitions were used to structure the program!

```
(define WIDTH 200)
(define HEIGHT 200)
(define BALL-IMG (circle 10 "solid" "red"))
(define BALL-RADIUS (/ (image-width BALL-IMG) 2))

(define-struct vel (delta-x delta-y))
; a Vel is a structure: (make-vel Number Number)
; interp. the velocity vector of a moving object

(define-struct ball (loc velocity))
; a Ball is a structure: (make-ball Posn Vel)
; interp. the position and velocity of an object

; Posn Vel -> Posn
```

```

; applies q to p and simulates the movement in one clock
tick (check-expect (posn+vel (make-posn 5 6) (make-vel 1 2))
  (make-posn 6 8))
(define (posn+vel p q)
  (make-posn (+ (posn-x p) (vel-delta-x q))
    (+ (posn-y p) (vel-delta-y q))))

; Ball -> Ball
; computes movement of ball in one clock tick
(check-expect (move-ball (make-ball (make-posn 20 30)
  (make-vel 5 10)))
  (make-ball (make-posn 25 40)
    (make-vel 5 10)))
(define (move-ball ball)
  (make-ball (posn+vel (ball-loc ball)
    (ball-velocity ball))
    (ball-velocity ball)))

A Collision is either
; - "top"
; - "down"
; - "left"
; - "right"
; - "none"
; interp. the location where a ball collides with a wall

; Posn -> Collision
; detects with which of the walls (if any) the ball
collides (check-expect (collision (make-posn 0 12)) "left")
(check-expect (collision (make-posn 15 HEIGHT)) "down")
(check-expect (collision (make-posn WIDTH 12)) "right")
(check-expect (collision (make-posn 15 0)) "top")
(check-expect (collision (make-posn 55 55)) "none")
(define (collision posn)
  (cond
    [(<= (posn-x posn) BALL-RADIUS) "left"]
    [(<= (posn-y posn) BALL-RADIUS) "top"]
    [(>= (posn-x posn) (- WIDTH BALL-RADIUS)) "right"]
    [(>= (posn-y posn) (- HEIGHT BALL-RADIUS)) "down"]
    [else "none"]))

Vel Collision -> Vel
; computes the velocity of an object after a
collision (check-expect (bounce (make-vel 3 4)
  "left")
  (make-vel -3 4))

```



```

(check-expect (bounce (make-vel 3 4) "top")
  (make-vel 3 -4))
(check-expect (bounce (make-vel 3 4) "none")
  (make-vel 3 4))
(define (bounce vel collision)
  (cond [(or (string=? collision "left")
    (string=? collision
      "right"))
    (make-vel (- (vel-delta-x vel)
      (vel-delta-y vel))]
    [(or (string=? collision "down")
      (string=? collision
        "top")) (make-vel (vel-delta-x
        vel)
        (- (vel-delta-y vel)))]
    [else vel]))

; WorldState is a Ball

WorldState -> Image
; renders ball at its position
(check-expect (image? (render INITIAL-BALL)) #true)
(define (render ball)
  (place-image BALL-IMG
    (posn-x (ball-loc ball))
    (posn-y (ball-loc ball))
    (empty-scene WIDTH
      HEIGHT)))

WorldState -> WorldState
; moves ball to its next location
(check-expect (tick (make-ball (make-posn 20 12) (make-
  vel 1 2)))
  (make-ball (make-posn 21 14) (make-vel 1 2)))
(define (tick ball)
  (move-ball (make-ball (ball-loc ball)
    (bounce (ball-velocity
      ball) (collision
        (ball-
  loc ball))))))

(define INITIAL-BALL (make-ball (make-posn 20 12)
  (make-vel 1 2)))

(define (main ws)
  (big-bang ws (on-tick tick 0.01) (to-draw render)))

```

```
; start with: (main INITIAL-BALL)
```

Try out what happens if the ball flies exactly into a corner of the pitch. How can you solve this problem? In view of this problem, reflect on why it is important to always test the extreme cases (*corner cases*) ;-)

6.6 Tagged Unions and Maybe

As already mentioned in section §5.4 "Distinguishability of alternatives", it is important for sum types that the individual alternatives are distinguishable. For example, take the data type `MaybeNumber` from section §5.3 "Sum types":

```
A MaybeNumber is one of:
; - a Number
; - #false
; interp. a number if successful, else false.
```

The concept that a calculation fails, or that an input is optional is very common. The above summation type is a good solution here.

Now we would like to use the same idea for optional truth values and therefore define them:

```
A BadMaybeBoolean is one of:
; - a Boolean
; - #false
; interp. a boolean if successful, else false.
```

The two variants are obviously not disjoint: The union of { `#true`, `#false` } and { `#false` } would again only have two elements.

To prevent this, we can use a *tag* to mark which alternative a value belongs to. To do this, we first define the structures

```
(define-struct some
(value)) (define-struct
none ())
```

We use the first structure to mark that the variant is an existing value - we use the second structure to mark that no value exists:

```
A MaybeBoolean is one of:
; - a structure: (make-some Boolean)
; - (make-none)
; interp. a boolean if successful, else none.
```

The sum type `MaybeBoolean` now has three elements { `(make-none)`, `(make-some #true)`, `(make-some #false)` }. The two alternatives can now be clearly distinguished. In particular, we can now simply implement a function that expects a `MaybeBoolean` according to the design recipe (in the version for sum types).

As a reminder:
However, we can only define the sum type in this way, as the value `#false` is not in `Number` and no number from `Number` is contained in the one-element set { `#false` } is contained. The two sets are disjoint.

If we `MaybeNumber` now also on these Define way and then the sum of `MaybeBoolean` and `MaybeNumber`, we would have the original problem again: It is not

6.7 Extension of the design recipe

The previous examples have shown that many problems require suitable data structures to be developed in parallel with functions. This means that the steps of the design recipe from section §3.3.3 "Design recipe for function definition" change as follows:

1. If information appears in a problem description that belongs together or describes a whole, structures are required. The structure corresponds to the "whole" and has a field for each "relevant" property.

A data definition for a field must specify a name for the set of instances of the structure described by this data definition. It must describe which data is permitted for which field. Only names of built-in data types or data you have already defined should be used for this purpose.

In the data definition, enter examples of instances of the structure that correspond to the data definition.

2. Nothing changes in the second step.
3. In the third step, use the examples from the first step to design tests. If one of the fields of a structure that is an input parameter has a sum type, there should be test cases for all alternatives. For intervals, the endpoints of the intervals should be tested.
4. A function that receives instances of structures as input will in many cases read the fields of the structure instance. To remind you of this possibility, the template for such functions should contain the selector expressions (for example `(posn-x param)` if `param` is a parameter of type `Posn`) for reading the fields.

However, if the value of a field is itself an instance of a structure, you should *not* include selector expressions for the fields of this nested structure instance in the template. In most cases, it is better to outsource the functionality relating to this substructure to a new auxiliary function.

5. Use the selector expressions from the template to implement the function. Please note that you may not need the values of all fields.
6. Test as soon as you have written the function header. Check that all tests fail at this point (except for those where the dummy value used happens to be correct). This step is important because it protects you from errors in the tests and ensures that your tests actually test a non-trivial property.

Test until all expressions in the program have been executed at least once during testing. The code coloring in DrRacket after testing will help you with this.

7 Data definition by alternatives and decomposition: Algebraic data types

In the last two chapters, we learned about two new types of data definitions: Sum types (in the form of enumerations and intervals), with which you can choose between different alternatives. Product types (in the form of data definitions for structures), which can be used to break down data.

In many cases, it makes sense to combine totals and products, for example to define totals types where some alternatives have a product type; or product types where some fields have totals types.

We call combinations of sum and product types *algebraic data types*, *ADT* for short.

7.1 Example: Collisions between shapes

Suppose you want to program a computer game in which game pieces appear in different geometric shapes, for example circles and Rectangles. We can model these as follows, for example:

```
(define-struct gcircle (center radius))
; A GCircle is (make-gcircle Posn Number)
; interp. the geometrical representation of a circle

(define-struct grectangle (corner-ul corner-dr))
A GRrectangle is (make-grectangle Posn Posn)
; interp. the geometrical representation of a rectangle
; where corner-ul is the upper left corner
; and corner-dr the down right corner
```

We choose the name `gcircle` (for *geometric circle*) to avoid a name conflict with the `circle` function from the `image.ss` teachpack.

The interesting thing about these two definitions is that circles and rectangles have a lot in common. For example, both can be moved, enlarged or drawn. Many algorithms make sense for any geometric shape and only differ in the details. Let's call the abstraction that denotes popular geometric figures `Shape`, for example. Let's assume we already have a function that can determine whether any geometric shapes overlap:

```
; Shape Shape -> Boolean
; determines whether shape1 overlaps with shape2
(define (overlaps shape1 shape2) ...)
```

Based on these and similar functions, we can program further functions that are completely independent of the specific shape (whether circle or rectangle), for example a function that tests for three shapes whether they overlap in pairs:

```

; Shape Shape Shape -> Boolean
determines whether the shapes overlap pairwise
(define (overlaps/3 shape1 shape2 shape3)
  (and
    (overlaps shape1 shape2)
    (overlaps shape1 shape3)
    (overlaps shape2
      shape3)))

```

The `overlaps` function must distinguish what exact form its parameters have, but the `overlaps/3` function no longer does. This is a very powerful form of abstraction and code reuse: We formulate abstract algorithms based on a set of simple functions like `overlaps` and can then apply these algorithms to *all* kinds of shapes.

So the `overlaps/3` function really stands for a whole family of algorithms: Depending on what shapes I am currently using as an argument, the auxiliary functions such as `overlaps` do something different, but the abstract algorithm in `overlaps/3` is always the same.

Let us now look at how we can define data types such as `Shape` and functions such as `overlaps`. In order to reduce the number of possible geometric figures we use a sum type whose alternatives are the product types from above:

```

; A Shape is either:
; - a GCircle
; - a GRectangle
; interp. a geometrical shape representing a circle or a rectangle

```

Not all algorithms are like `overlaps/3` - they are different depending on which shape is currently available. As already known from the design recipe for sum types, we structure such functions by making a case distinction.

Example:

```

; Shape Posn -> Boolean
Determines whether a point is inside a shape
(define (point-inside shape point)
  (cond [(gcircle? shape) (point-inside-circle shape point)]
        [(grectangle? shape) (point-inside-rectangle shape point)]))

```

In this function, it is noticeable that auxiliary functions are called in the branches of the `cond` expression, which we still have to implement. You could write the implementation of these functions directly in the `cond` expression instead, but if the alternatives of a sum type are products, these expressions often become so complex that it is better to outsource them to separate functions. In addition, outsourcing them to separate functions often results in opportunities to reuse these functions.

In our example, the implementations of these auxiliary functions look like this off:

Complete the
Definitions for
vector-length
and posn-!

```
GCircle Posn -> Boolean
; Determines whether a point is inside a circle
(define (point-inside-circle circle point)
  (<= (vector-length (posn- (gcircle-center circle)
                             point)) (gcircle-radius circle)))

GRectangle Posn -> Boolean
; Determines whether a point is inside a rectangle
(define (point-inside-rectangle rectangle point)
  (and
    (<= (posn-x (grectangle-corner-ul rectangle)) (posn-x point)
        (posn-x (grectangle-corner-dr rectangle)))
    (<= (posn-y (grectangle-corner-ul rectangle)) (posn-y point)
        (posn-y (grectangle-corner-dr rectangle)))))
```

In some functions, several parameters have a sum type. In this case, you can first make the case distinction for the first parameter, then nested for the second parameter and so on. However, it is often possible to combine the case distinctions. This is the case, for example, with the `overlaps` mentioned above:

```
; Shape Shape -> Boolean
; determines whether shape1 overlaps with
shape2 (define (overlaps shape1 shape2)
  (cond [(and (gcircle? shape1) (gcircle?
                                shape2)) (overlaps-circle-circle shape1
                                                                    shape2)]
        [(and (grectangle? shape1) (grectangle?
                                      shape2)) (overlaps-rectangle-rectangle shape1
                                                                                shape2)]
        [(and (grectangle? shape1)
              (grectangle? shape2)) (overlaps-rectangle-
                                     rectangle shape1 shape2)]
        [(and (grectangle? shape1) (gcircle? shape2)) (overlaps-rectangle-
                                                           circle shape1 shape2)]
        [(and (gcircle? shape1)
              (grectangle? shape2)) (overlaps-rectangle-
                                     circle shape2 shape1)])))
```

Once again, we have outsourced the implementation of the individual cases to functions. The fact that we call the same function in the last two cases illustrates that these auxiliary functions facilitate the reuse of code. promote. Here is the implementation of the auxiliary functions

if you enjoy
geometry,
(def
ine

```
GCircle GCircle -> Boolean
; determines whether c1 overlaps with c2
```

```
(overlaps-circle-circle c1 c2)  
; Two circles overlap if and only if the distance of  
their
```

supplement the implementation
of
`overlaps-rectangle-rectangle`
and
`overlaps-rectangle-circle`.


```

; centers is smaller than the sum of their radii
(<= (vector-length (posn- (gcircle-center c1) (gcircle-
center c2))))
(+ (gcircle-radius c1) (gcircle-radius c2)))

GRectangle GRectangle -> Boolean
; determines whether r1 overlaps with r2
(define (overlaps-rectangle-rectangle r1 r2) ...)

GRectangle GCircle -> Boolean
; determines whether r overlaps with c
(define (overlaps-rectangle-circle r c) ...)

```

7.2 Program design with ADTs

We supplement the design recipe from section §3.3.3 "Design recipe for function definition" as follows:

1. If you want to program a function that receives information as input or produces information as output that is best represented by ADTs, you should define this ADT before programming this function (if you have not already defined it in the context of another function).

An ADT is useful if different types of information are differentiated in your problem definition, each of which can be formulated as a product type, but which represent a common concept.

An important point is that data can be organized hierarchically with ADTs. The field types of the products that you define can therefore have an ADT themselves. So if a product type has a large number of fields, or a sum type has a large number of alternatives, this indicates that you should use a deeper hierarchy (by nesting ADTs).

2. In the second step, the definition of the function signature and the task description, nothing changes - however, you can and should of course now use the names of the defined ADTs.
3. When defining the test cases, you should define at least one test per alternative for summary types. Please note that there may now be many more alternatives due to the nesting of sum and product types. With very large data types, it may no longer be realistic to test every alternative because, in the worst case, the number of alternatives grows exponentially with the depth of the data type.
4. There are now two dimensions in the definition of the function template: The sum type and the product types in (some of its) alternatives.

If we have a sum type as the outermost type, we should first enter it in a `cond` expression distinguish all cases.

For all alternatives that are product types, an auxiliary function should generally be defined to cover this case. You should only implement this auxiliary function in this step (by reapplying this design recipe) up to the template step.

Only if the implementation of this case is likely to be very simple should the selectors for the product fields be included in the template.

However, there is one important case in which you should *not* include a `cond` expression to differentiate the alternatives in the template, namely when it is possible to formulate the function abstractly - i.e. you do not differentiate the cases, but merely call existing functions that have also been implemented on the basis of the ADT. As an example, we have seen the [overlap/3](#) function.

5. In this step, you should turn the template into an executable program. As you may have defined templates for auxiliary functions in the previous step, you must now also implement these auxiliary functions.
6. Test them. If tests fail, go back to the previous step.
7. When refactoring, also check the newly defined ADTs. Here, too, there may be violations of the DRY principle, e.g. if there are large commonalities between ADTs. For example, if there are two fields to represent coordinates in several data types, it is advisable to use the [posn](#) structure instead. Avoid very wide but flat ADTs; the logical grouping of data through a hierarchy of ADTs promotes the reusability and readability of the code.

8 Meaning of BSL

In this chapter we will summarize and formally define the meaning of (almost) all BSL language constructs.

This is done in two steps: First, we will define the *syntax* of the language. The syntax defines which texts are BSL programs. The syntax is defined in the form of a *grammar*. The grammar not only says which texts are BSL programs, but also breaks down a BSL program into its parts, just as a grammar for natural language breaks down a sentence into parts such as subject, predicate and object.

In the second step, we define what the grammatically correct BSL programs mean. We determine the meaning by defining reduction steps with which BSL programs can be evaluated to values (provided no error occurs and they terminate).

We have already defined these reduction steps for most language constructs in sections §1.5 "Meaning of BSL expressions", §2.3 "Meaning of function definitions", §2.4.2 "Meaning of conditional expressions" and §2.7 "Meaning of function and constant definitions". We will specify these reduction steps again here using the formal syntax definition. In addition, we will now also define the meaning of structures.

There are different ways to define the meaning of a programming language; the one we use is called *reduction semantics* or *structural operational semantics* or *Plotkin semantics* (after Gordon Plotkin). For the formalization of the evaluation positions we have talked about in the previous chapters, we use so-called *evaluation contexts*, which were proposed by Matthias Felleisen and Robert Hieb in 1989. All this may sound scary for a beginner programmer, but you will see that it is not as complicated as it sounds :-)

8.1 Why?

Most programmers in this world program without having seen a formal definition of the meaning of their programming language. In this respect, the question of why we "do this to ourselves" is justified.

It should be noted that many programmers do not really understand the programming language they are using. This leads to a methodology in which, instead of systematic program design, the program is simply "tinkered with" until it "runs". Validating a program by executing and testing it makes sense, but this cannot replace the mental process of how a program must run so that the correct output is produced for each input. To do this, it is essential that you understand exactly what the code you have just programmed means.

In principle, we want you to be able to run your programs on a piece of paper and predict exactly what your code will do. Even though you may find the more theoretical concepts in this chapter most

Although it is difficult to get started, we believe that this chapter can help you become a better and more effective programmer.

That being said, you will see that the theoretical concepts you learn about in this chapter have an elegance that makes them worth studying for that reason alone.

8.2 Context-free grammars

So far, we have only described informally what BSL programs look like. With the help of a *grammar*, this informal description can be presented precisely and concisely. There are many different types of grammars. In the field of programming languages, so-called *context-free* grammars are mostly used. These and other grammar formalisms are discussed in detail in the lecture "Theoretical Formatting"; we will only discuss grammars here to the extent necessary to understand the definitions.

There are different notations for context-free grammars. We use the so-called EBNF - the Extended Backus Naur Form.

Here is an example of a grammar for numbers:

```
xCountly      ::= xPositiveZahly
                | xPositiveCountly
xPositiveCountly ::=
xWholeNumber  | xCommaNumber
xWholeNumber  ::= xDigitNotNully xDigit*
                | 0
xCommaCountly ::= xIntegerCountly . xDigits+
xDigitNotNully ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
xDigits       ::= 0 | xDigitNotNully
```

Examples of texts that correspond to the xZahly definition of this grammar are:

0, 420, -87, 3.1416, -2.09900.

Examples of texts that do not correspond to the xZahly definition of this grammar are: 007, -.65, 13., twelve, 111Nonsense222.

The identifiers marked with angle brackets such as *xZahly* are called *non-terminals*; the symbols marked in color such as 3 or . are called *terminal symbols*. A clause like the first two lines of the above grammar is called a *production*. A production consists of a non-terminal on the left-hand side of the definition and a set of alternatives on the right-hand side, which are separated from each other by the symbol |. There is exactly one production for each non-terminal.

A set of derivation *trees* can be formed for each non-terminal. A derivation tree is created by replacing the non-terminals in one of the alternatives of the corresponding production with derivation trees for these non-terminals. The construction of derivation trees is therefore a recursive process. The process stops where an alternative is chosen that consists only of terminal symbols. If a non-terminal is marked with an asterisk or a plus sign, such as *xZiffery** or *xZiffery+* above, this means 0 or more repetitions (for *) or 1 or more repetitions (for +) of the non-terminal.

Each derivation tree represents a text (often called a *word* or *sentence*), namely the sequence of terminal symbols that occur in the tree, read from left to right in the tree. The language defined by a grammar is the set of all words for which derivation trees can be formed.

Here are some examples of derivation trees² of the non-terminal *xZahly* and the words they represent. For the sake of simplicity, we represent the trees by indenting the text. Since the trees are rotated by 90 degrees compared to the standard representation, the terminal symbols must be read from top to bottom (instead of left to right).

The derivation tree for *0* is:

```
xZahly
xPositiveZahly
  xWholeNumber
    0
```

The derivation tree for *420* is:

```
xZahly
xPositiveZahly
  xWholeNumber
    xNumberNotNully
      4
    xZiffery
      xNumberNotNully
        2
      xZiffery
        0
```

8.3 Syntax of BSL

After this preliminary work, we can now precisely define the syntax of BSL using a text-free grammar. This syntax is complete except for the following language features, which we have not yet covered: Definition of functions by lambda expressions, quoting, character data, library imports. In addition, the grammar ignores aspects that are irrelevant to the meaning of the language, such as comments, line breaks and spaces. For this reason, grammars such as the following for BSL are often referred to as the abstract *syntax* of a programming language, in contrast to the *concrete syntax*, which also includes aspects such as comments and line breaks. Similarly, derivation trees, as described above, are used in the context of abstract syntax.

² If you would like to experiment with grammars and derivation trees, take a look at the grammar tool at <http://www.cburch.com/proj/grammar/index.html>. With this tool, you can automatically enter derivation trees for words of context-free grammars. However, the notation for context-free grammars in the tool is slightly different and it does not support the + and * operators and only alphanumeric non-terminals. A variant of the grammar above, which this tool understands, can be found at the URL <https://github.com/ps-mr/KdP2014/blob/master/materials/grammar>.

often referred to as *abstract syntax trees (AST)*.

```

xprogramy      ::= xdef-or-expry*
xdef-or-expry  ::= xdefinitiony | xkey
xdefinitiony   ::= ( (define ( xnamey xnamey+ ) xkey )
                  | (define xnamey xkey )
                  | (define-struct xnamey ( xnamey* ) )
xkey           ::= ( xnamey xkey* )
                  | (cond { [ xkey xkey ] }+ )
                  | (cond { [ xkey xkey ] }* [ else xkey ] )
                  | (if xkey xkey xkey )
                  | (and xkey xkey+ )
                  | (or xkey xkey+ )
                  | xnamey
                  | xvy
xvy            ::= < make-xnamey xvy* >
                  | xnumbery
                  | xbooleany
                  | xstringy
                  | ximagey

```

The non-terminal *xprogramy* stands for the syntax of entire programs; *xdef-or-expry* for definitions or expressions, *xdefinitiony* for function/constant definitions, */structure definitions*, *xkey* for expressions and *xvy* for values.

The curly braces around subsequences as in `{ [xkey xkey] }+` are used to apply the * or ⁺ operator to a whole sequence of terminal symbols and non-terminals and not just to a single non-terminal. In this example, it means that 1 or more occurrences of `[xkey xkey]` are expected.

The productions for some non-terminals, whose exact form is not interesting, have been omitted from the grammar: *xnamey* stands for the permitted designators for functions, structures and constants. *xnumbery* stands for the permitted numbers. *xbooleany* stands for `#true` or `#false`. *xstringy* stands for all strings such as `"asdf"`. The non-terminal *ximagey* stands for images in the program text (image



literals) such as `image`.

The values of the form `< make-xnamey xvy* >` are used to create instances of structures to represent the program text. They may not appear directly in the original program text in BSL, but they are generated during the reduction and inserted into the program.

8.4 The BSL core language

When defining the meaning of a language, you usually want this definition to be as short as possible, because only then can a user easily understand it and draw conclusions.

For this reason, we identify a sublanguage of BSL that is already sufficient to write all programs that can also be written in BSL. The

The only difference is that in some places you may have to write something more complicated.

We have already become acquainted with an intellectual tool for distinguishing core language elements from rather unimportant accessories, namely syntactic sugar. In section §2.4.2 "Meaning of conditional expressions" we have seen that it is not necessary to support `if` expressions and the `else` operator within `cond` expressions, because these language features can easily be simulated with the simple `cond` expression. We therefore consider the transformations given in §2.4.2 "Meaning of conditional expressions" to be the *definition of* these language features and will therefore only consider the core expressions in the following.

language into which these transformations are mapped.

The syntax above also contains special syntax for the logical functions `and` and `or`, because their arguments are evaluated differently from the arguments of normal functions. However, in our core language it is not necessary to consider both functions, since `or` can be expressed using `and` and `not`. We therefore "decode" `(or e-1 ... e-n)` to `(not (and (not e-1) ... (not e-n)))`.

You could also try to transform `and` away and replace it with a `cond` expression: `(and e-1 e-2)` is transformed to `(cond [e-1 e-2] [else #false])`. Although this correctly simulates the evaluation order, this transformation is not adequate for the behavior implemented in DrRacket, as the following example illustrates:

```
> (and #true 42)
and: question result is not true or false: 42

> (cond [#true 42] [else
#false]) 42
```

The grammar of our core language thus looks as follows. The grammar for values `xvy` remains unchanged.

```
xprogramy ::= xdef-or-expry*
xdef-or-expry ::= xdefinitiony | xkey
xdefinitiony ::= ( define ( xnamey xnamey+ ) xkey ) █
                | ( define xnamey xkey ) █
                | ( define-struct xnamey ( xnamey* ) ) █
xkey ::= ( xnamey xkey* )
        | ( cond { [ xkey xkey ] }+ ) █
        | ( and xkey xkey+ ) █
        | xnamey
        | xvy
```

Do your research, what *de Morgan's rules* are, if you are not aware of the transformation.

8.5 Values and environments

What do programs mean in the language whose syntax was defined above? We want to model the meaning of an expression as a sequence of reduction steps that leads to a value at the end (or terminates with an error or does not terminate).

We have already defined values above using the grammar. All constants such as `#true`, `42` or `"xyz"` are therefore values. In addition, instances of structures are values; the values of all fields in the structure must also be values. So, for example, `<make-posn 3 4>` is a value. We model structures in such a way that expressions such as `(make-posn 3 (+ 2 2))` are evaluated to this value - so here the expression that calls `make-posn` (with round brackets) is of the value `<make-posn 3 4>` (with angle brackets).

If functions, constants or structures are used in expressions, the evaluation of an expression cannot take place in a "vacuum", but you must know the *environment* (*environment*, abbreviated as *env* in the following) in which the expression is evaluated in order to have access to the associated definitions. In principle, the environment is simply the part of the program up to the expression that is currently being evaluated. However, constant definitions are also evaluated (see §2.7 "Meaning of function and constant definitions"). This is expressed by the following definition. Note that, in contrast to the BSL grammar, constant definitions have the form

```
( define xnamey xvy ) and not ( define xnamey key )
xenvy          ::= xenv-elementy*
xenv-elementy ::= ( ( define ( xnamey xnamey+ ) xkey )
                  | ( ( define xnamey xvy )
                  | ( ( define-struct xnamey ( xnamey* ) ) )
```

An environment therefore consists of a sequence of function, constant or structure definitions, whereby the expression in constant definitions has already been evaluated to a value.

8.6 Evaluation positions and the congruence rule

In section §1.5 "Meaning of BSL expressions", we talked about evaluation positions and the congruence rule for the first time. The evaluation positions mark which part of a program is to be evaluated next. The congruence rule says that you can evaluate sub-expressions in evaluation positions and incorporate the result of the evaluation back into the whole expression.

Let's look at the expression `(* (+ 1 2) (+ 3 4))` as an example. The sub-expression `(+ 1 2)` is in the evaluation position and can be evaluated to `3`. According to the congruence rule, I can therefore reduce the total expression to `(* 3 (+ 3 4))`.

We will formalize evaluation positions and the congruence rule using an evaluation *context*. An *evaluation context* is a grammar for programs that contain a "hole", `[]`. In relation to the DrRacket stepper, the evaluation context can be understood as the part of the expression that is not color-coded during a reduction. The grammar is structured in such a way that each element of the defined language contains exactly one "hole".

```
xEy ::= []
      | ( xnamey xvy* xEy xkey* )
      | ( cond [ xEy xkey ] { [ xkey xkey ] }* )
```



```
| (and xEyxy+) |
| (and #true xEy) |
```

Here are some examples of evaluation contexts:

```
( * [] (+ 3 4) )
```

```
(posn-x (make-posn 14 []))
```

```
(and #true (and [] x))
```

None of these are evaluation contexts:

```
( * (+ 3 4) [] )
```

```
(posn-x (make-posn 14 17))
```

```
(and #true (and x []))
```

The "hole" in these expressions represents exactly the sub-expressions in a program that are in evaluation position. We use the definition for values, xvy , from above to control that the evaluation of arguments in function calls is done from left to right.

Previously (in section §1.5 "Meaning of BSL expressions" and §2.3 "Meaning of function definitions"), we had defined the evaluation positions in such a way that the arguments can be evaluated in any order when calling functions. The evaluation contexts as defined above determine this order, namely from left to right. We will say more about this difference later.

Let E be an evaluation context. We use the notation $E[e]$ to replace the "hole" in the evaluation context with an expression e .

Example: If $E = (* 3 [] (+ 3 4))$, then $E[(+ 1 2)] = (* (+ 1 2) (+ 3 4))$.

Using this notation, we can now define the congruence rule as follows:

(KONG): If $e-1 \rightarrow e-2$, then $E[e-1] \rightarrow E[e-2]$.

We generally write the evaluation rules in such a way that we give each rule a name. This rule is called *(KONG)*.

Example: Consider the expression $e = (* (+ 1 2) (+ 3 4))$. This can be broken down into an evaluation context $E = (* [] (+ 3 4))$ and an evaluation context pressure $e-1 = (+ 1 2)$, so that $e = E[e-1]$. Since we can reduce $e-1$, $(+ 1 2) \rightarrow 3$, we can also reduce e to $E[3] = (* 3 (+ 3 4))$ thanks to *(KONG)*.

8.7 Non-terminals and metavariables - Don't panic!

In the congruence rule above, names such as $e-1$ and $e-2$ stand for any expression and E for any evaluation context.

In general, we use the convention that the name x and variants such as $x-1$ and $x-2$ stand for any words of the non-terminal xy (for example, for $xy = xey$ or $xy = xvy$). Such identifiers as $v-1$ or $e-2$ are also called *meta variables*, because they are not variables of BSL, but variables that stand for parts of BSL programs.

If we interpret nonterminals as sets (namely the set of words for which there are derivation trees), we could also write down the rule from above in this way:

For all $e-1Pxey$ and all $e-2Pxey$ and all $EPxEy$: If $e-1 e-2$, then $E[e-1]E[e-2]$.

However, as this makes the rules much longer, we use the convention described here.

8.8 Importance of programs

According to our grammar, a program consists of a sequence of definitions and expressions. We call the evaluation rule for programs (*PROG*):

(PROG): A program is executed from left to right and starts with the empty environment. If the next program element is a function or structure definition, this definition is included in the environment and execution is continued with the next program element in the extended environment. If the next program element is an expression, it is evaluated to a value in the current environment according to the rules below. If the next program element is a constant definition (**define** x e), it is evaluated to a value in the current environment. e is first evaluated to a value v and then (**define** x v) is added to the current Environment added.

Example: The program is:

```
(define (f x) (+ x 1))
(define c (f 5))
(+ c 3)
```

In the first step, (**define** $(f\ x)\ (+\ x\ 1)$) is added to the (empty) environment. The constant definition is evaluated in the environment (**define** $(f\ x)\ (+\ x\ 1)$) to (**define** $c\ 6$) and then added to the context. Finally, the expression $(+ c\ 3)$ is added to the environment

```
(define (f x) (+ x 1))
(define c 6)
```

evaluated.

8.9 Meaning of expressions

Each expression is evaluated in an environment *env* as defined in the previous section. In order not to overload the notation, we will not add *env* explicitly to each reduction rule but assume it to be implicitly given.

The evaluation is defined, as known from section §1.5 "Meaning of BSL expressions", in the form of reduction rules of the form $e \rightarrow e'$. An expression e is evaluated by reducing it until a value is obtained: $e \rightarrow v$.

An error during the evaluation manifests itself in the fact that the reduction "gets stuck", i.e. we arrive at an expression that is not a value and that cannot be reduced any further.

8.9.1 Meaning of function calls

Functions are executed differently depending on whether the function name is a primitive function or a function defined in the environment. In the first case, the primitive function is evaluated on the arguments. If this is successful, the expression can be reduced to the result. If this is not successful, the expression cannot be reduced.

If, however, the function is defined in the environment, the call is reduced to the body of the function definition, whereby all parameter names are replaced by the current parameter values beforehand. This is the meaning of the notation $e[name \rightarrow v]$.

The reduction rules are therefore:

(FUN): If (`define (name name-1 ... name-n) e`) in the environment,
 then (`name v-1 ... v-n`) $\rightarrow e[name \rightarrow v]$
 (PRIM): If `name` is a primitive function f and $f(v-1, \dots, v-n) = v$,
 then (`name v-1 ... v-n`) $\rightarrow v$.

8.9.2 Meaning of constants

Constants are evaluated by looking them up in the environment:

(CONST): If (`define name v`) in the environment, then `name` $\rightarrow v$.

8.9.3 Meaning of conditional expressions

Conditional expressions are evaluated as already described in §2.4.2 "Meaning of conditional expressions". According to the definition of the evaluation context, only the first conditional expression is evaluated. Depending on whether this results in `#true` or `#false`, it is reduced to the result expression or the `cond` expression shortened by the failed condition:

(COND-True): (`cond [#true e]`) $\rightarrow e$
 (COND-False): (`cond [#false e-1] [e-2 e-3] ...)` (`cond [e-2 e-3]`)

8.9.4 Meaning of boolean expressions

The definition of the evaluation of Boolean expressions only evaluates the conditions as far as necessary. In particular, the evaluation is aborted as soon as one of the expressions returns `#false`.

The first two reduction rules are required to check that all arguments are boolean values; otherwise, the two rules could have been combined into `(and #true v) v`. In total, we need the following four rules for Boolean expressions:

```
(AND-1): ( and #true #true ) #true
(AND-2): ( and #true #false ) #false
(AND-3): ( and #false ... ) #false
(AND-4): ( and #true e-1 e-2 ... ) ( and e-1 e-2 ... )
```

8.9.5 Meaning of structure constructors and selectors

Structure definitions define three types of functions: Constructors like `make-posn`, selectors like `posn-x`, and predicates like `posn?`. We need a reduction rule for each of these three types.

Constructors create instances of a structure. This is possible if a structure with the same name can be found in the environment and this has as many fields as the constructor has parameters. This brings us to the following rule:

(STRUCT-make): If `(define-struct name (name-1 ... name-n))` in the environment, then `(make-name v-1 ... v-n) < make-name v-1 ... v-n >`.

Selector calls are reduced by looking up the structure definition in the environment to map the name of the field to the argument position of the constructor call. The corresponding value of the field is then returned:

(STRUCT-select): If `(define-struct name (name-1 ... name-n))` in the environment, then `(name-name-i < make-name v-1 ... v-n >) v-i`.

For predicates, the system checks whether the argument of the predicate is a structure instance of the structure in question or not, and returns `#true` or `#false` depending on this:

```
(STRUCT-predtrue): ( name? < make-name ... > ) #true
(STRUCT-predfalse): If v is not < make-name ... >, then ( name? v )
#false
```

8.10 Reduction using the example of

Consider the following program, the meaning of which we will determine step by step with the help of the evaluation rules:

```
(define-struct s (x y))
(define (f x) (cond [(< x 1) (/ x 0)]
                   [#true (+ x 1)]
                   [#true x]))
(define c (make-s 5 (+ (* 2 3)
4))) (f (s-x c))
```

- According to (PROG), we start with the empty environment $env = \text{empty}$. The first program element is a structure definition, therefore according to (PROG) the environment in the next step is $env = (\text{define-struct } s \ (x \ y))$.

- The next program element is a function definition, so according to (*PROG*) the environment in the next step *env* =

```
(define-struct s (x y))
(define (f x) (cond [(< x 1) (/ x 0)]
                  [#true (+ x 1)]
                  [#true x]))
```

- The next program element is a constant definition. According to (*PROG*) we must first evaluate (*make-s* 5 (+ (* 2 3) 4)):

- *e* = (*make-s* 5 (+ (* 2 3) 4)) decomposes into *E* = (*make-s* 5 (+ *e*-1 6 4)) and *e*-1 = (* 2 3). According to (*PRIM*) *e*-1 6 applies; according to (*KONG*) *e* (*make-s* 5 (+ 6 4)) therefore applies.
- *e* = (*make-s* 5 (+ 6 4)) decomposes into *E* = (*make-s* 5 []) and *e*-1 = (+ 6 4). According to (*PRIM*) *e*-1 10 applies; according to (*KONG*) *e* (*make-s* 5 10) therefore applies.
- (*make-s* 5 10) <*make-s* 5 10> according to (*STRUCT-*

make). According to (*PROG*), our new environment is therefore

now *env* =

```
(define-struct s (x y))
(define (f x) (cond [(< x 1) (/ x 0)]
                  [#true (+ x 1)]
                  [#true
x]))) (define c <make-s 5 10>)
```

- The last program element is an expression that we evaluate according to (*PROG*) in the current environment:
 - *e* = (f (s-x c)) decomposes into *E* = (f (s-x [])) and *e*-1 = c. According to (*CONST*), c <*make-s* 5 10> applies; according to (*KONG*), *e* (f (s-x <*make-s* 5 10>)) therefore applies.
 - *e* = (f (s-x <*make-s* 5 10>)) decomposes into *E* = (f []) and *e*-1 = (s-x <*make-s* 5 10>). According to (*STRUCT-select*) *e*-1 5 applies; according to (*KONG*) therefore *e* (f 5) applies.
 - (f 5) (cond [(< 5 1) (/ 5 0)] [#true (+ 5 1)] [#true 5]) according to (*FUN*).
 - *e* = (cond [(< 5 1) (/ 5 0)] [#true (+ 5 1)] [#true 5]) zero-falls in *E* = (cond [[] (/ 5 0)] [#true (+ 5 1)] [#true 5]) and *e*-1 = (< 5 1). According to (*PRIM*), *e*-1 is #false; according to (*KONG*), *e* is therefore (cond [#false (/ 5 0)] [#true (+ 5 1)] [#true 5]).
 - (cond [#false (/ 5 0)] [#true (+ 5 1)] [#true 5]) (cond [#true (+ 5 1)] [#true 5]) according to (*COND-False*).
 - (cond [#true (+ 5 1)] [#true 5]) (+ 5 1) according to (*COND- True*).

– $(+ 5 1) 6$ according to *(PRIM)*.

8.11 Meaning of data and data definitions

Data definitions have no influence on program behaviour, as they are defined in the form of a comment. Nevertheless, we can give them a precise meaning that helps to understand their role.

To do this, it is important to understand the *data universe* of a program. The data universe includes all data that can potentially occur in a given program. Which values these are is described by our grammar for values, *xvy*, above. However, not all values described by *xvy* can occur in a program, but only those for which the structures used are actually defined in the program.

Example: A program contains the structure definitions

```
(define-struct circle (center radius))
(define-struct rectangle (corner-ul corner-dr))
```

The data universe for this program includes all values of the basic types, *but* also all structure instances that can be created on the basis of these structure definitions, for example `<make-circle 5 6>`:

```
<make-circle <make-circle <make-rectangle 5 <make-
rectangle #true "asdf">> 77> 88>
```

The data universe is therefore all values that can be determined by the grammar of *xvy* limited to the structures defined in the program.

A structure definition therefore adds new values to the data universe, namely all values in which this structure is used at least once.

A data definition, on the other hand, does not expand the data universe. A data definition defines a *subset* of the data universe.

Example:

```
; a Posn is a structure: (make-posn Number Number)

<make-posn 3 4 > is an element of the defined subset, but <make-posn
#true "x" > or <make-posn <make-posn 3 4> 5> are not.
```

A data definition generally describes a coherent subset of the data universe. Functions can use their signature to make clear which values of the data universe they accept as arguments and which results they produce.

8.12 Refactoring of expressions and closing by equations

In section §1.5 "Meaning of BSL expressions", we introduced how to define an equivalence relation on expressions based on the reduction rules

can. These equivalences can be used to refactor programs - i.e. program changes that do not change the meaning but improve the structure of the program. They can also be used to derive properties of your program, for example that the `overlaps-circle` function from the previous chapter is commutative, i.e. `(overlaps-circle c1 c2) (overlaps-circle c2 c1)`.

However, the equivalence relation from Section §1.5 "Meaning of BSL expressions" was too small for many practical purposes, because it requires, for example, that we can only resolve function calls if all arguments are values.

However, BSL has a remarkable property that allows us to define a much more powerful equivalence relation: It does not matter for the result of a program in which order expressions are evaluated. In particular, it is not necessary to evaluate the arguments before a function call; you can simply use the argument expressions.

The idea is expressed in the following, more general evaluation context:

```
xEy ::= []
      | ( xnamey key* xEy key* )
      | ( cond { [ key key ] }* [ xEy key ] { [ key key ] }* )
      | ( cond { [ key key ] }* [ key xEy ] { [ key key ] }* )
      | ( and key* xEy key* )
```

Together with the following congruence rule for our equivalence relation, this evaluation context expresses that "like may be replaced with like" everywhere:

(EKONG): If $e-1 \sim e-2$, then $E[e-1] \sim E[e-2]$.

An equivalence relation should be as large as possible so that we can show as many equivalences as possible. At the same time, it should be correct. This means that equivalent programs have the same behavior, i.e. in particular - if they terminate - result in the same value when evaluated.

We now gradually define the rules that should apply to the equivalence relation. First of all, it should actually be an equivalence relation - i.e. reflexive, commutative and transitive:

(EREFL): $e \sim e$.

(EKOMM): If $e1 \sim e2$, then $e2 \sim e1$.

(ETRANS): If $e-1 \sim e-2$ and $e-2 \sim e-3$, then $e-1 \sim e-3$.

The link to the evaluation relation is created by this rule: Reduction receives equivalence.

(ERED): If $e-1 \sim e-2$ then $e-1 \sim e-2$.

So that we can also evaluate functions "symbolically", we extend the rule for function calls so that it is not necessary to evaluate the arguments to determine equivalences.

(EFUN): If (define (name name-1 ... name-n) e) in the environment, then (name e-1 ... e-n) e [name-1 := e-1 ... name-n := e-n]

With the conjunction, we know that the overall expression evaluates to `#false` (or does not terminate) if at least one of the arguments is equivalent to `#false`.

(EAND): (and ... #false ...) #false

We can also use the knowledge we have about the built-in functions when reasoning with equivalences. For example, we know that $(+ a b) \equiv (+ b a)$. We summarize the set of equivalences that apply to the built-in functions under the name *(EPRIM)*.

However, there is still a small catch. One would like an equivalence relation for programs to have the following property: If $e-1 \equiv e-2$ and $e-1 \rightarrow v$, then also $e-2 \rightarrow v$. However, this property does not apply because there can be that $e-1$ terminates but $e-2$ does not.

Example: Consider the following program:

```
(define (f x) (f
x)) (define (g x)
42) (g (f 1))
```

Since $(f 1) \equiv (f 1)$, the calculation of the argument for g does not terminate, and according to the congruence rule $(g (f 1)) \equiv (g (f 1))$ applies, therefore the calculation of the expression $(g (f 1)) \equiv (g (f 1))$ does not terminate. On the other hand, according to *(EFUN)* $(g (f 1)) \equiv 42$ applies. When using the equivalence rules, it must therefore be taken into account that the equivalence only applies on the condition that the terms terminate on both sides.

However, the following somewhat weaker property applies, which we list without proof:

If $e-1 \equiv e-2$ and $e-1 \rightarrow v-1$ and $e-2 \rightarrow v-2$, then $v-1 = v-2$.

So if $e-1$ and $e-2$ are the same and both terminate, then the value that comes out is the same.

9 Data of any size

The data types we have seen so far basically represent data consisting of a fixed number of atomic data. This is because each data definition may only use other data definitions that have already been defined. For example, let's consider

```
(define-struct gcircle (center radius))  
; A GCircle is (make-gcircle Posn Number)  
; interp. the geometrical representation of a circle
```

we know that a `Posn` consists of two and a `Number` of one atomic data (both numbers) and thus a `GCircle` consists of exactly three atomic data.

In many situations, however, we do not know at the time of programming how many atomic data a composite datum consists of. In this chapter, we look at how we can represent data of any size (unknown at the time of programming) and program functions that process such data.

9.1 Recursive data types

As an example, let's look at a program that can be used to manage family trees of people. Each person in the family tree has a father and a mother; sometimes the father or mother of a person is unknown.

For example, we could use the existing means to represent a person's preferences:

```
(define-struct parent (name grandfather grandmother))  
A Parent is: (make-parent String String String)  
; interp. the name of a person with the names of his/her  
grandparents  
  
(define-struct person (name father mother))  
A Person is: (make-person String Parent Parent)  
; interp. the name of a person with his/her parents
```

However, we can only represent the ancestors up to exactly the grandparents. Of course, we could add further definitions for the great-grandparents etc., but we would always have a fixed size when programming. In addition, we violate the DRY principle, because the data definitions look very similar for each previous generation.

What can we do to deal with this problem? Let's take a closer look at what kind of data we actually want to model here. The insight we need here is that a family tree has a recursive structure: A person's family tree consists of the person's name and the family tree of his mother and the family tree of his parents. A family tree therefore has a property,

which is also called *self-similarity*: a part of a whole has the same structure as the whole.

This means that we have to drop the restriction that only the names of previously defined data types may appear in a data definition. In our example, we write

```
(define-struct person (name father mother))
```

A FamilyTree is: (make-person String FamilyTree FamilyTree)
; interp. the name of a person and the tree of his/her
parents.

The definition of FamilyTree therefore uses FamilyTree itself. First of all, it is not entirely clear what this means. Above all, it is also not clear how we are supposed to create a family tree. If we try to create one, we get a problem:

```
(make-person "Heinz" (make-person "Horst" (make-person "Joe"
...)))
```

We cannot generate a family tree at all, because we must already have a family tree to generate it. An expression that generates a family tree would therefore be infinitely large.

On closer consideration, however, family trees are never infinite, but end at some generation - for example, because the ancestors are unknown or not of interest.

We can take this fact into account by turning FamilyTree into a sum type as follows:

```
(define-struct person (name father mother))
```

A FamilyTree is either:
; - (make-person String FamilyTree FamilyTree)
; - #false
; interp. either the name of a person and the tree of its par-
ents,
; or #false if the person is not known/relevant.

This new, non-recursive base case now allows us to generate values of this type. Here is an example:

```
(define HEINZ
  (make-person "Heinz"
    (make-person "Elke" #false
      #false) (make-person "Horst"
        (make-person "Joe"
          #false
            (make-person "Rita"
              #false
```

```
#false)))
#false)))
```

So what does such a recursive data definition mean? Up to now, we have always interpreted data types as sets of values from the data universe, and this is still possible, only the construction of the set that the type represents is now somewhat more complex:

Let ft_0 be the empty set, ft_1 the set `t#falseu`, ft_2 the union of `t#falseu` and the set of `(make-person name false false)` for all strings `name`. In general, let ft_{i+1} be the union of `t#falseu` and the set of `(make-person name p1 p2)` for all strings `name` as well as for all `p1` and all `p2` from ft_i . For example, `HEINZ` is an element of ft_5 (and therefore also ft_6 , ft_7 etc.) but not of ft_4 .

Then the meaning of `FamilyTree`, ft , is the union of all these sets, so ft_0 united with ft_1 united with ft_2 united with ft_3 united with

In mathematical notation, we can summarize the construction as follows:

$$\begin{aligned} \text{ft}_0 &= \emptyset \\ \text{ft}_{i+1} &= \text{t\#falseu} \cup \{ \text{make-person } n \ p_1 \ p_2 \ q \mid n \in \text{String}, p_1 \in \text{ft}_i, p_2 \in \text{ft}_i, q \in \text{ft}_i \} \end{aligned}$$

It is not difficult to see that ft_i always contains ft_{i-1} ; the next quantity therefore always includes the previous one.

This set construction makes it clear why recursive data types make it possible to represent data of any size: Each set ft_i contains the values whose maximum depth in tree representation is i . Since we combine all ft_i with each other, the depth (and therefore also the size) is unlimited.

This set construction can be defined for any recursive data type. If there are several recursive alternatives, the i th set is the union of the i th sets for each alternative. For example, if there were an additional `FamilyTree` alternative `(make-celebrity String Number FamilyTree FamilyTree)` for which the property is specified in addition to the name, the i -th set would be

$$\begin{aligned} \text{ft}_{i+1} &= \text{t\#falseu} \cup \{ \text{make-person } n \ p_1 \ p_2 \ q \mid n \in \text{String}, p_1 \in \text{ft}_i, p_2 \in \text{ft}_i, q \in \text{ft}_i \} \\ &\cup \{ \text{make-celebrity } n \ w \ p_1 \ p_2 \ q \mid n \in \text{String}, w \in \text{Number}, p_1 \in \text{ft}_i, p_2 \in \text{ft}_i, q \in \text{ft}_i \} \end{aligned}$$

9.2 Programming with recursive data types

In the last section, we saw how to define recursive data types, what they mean and how to create instances of these data types. Now we want to consider how to program functions that have instances of such types as arguments or produce them as results.

Consider the following task: Program a function that finds out whether there is an ancestor with a certain name in a person's family tree.

A signature, task description and tests are quickly defined:

```
FamilyTree String -> Boolean
; determines whether person p has an ancestor a
(check-expect (person-has-ancestor HEINZ "Joe")
#true)
(check-expect (person-has-ancestor HEINZ "Emil") #false)
(define (person-has-ancestor p a) ...)
```

Since `FamilyTree` is a sum type, our design recipe says that we make a case distinction. In each branch of the case distinction, we can enter the selectors for the fields. In our example, this results in

```
(define (person-has-ancestor p a)
  (cond [(person? p)
        ... (person-name p) ...
        ... (person-father p) ...
        ... (person-mother p) ...]
        [else ...]))
```

The expressions `(person-father a)` and `(person-mother a)` stand for values that have a complex sum type, namely `FamilyTree` again. Obviously, a person has an ancestor `a` if the person is either `a` themselves or the mother or father has an ancestor with the name `a`.

Our design recipe suggests defining your own auxiliary functions for cases in which fields of a structure themselves have a complex sum/product type. We can indicate this as follows:

```
(define (person-has-ancestor p a)
  (cond [(person? p)
        ... (person-name p) ...
        ... (father-has-ancestor (person-father p) ...) ...
        ... (mother-has-ancestor (person-mother p) ...) ...]
        [else ...]))
```

We interpret the word "ancestor" to mean that a person is their own ancestor. What would the program have to look like in order to implement the non-reflective interpretation of the word "ancestor"?

However, if we take a closer look at what `father-has-ancestor` and `mother-has-ancestor` are supposed to do, we realize that they have the same signature and task description as `person-has-ancestor`! This means that the templates for these functions each require two new helper functions. Since we do not know how deep the family tree is, we cannot implement the program in this way.

Fortunately, however, we already have a function whose task is identical to that of `father-has-ancestor` and `mother-has-ancestor`, namely `person-has-ancestor` itself. This motivates the following template:

```
(define (person-has-ancestor p a)
  (cond [(person? p)
        ... (person-name p) ...
        ... (person-has-ancestor (person-father p) ...)...
        ... (person-has-ancestor (person-mother p) ...)...]
        [else ...]))
```

The structure of the data therefore dictates the structure of the functions. Where the data is recursive, the functions that process such data are also recursive.

Completing the template is now easy:

```
(define (person-has-ancestor p a)
  (cond [(person? p)
        (or
         (string=? (person-name p) a)
         (person-has-ancestor (person-father p) a)
         (person-has-ancestor (person-mother p)
                               a))]
        [else #false]))
```

The successful tests illustrate that this function appears to do what it is supposed to do, but why?

Let's compare the function with another approach that is not as successful. Let's look again at the beginning of the programming of the function, this time with a different name:

```
FamilyTree -> Boolean
; determines whether person p has an ancestor a
(check-expect (person-has-ancestor-stupid HEINZ "Joe") #true)
(check-expect (person-has-ancestor-stupid HEINZ "Emil")
               #false) (define (person-has-ancestor-stupid p a) ...)
```

If we already had a function that could determine whether a person had a certain ancestor, we could simply call this function. But fortunately, we are already in the process of programming this function. So let's just call this function:

```
FamilyTree -> Boolean
; determines whether person p has an ancestor a
(check-expect (person-has-ancestor-stupid HEINZ "Joe") #true)
(check-expect (person-has-ancestor-stupid HEINZ "Emil")
               #false) (define (person-has-ancestor-stupid p a)
  (person-has-ancestor-stupid p a))
```

Somehow this solution seems too simple. Running the tests confirms the suspicion. However, the tests do not fail, but the execution of the tests does not terminate and we have to cancel them by pressing the "Stop" button.

Why does `person-has-ancestor` work but not `person-has-ancestor-stupid`?

First of all, we can compare the programs operationally. If we consider the expression `(person-has-ancestor-stupid HEINZ "Joe")`, our reduction semantics says:

```
(person-has-ancestor-stupid HEINZ "Joe") ( person-has-
ancestor-stupid HEINZ "Joe")
```

There is therefore no progress in the evaluation. This explains why the execution does not stop.

This is different with `(person-has-ancestor HEINZ "Joe")`. The recursive calls always call `person-has-ancestor` on ancestors of the person. Since the family tree only has a finite depth, `(person-has-ancestor #false "Joe")` must be called at some point, and we end up in the second case of the conditional expression, which no longer contains any recursive expressions.

We can also look at the program from the point of view of the meaning of the recursive data type definition. For each input `p` there is a minimum `i` so that `p` is in ft_i . For `HEINZ`, this `i`=5. Since the values of the `mother` and `father` fields of `p` are thus in ft_{i-1} , it is clear that the `p` parameter is in ft_{i-1} for all recursive calls. Since the program obviously terminates for values from ft_0 (in the example the set `{#false}`), it is clear that all calls with values from ft_1 must also terminate, and thus also the calls with values from ft_2 and so on. We have thus shown that the function is well-defined for all values from ft_i for any `i` - in other words: for all values from `FamilyTree`.

This informally presented argument from the previous paragraph is mathematically. From a technical point of view, this is induction evidence.

With `person-has-ancestor-stupid` the situation is different, because in the recursive call the argument is not from ft_{i-1} .

The conclusion from these considerations is that recursion in functions is unproblematic and well-defined as long as it follows the structure of the data. Since values of recursive data types have an undefined but finite size, this so-called *structural recursion* is always well-defined. Our design recipe suggests that wherever data types are recursive, the functions that operate on them should also be (structurally) recursive.

Before we look at the customized design recipe in detail, let's first consider what functions that *produce* examples of recursive data types look like.

As an example, let's look at a function that is very useful for making your own family tree look a little more impressive, namely one that can be used to add a title to the name of all ancestors.

Here is the signature, task description and a test for this function:

```
; FamilyTree -> FamilyTree
; prefixes all members of a family tree p with title
t (check-expect
  (promote HEINZ "Dr.
") (make-person
```

```

"Dr. Heinz"
(make-person "Dr. Elke" #false
#false) (make-person
"Dr. Horst"
(make-person "Dr. Joe" #false
(make-person "Dr. Rita" #false #false)) #false)))

(define (promote p t) ...)

```

The function therefore consumes and produces a value of type `FamilyTree` at the same time. The template for such functions does not differ from that for `person-has-ancestor`. Here, too, we apply recursion where the data type is recursive:

```

(define (promote p t)
  (cond [(person? p)
        ... (person-name p) ...
        ... (promote (person-father p) ...) ...
        ... (promote (person-mother p) ...) ...]
        [else ...]))

```

Now it is quite easy to complete the function. To produce values of type `FamilyTree`, we build the results of the recursive calls into a call of the `make-person` constructor:

```

(define (promote p t)
  (cond [(person? p)
        (make-person
         (string-append t (person-name p))
         (promote (person-father p) t)
         (promote (person-mother p) t))]
        [else p]))

```

9.3 Lists

The `FamilyTree` data definition from above represents a set of trees in which each node has exactly two outgoing edges. Of course, we can also represent trees that have three or five outgoing edges in the same way - by having an alternative of the sum type in which the data type occurs three or five times respectively.

A particularly important special case is where each node has exactly one outgoing edge. These degenerate trees are also called *lists*.

9.3.1 Lists, homemade

Here is a possible definition for lists of numbers:


```
(define-struct lst (first rest))
```

A List-of-Numbers is either:
; - (make-lst Number List-Of-Numbers)
; - #false
; interp. the head and rest of a list, or the empty list

Here is an example of how we can create a list with the numbers 1 to 3:

```
(make-lst 1 (make-lst 2 (make-lst 3 #false)))
```

The design of functions on lists works in exactly the same way as the design of functions on all other trees. As an example, let's look at a function that adds up all the numbers in a list.

Here is the specification of this function:

```
List-Of-Numbers -> Number  
; adds up all numbers in a list  
(check-expect (sum (make-lst 1 (make-lst 2  
(make-lst 3 #false)))) 6)  
(define (sum l) ...)
```

For the template, the design recipe suggests differentiating between the various alternatives and incorporating recursive function calls in the recursive alternatives:

```
(define (sum l)  
  (cond [(lst? l) ... (lst-first l) ... (sum (lst-rest l)) ...]  
        [else ...]))
```

Completion is now simple on the basis of this template:

```
(define (sum l)  
  (cond [(lst? l) (+ (lst-first l) (sum (lst-rest l)))]  
        [else 0]))
```

9.3.2 Lists from the can

Because lists are such a common data type, there are predefined functions for lists in BSL. As `List-of-Numbers` shows, we don't actually need these predefined functions because we can simply represent lists as degenerate trees. Nevertheless, the built-in list functions make programming with lists a little more convenient and safer.

The built-in constructor function for lists in BSL is called `cons`; the empty list is not represented by `#false` but by the special constant `empty`.

It makes sense to represent the empty list with a new value that does not stand for anything else, because then there can never be any confusion. If

we wanted to do something analogous in our self-built data structure for lists, we could achieve this by defining a new structure without fields and defining its only value as a variable:

```
(define-struct empty-lst ())
(define EMPTYLIST (make-empty-lst))
```

Accordingly, the example list from above would now be constructed as follows:

```
(make-lst 1 (make-lst 2 (make-lst 3 EMPTYLIST)))
```

The `cons` operation corresponds to our `make-lst` from above, but with one important difference: It also checks that the second argument is also a list:

```
> (cons 1 2)
cons: second argument must be a list, but received 1 and 2
```

So you can imagine `cons` like this self-built variant of `cons` on the basis of `list-of-numbers`:

```
(define (our-cons x l)
  (if (or (empty-lst? l) (lst?
    l)) (make-lst x l)
      (error "second argument of our-cons must be a list")))
```

The most important built-in list functions are `empty`, `empty?`, `cons`, `first`, `rest` and `cons?`

They correspond to our self-made list type:

`EMPTYLIST`, `empty-lst?`, `our-cons`, `lst-first`, `lst-rest` and `lst?`

The function that corresponds to `make-lst` is hidden by BSL to ensure that all lists are always constructed with `cons` and that the invariant is enforced accordingly that the second field of the structure is really a list. We cannot recreate this "hiding" of functions with our previous means; we will come to this later when we talk about modules.

Our example from above looks like this when using the built-in list functions:

```
A List-of-Numbers is one of:
; - (cons Number List-Of-Numbers)
; - empty

List-Of-Numbers -> Number
; adds up all numbers in a list
(check-expect (sum (cons 1 (cons 2 (cons 3 empty)))) 6)
(define (sum l)
  (cond [(cons? l) (+ (first l) (sum (rest
    l)))] [else 0]))
```

9.3.3 The `list` function

Constructing lists with the help of `cons` and `empty` quickly turns out to be a bit tedious. For this reason, there is some syntactic sugar to create lists more conveniently: the `list` function.

With the `list` function we can create the list `(cons 1 (cons 2 (cons 3 empty)))` like this:

```
(list 1 2 3)
```

However, the `list` function is just a bit of syntactic sugar, defined as follows is:

```
(list exp-1 ... exp-n)
```

stands for `n` nested `cons` expressions:

```
(cons exp-1 (cons ... (cons exp-n empty)))
```

It is important to understand that this is really just a shorthand notation. Even if you can define lists more easily using `list`, it is important to always keep in mind that this *is* just a shorthand way of using `cons` and `empty`, because this is the only way the design recipe for recursive data types can be applied to lists.

9.3.4 Data type definitions for lists

We have seen a `List-of-Numbers` data definition above. Should we also write separate data definitions for `List-of-Strings` or `List-of-Booleans`? Should they use the same structure, or should we have separate structures for each of these data types?

It makes sense that all these data types use the same structure, namely in the form of the built-in list functions. There are two reasons for this: Firstly, all these structures would be very similar and we would be violating the DRY principle. Secondly, there is a whole range of functions that work on *arbitrary* lists, for example a `second` function that returns the second list element of a list:

```
(define (second l)
  (if (or (empty? l) (empty? (rest l)))
      (error "need at least two list elements")
      (first (rest l))))
```

This function (which, by the way, is already predefined, just like `third`, `fourth` and so on) works for any list, regardless of the type of data stored. We could not write such functions if we used different structures for different types of list elements.

The most common use case for lists is that the lists are *homogeneous*. This means that all list elements have a common type. This is not a

This is a very large restriction, because this common type can also be a sum type with many alternatives, for example. In this case, we use data definitions with *type parameters* like this:

```
; A (List-of X) is one of:  
; - (cons X (List-of X))  
; - empty
```

We use these data definitions by specifying a type for the type parameter. To do this, we use the syntax for function application; we therefore write `(List-of String)`, `(List-of Boolean)`, `(List-of (List-of String))` or `(List-of FamilyTree)` and implicitly mean the data definition listed above.

But what is a suitable data type for the signature of `second` above, i.e. in general for functions that work on arbitrary lists?

To make it clear that the functions work for lists with any element type, we say this explicitly in the signature. In the example of the `second` function, it looks like this:

```
[X] (List of X) -> X  
(define (second l) ...)
```

The `[X]` at the beginning of the signature says that this function has the following signature for each type `X`, i.e. `(List-of-X) -> X` for each possible replacement of `X` by a type. Variables like `X` are called *type variables*. For example, `second` has the type `(List-of Number) -> Number` or `(List-of (List-of String)) -> (List-of String)`. However, at the moment we will only program functions like `second` ourselves in exceptional cases. Most of the functions that we want to program at the moment process lists with a concrete element type. So-called *polymor-*

We will come back to functions such as `second` later.

9.3.5 But are lists really recursive data structures?

If you look at lists from the real world (on a sheet of paper, in a spreadsheet, etc.), they often do not suggest a recursive, nested structure but look "flat". So is it "natural" to define lists as we have done, using a recursive data type - as a degenerate tree?

In many (especially older) programming languages, there is more direct support for lists. Lists are built into these programming languages. Lists are not recursive in these languages; instead, the elements of the list are often accessed via an index. In order to support programming with lists, there is a whole set of programming language constructs (such as various loop and iteration constructs) that are dedicated to supporting these fixed list elements. built-in lists.

From a hardware point of view, such lists, which are accessed with an index, are very natural, because they correspond well to the access to the main memory supported by the hardware. In this respect, these special list constructs are in part

However, there are situations in which recursively structured lists are more efficient. More on this later.

efficiency. There are also such lists in Racket (but not in BSL); there they are called *vectors*.

The attraction of the recursive formulation is that no additional support for lists is required in the programming language. We have seen above that we can program the kind of lists that are directly supported by BSL ourselves. It is simply "yet another recursive datatype", and all constructs, design recipes and so on also work seamlessly for lists. We don't need special loops or other special constructs to process lists, we just do what we do with any other recursive datatype. This is different for built-in index-based lists; there are many examples of language constructs that work for "normal values" but not for lists, and vice versa.

BSL and Racket are in the tradition of the *Scheme* programming language. The philosophy that can be found in the first sentence of the Scheme language specification (<http://www.r6rs.org>) is therefore also valid for BSL and Racket:

Programming languages should be designed not by piling feature on top of feature, but by removing the weaknesses and restrictions that make additional features appear necessary. Scheme demonstrates that a very small number of rules for forming expressions, with no restrictions on how they are composed, suffice to form a practical and efficient programming language that is flexible enough to support most of the major programming paradigms in use today.

The treatment of lists as recursive data types is an example of this philosophy.

Some novice programmers find it counterintuitive to formulate lists and functions recursively. But isn't it better to use a universal tool that can be used in many situations than a special tool that is no better than the universal tool and can only be used in one situation?

9.3.6 Natural numbers as a recursive data structure

In BSL, there are not only functions that consume lists, but also functions that produce lists. One of these is `make-list`. Here is an example:

```
> (make-list 4 "abc")  
'("abc" "abc" "abc"  
  "abc")
```

The `make-list` function therefore consumes a natural number n and a value and produces a list with n repetitions of the value. Although this function only consumes atomic data, it produces an arbitrarily large result. How is that possible?

An illuminating answer to this is that natural numbers can also be seen as instances of a recursive data type. Here is a possible definition:

```
A Nat (Natural Number) is one of:  
; - 0  
; - (add1 Nat)
```

Also in the
In mathematics,
natural numbers are
defined recursively
in a similar way.
Research what the
Peano axioms are.

For example, we can represent the number 3 as `(add1 (add1 (add1 0)))`. The `add1` function therefore has a role similar to the constructor functions in structures. The role of the selector function is assumed by the `sub1` function. The predicate for the first alternative of `Nat` is `zero?`; the predicate for the second alternative is `positive?`

With this view, we are now able to define functions like `make-list` ourselves using our standard design recipe. Let's call our variant of `make-list` `iterate-value`. Here is the specification:

```
X] Nat X -> (List of X)
; creates a list with n occurrences of x
(check-expect (iterate-value 3 "abc") (list "abc" "abc" "abc"))
(define (iterate-value n x) ...)
```

According to our design recipe for recursive data types, we get the following template:

```
(define (iterate-value n x)
  (cond [(zero? n) ...]
        [(positive? n) ... (iterate-value (sub1 n) ...)...]))
```

Completing this template is now just a small step:

```
(define (iterate-value n x)
  (cond [(zero? n) empty]
        [(positive? n) (cons x (iterate-value (sub1 n) x))]))
```

9.4 Several recursive data types at the same time

A more difficult case is when several parameters of a function have a recursive data type. In this case, it usually makes sense to give preference to one of these parameters and ignore the fact that other parameters are also recursive. Which parameter should be preferred is determined by the question of how you can best break down the input of the function so that you can most easily calculate the overall result from the result of the recursive call and the other parameters.

As an example, let's look at a function for concatenating two lists. This function is already built in under the name `append`, but let's recreate it. We have two parameters, both of which have recursive data types. One possibility would be to give preference to the first parameter. This gives us the following template:

```
; [X] (list-of X) (list-of X) -> (list-of X)
; concatenates l1 and l2
(check-expect (lst-append (list 1 2) (list 3 4)) (list 1 2 3
4)) (define (lst-append l1 l2)
  (cond [(empty? l1) ...l2...]
        [(cons? l1) ... (first l1) ... (lst-
append (rest l1) ...)]))
```

In fact, in this case it is easy to complete the template. If we have the result of `(lst-append (rest l1) l2)`, we only have to append `(first l1)` to the front:

```
(define (lst-append l1 l2)
  (cond [(empty? l1) l2]
        [(cons? l1) (cons (first l1) (lst-
append (rest l1) l2))]))
```

Let's now play through the second option: We prefer the second parameter. This results in the following template:

```
(define (lst-append l1 l2)
  (cond [(empty? l2) ...l1...]
        [(cons? l2) ... (first l2) ... (lst-
append .. (rest l2))]))
```

The basic case is trivial, but in the recursive case it is not obvious how we can complete the template. For example, if we look at the recursive call `(lst-append l1 (rest l2))`, we can consider that this result does not help us at all, because we would have to insert `(first l2)` somewhere in the middle (but where exactly is not clear) of the result.

9.5 Design recipe for functions with recursive data types

We have seen how to use a design recipe to design simple functions (§3.3.3 "Design recipe for function definition"), functions with sum types (§5.3.1 "Design with sum types"), functions with product types (§6.7 "Extending the design recipe") and functions with algebraic data types (§7.2 "Program design with ADTs").

Here we summarize how we extend the design recipe to handle data of any size.

1. If there is information of unlimited size in the problem domain, you need a self-referencing data definition. For a self-referencing data definition to be valid, it must fulfill three conditions: 1) The data type is a sum type. 2) There must be at least two alternatives. 3) At least one of the alternatives does not reference the data type just defined, i.e. it is not recursive.

You should provide data examples for recursive data types to validate that your definition makes sense. If it is not obvious how to make your data examples arbitrarily large, something is probably wrong.

2. Nothing changes in the second step: As always, you need a signature, a task description and a dummy implementation.

3. With self-referencing data types, it is no longer possible to specify a test case for each alternative (because there are an infinite number of alternatives if you go into depth). In any case, the test cases should cover all parts of the function. Continue to try to identify and test critical edge cases.
4. Recursive data types are algebraic data types, so the methodology from §7.2 "Program design with ADTs" can be used to design the template. The most important addition to the design recipe concerns the case that an alternative must be implemented that is self-referencing. Instead of using a new auxiliary function in the template as usual, in this case a recursive call of the function that you are currently implementing is included in the template. The call of the selector that extracts the value belonging to the data recursion is included in the template as an argument of the recursive call. For example, if you define a function `(define (f a-list-of-strings ...) ...)` on lists, the call `(f (a-list-of-strings ...) ...)` should be included in the template in the `cons?` case of the function should contain the call `(f (rest a-list-of-strings) ...)`.
5. When designing the function body, we start with the cases of the function that are not recursive. These cases are also called *base cases*, in analogy to base cases in proofs by induction. The non-recursive cases are typically simple and should result directly from the test cases.

In the recursive cases, we need to consider what the recursive call means. To do this, we assume that the recursive call calculates the function correctly, as we specified in the task description in step 2. With this knowledge, we now only have to assemble the available values into the solution.
6. As usual, test whether your function works as desired and check whether the tests cover all interesting cases.
7. For programs with recursive data types, there are some new refactorings that may be useful. Check whether your program contains data types that are not recursive, but that could be simplified by a recursive data type. For example, if your program contains separate data types for employee, group manager, department manager, division manager and so on, this could be simplified by a recursive data type that can be used to model management hierarchies of any depth.

9.6 Refactoring of recursive data types

With regard to the data type refactorings from §??? "[missing]", new type isomorphisms result from "inlining" or expansion of recursive data definitions. For example, in the following example `list-of-number` is isomorphic to `list-of-number2`, because the latter definition results from the first by expanding the recursion once.


```

A list-of-numbers is either:
; - empty
; - (cons Number list-of-number)

(define-struct Number-and-Number-List (num numlist))
; A list-of-number2 is either:
; - empty
; - (make-Number-and-Number-List Number list-of-number)

```

Try to avoid definitions like `list-of-number2`, because functions defined on it are more complex than those that use `list-of-number`. If we use the notation from §???

"[missing]", we can model recursive data types as functions, where the function parameter models the recursive occurrence of the data type. For example, the data type of lists of numbers can be modeled by the function $F(X) = (+ \text{Empty} (* \text{Number } X))$ ³. The nice thing about this notation is that it is very easy to define when recursive data types are isomorphic. The expansion of a recursive data type results from replacing the function parameter X with the right-hand side of the definition, i.e. in the example

$F(X) = (+ \text{Empty} (* \text{Number} (+ \text{Empty} (* \text{Number } X))))$. Inlining is the reverse operation. The justification for these operations arises from the fact that recursive data types can be understood as the smallest fixed point of such functors. For example, the smallest fixed point of $F(X) = (+ \text{Empty} (* \text{Number } X))$ is the infinite sum $(+ \text{Empty} (* \text{Number } \text{Empty}) (* \text{Number } \text{Number } \text{Empty}) (* \text{Number } \text{Number } \text{Number } \text{Empty}) \dots)$. The smallest fixed point does not change due to expansion or inlining, therefore such data

types are isomorphic.

9.7 Program equivalence and induction proofs

Consider the following two functions:

```

; FamilyTree -> Number
; computes the number of known ancestors of
p (check-expect (numKnownAncestors HEINZ) 5)
(define (numKnownAncestors p)
  (cond [(person? p) (+ 1
                        (numKnownAncestors (person-father p))
                        (numKnownAncestors (person-
mother p)))]
        [else 0]))

; FamilyTree -> Number
; computes the number of unknown ancestors of
p (check-expect (numUnknownAncestors HEINZ) 6)

```

³ Such functions are also called *functors* and are known in universal algebra as *F-algebras*. of important significance.

```

(define (numUnknownAncestors p)
  (cond [(person? p) (+ (numUnknownAncestors (person-
father p)))
        (numUnknownAncestors (person-
mother p)))]
    [else 1]))

```

The tests suggest that the following equivalence holds for all persons p : $(+ (\text{numKnownAncestors } p) 1) (\text{numUnknownAncestors } p)$. But how can we show that this property is actually true?

Closing by equations, as we learned in §8.12 "Refactoring expressions and closing by equations", is not sufficient on its own. Because the functions are recursive, we can always generate larger terms using *EFUN*, but we will never get from *numKnownAncestors* to *numUnknownAncestors*.

However, for structurally recursive functions on recursive data types, we can use another very powerful proof principle, namely the principle of *induction*. Consider again the set construction of ft_i from §9.1 "Recursive data types". We know that the type *FamilyTree* is the union of all ft_i . Furthermore, we know that if p is in ft_{i+1} , then $(\text{person-father } p)$ and $(\text{person-mother } p)$ are in ft_i . This justifies the use of the proof principle of induction: We show the desired equivalence for the base case $i = 1$, i.e. $p = \text{\#false}$. Then we show the equivalence for the case $i = n+1$, assuming that the equivalence is already valid for $i = n$. In other words, we show the equivalence for the case $p = (\text{make-person } n \text{ } p1 \text{ } p2)$ under the assumption that the equivalence holds for $p1$ and $p2$.

Typically, this type of induction proof omits the set construction with its indices and "translates" the indices directly into the data type notation. This means that the desired statement is first shown for the non-recursive cases of the data type, and then in the induction step the statement is shown for the recursive cases under the assumption that the statement is already valid for the subcomponents of the data type. This type of induction proof is also called *structural induction*.

We want to prove: $(+ (\text{numKnownAncestors } p) 1) (\text{numUnknownAncestors } p)$ for all persons p . Let us first consider the base case $p = \text{\#false}$.

Then we can close:

```

(+ (numKnownAncestors \#false) 1)
  (according to EFUN and EKONG)
(+ (cond [(person? \#false) ...] [else 0]) 1)
  (according to STRUCT-predfalse and EKONG)
(+ (cond [\#false ...] [else 0]) 1)
  (according to COND-False and EKONG)
(+ (cond [else 0]) 1)
  (according to COND-True and EKONG)
(+ 0 1)

```

(according to *PRIM*)

1

Using *ETRANS*, we can thus close `(+ (numKnownAncestors #false) 1) 1`.
In the same way, we can conclude: `(numUnknownAncestors #false) 1`.
We have thus shown the base case.

For the induction step, we must show the equivalence for $p = (\text{make-person } n \text{ } p1 \text{ } p2)$ and may use the fact that the statement applies to $p1$ and $p2$, i.e. `(+ (numKnownAncestors p1) 1) (numUnknownAncestors p1)` and `(+ (numKnownAncestors p2) 1) (numUnknownAncestors p2)`.

We can now conclude as follows:

`(+ (numKnownAncestors (make-person n p1 p2)) 1)`
(according to *EFUN* and *EKONG*)

```
(+ (cond [(person? (make-person n p1 p2))
          (+ 1
            (numKnownAncestors (person-father (make-person n p1 p2)))
            (numKnownAncestors (person-mother (make-person n p1 p2))))]
    [else 0])
  1)
```

(according to *STRUCT-predtrue* and *EKONG*)

```
(+ (cond [#true
          (+ 1
            (numKnownAncestors (person-father (make-person n p1 p2)))
            (numKnownAncestors (person-mother (make-person n p1 p2))))]
    [else 0])
  1)
```

(according to *COND-True* and *EKONG*)

```
(+ (+ 1
      (numKnownAncestors (person-father (make-person n p1 p2)))
      (numKnownAncestors (person-mother (make-person n p1 p2))))
  1)
```

(according to *STRUCT-select* and *EKONG*)

```
(+ (+ 1
      (numKnownAncestors p1)
      (numKnownAncestors p2))
  1)
```

(according to *EPRIM*)

```
(+  
  (+ (numKnownAncestors p1) 1)  
  (+ (numKnownAncestors p2) 1))
```

(according to induction assumption and *EKONG*)

```
(+  
  (numUnknownAncestors p1)  
  (numUnknownAncestors p2))
```

(according to *STRUCT-select* and *EKOMM* and *EKONG*)

```
(+  
  (numUnknownAncestors (person-father (make-person n p1 p2)))  
  (numUnknownAncestors (person-mother (make-person n p1  
p2)))))
```

(according to *EFUN* and *EKOMM*)

```
(numUnknownAncestors (make-person n p1 p2))
```

We have thus proved the equivalence (using *ETRANS*). This proof is very detailed and in small steps. When you are more practiced in using program equivalences, your proofs will become more detailed and therefore more compact.

The same method of proof can be used for all recursive data types. In particular, it can also be used for lists.

10 Quote and unquote

Lists play an important role in functional languages, especially in the family of languages derived from LISP (such as Racket and BSL).

If you work a lot with lists, it is important to have an efficient notation for them. You have already become familiar with the `list` function, which can be used to write simple lists in a compact format.

However, there is an even more powerful mechanism in BSL/ISL (and many other languages), namely `quote` and `unquote`. This mechanism has existed in LISP since the 1950s, and even today template languages such as Java Server Pages or PHP emulate this model.

To work with `quote` and `unquote`, please change the language level to "Beginner with List Abbreviations" or "Beginning Student with List Abbreviations".

10.1 Quote

The `quote` construct serves as a compact notation for large and nested lists. For example, we can use the notation `(quote (1 2 3))` to create the list `(cons 1 (cons 2 (cons 3 empty)))`. This is not yet particularly impressive, because the effect is the same as

```
> (list 1 2 3)
'(1 2 3)
```

First of all, there is an abbreviation for the key word `quote`, namely the apostrophe, `'`.

```
> '(1 2 3)
'(1 2 3)
```

```
> '("a" "b" "c")
'("a" "b" "c")
```

```
> '(5 "xx")
'(5 "xx")
```

So far, `quote` looks like a minimal improvement of the `list` function. This changes when we use it to create nested lists, i.e. trees.

```
> '(("a" 1) ("b" 2) ("c" 3))
'(("a" 1) ("b" 2) ("c" 3))
```

We can therefore also use `quote` to generate nested lists very easily, with minimal syntactic effort.

The meaning of `quote` is defined by a recursive syntactic transformation.

- `'(e-1 ... e-n)` is transformed to `(list 'e-1 ... 'e-n)`. The transformation is applied recursively to the generated sub-expressions `'e-1` etc.
- If `l` is a literal (a number, a string, a truth value⁴ or an image), then `'l` is transformed to `l`.
- If `n` is a name/identifier, then `'n` is transformed to the *symbol* `'n`.

Ignore the third rule for a second and consider the expression `'(1 (2 3))`. According to the first rule, this expression is transformed in the first step to `(list '1 '(2 3))`. According to the second rule, the sub-expression `'1` becomes `1` and according to the application of the first rule, the sub-expression `'(2 3)` becomes `(list '2 '3)` in the next step. According to the second rule, this expression is again transformed to `(list 2 3)`. Overall, we therefore obtain the result `(list 1 (list 2 3))`.

As you can see, `quote` can be used very efficiently to create nested lists (a form of trees). You may wonder why we didn't use `quote` from the beginning. The reason is that these convenient ways of creating lists hide the structure of lists. In particular, when designing programs (and using the design recipe) you should always keep in mind that lists are composed of `cons` and `empties`.

10.2 Symbols

Symbols are a type of value that you are not yet familiar with. Symbols are used to represent symbolic data. Symbols are related to strings; instead of using quotation marks at the front and back as with a string, `"This is a string"`, symbols are identified by a simple apostrophe: `'this-is-a-symbol`. Symbols have the same syntax as names/identifiers, so spaces, for example, are not allowed.

Unlike strings, symbols are not intended to represent texts. For example, you cannot (directly) concatenate symbols. There is only one important operation for symbols, namely the comparison of symbols using `symbol=?`

```
> (symbol=? 'x
  'x) #true
```

```
> (symbol=? 'x
  'y) #false
```

Symbols are intended to represent "symbolic data". This is data that has an important meaning "in reality", but which we only want to represent with a symbol in the program. Colors are an example of this: `'red`,

⁴ If you want to quote Boolean literals, you must use the syntax `#true` and `#false` for the truth values; you will receive a symbol for `true` and `false`.

'green, 'blue. It makes no sense to consider the names of colors as text. We just want a symbol for each color and be able to compare whether a color is, for example, 'red' (using `symbol=?`).

10.3 Quasi-quota and unquota

The `quote` mechanism holds yet another surprise. Take a look at the following program:

```
(define x 3)
(define y '(1 2 x 4))
```

What value does `y` have after evaluating this program? If you apply the rules above, you will see that the result is not `(list 1 2 3 4)` but `(list 1 2 'x 4)`. The identifier `x` therefore becomes the *symbol* 'x.

Let's look at another example:

```
> '(1 2 (+ 3 4))
'(1 2 (+ 3 4))
```

Anyone expecting the result `(list 1 2 7)` will be disappointed. Applying the transformation rules produces the result: `(list 1 2 (list '+ 3 4))`. The identifier `+` becomes the symbol '+. The symbol '+' has no direct relationship to the addition function, just as the symbol 'x in the example above has no direct relationship to the constant name x.

But what if you want to calculate parts of the (nested) list after all? Let's look at the following function as an example:

```
; Number -> (List of Number)
; given n, generates the list ((1 2) (m 4)) where m is
n+1 (check-expect (some-list 2) (list (list 1 2) (list 3
4))))
(check-expect (some-list 11) (list (list 1 2) (list 12 4)))
(define (some-list n) ...)
```

A naive implementation would be:

```
(define (some-list n) '((1 2) ((+ n 1) 4)))
```

But of course this function does not work as desired:

```
> (some-list 2)
'((1 2) ((+ n 1) 4))
```

Quasiquote is *suited* for such cases. The `quasiquote` construct initially behaves like `quote`, except that it is abbreviated with an oblique apostrophe instead of an even apostrophe:

```
> `(1 2 3)
'(1 2 3)
```

```
> `(a ("b" 5) 77)
'(a ("b" 5) 77)
```

The special thing about `quasiquote` is that you can use it to jump back to the programming language within a quoted area. This option is called "unquote" and is supported by the `unquote` construct. `Unquote` also has an abbreviation, namely the comma character.

```
> `(1 2 ,(+ 3 4))
'(1 2 7)
```

With the help of `quasiquote`, we can now also implement our example from above correctly.

```
(define (some-list-v2 n) `((1 2) (,(+ n 1) 4)))

> (some-list-v2 2)
'((1 2) (3 4))
```

The rules for transforming `quasiquote` are exactly the same as those for `quote` with one additional case: If `quasiquote` meets an `unquote`, both neutralize each other. An expression like ``,e` is thus transformed to `e`.

10.4 S-Expressions

Consider the `person-has-ancestor` function from §9.2 "Programming with recursive data types". A similar function can also be defined for many other structurally organized data types, for example those for representing folder hierarchies in file systems or for representing the hierarchy within a company.

Of course, in addition to `person-has-ancestor`, we could now also implement `file-has-enclosing-directory` and `employee-has-manager`, but these would have a very similar structure to `person-has-ancestor`. We would therefore be violating the DRY principle.

There is a whole range of functions that could be defined on many tree-like data types: Calculate the depth of a tree, search for occurrences of a string, find all "nodes" of the tree that fulfill a predicate, and so on.

In order to be able to define such functions generically (i.e. once for all data types), we need to be able to abstract from the exact structure of data types. This does not work with the "typed" data types that we have looked at so far.

One of the great innovations of the LISP programming language was the idea of a universal data format: a format with which any structured data can be represented in such a way that the data format is part of the data and can be abstracted accordingly. This idea is

typically reinvented every few years; currently, for example, XML and JSON are popular universal data formats.

The mechanism that has existed in LISP since the late 1950s is called *S-expressions*. What are S-expressions? Here is a data definition that describes this exactly:

```
An S-Expression is one of:  
; - a Number  
; - a String  
; - a symbol  
; - a Boolean  
; - an image  
; - empty  
; - a (list-of S-expression)
```

Examples of S-expressions are: `(list 1 (list 'two 'three) "four")`, `"Hi"`. These are not S-expressions: `(make-posn 1 2)`, `(list (make-student "a" "b" 1))`.

S-expressions can be used as a universal data format by making the structuring of the data part of the data. Instead of `(make-posn 1 2)` you can also use the S-expression `'(posn 1 2)` or `'(posn (x 1) (y 2))`; instead of `(make-person "Heinz" (make-person "Horst" false false false) (make-person "Hilde" false false false))` you can also use the S-expression `'(person "Heinz" (person "Horst" #false #false) (person "Hilde" #false #false))` or `'(person "Heinz" (father (person "Horst" (father #false) (mother #false)) (mother (person "Hilde" (father #false) (mother #f))))`.

The advantage of the second variant is that any structured data can be expressed uniformly in this way and the structure itself is part of the data. This makes it possible to define very generic functions that work on any structured data. The disadvantage is that you lose security and typing. It is difficult to say, for example, that a function can only process S-expressions as input that represent a family tree.

The quote operator has the property that it always generates S-expressions. You can even take any definitions or expressions in BSL, write a quote operator around them and you will get an S-expression that represents this program.

```
> (first '(define-struct student (firstname lastname  
matnr))) 'define-struct
```

This property, sometimes called *homoiconicity*, makes it particularly easy to represent programs as data and to write programs that take the representation of a program as input or produce it as output. In Scheme and (full) Racket there is even a function `eval` that takes a representation of an expression as S-expression as input and then interprets this expression and returns the result. For example

`(eval '(+ 1 1))` would `return` result `2`. This makes it possible to calculate programs at runtime and then execute them directly - a very powerful but also very dangerous possibility.

10.5 Application example: Dynamic websites

Since S-Expressions are a universal data format, it is easy to encode other data formats in it, for example HTML (the language in which most web pages are defined).

Together with `quasiquote` and `antiquote`, S-Expressions can therefore be easily used to create dynamic web pages in which the fixed parts are defined as templates. For example, a simple function for creating a dynamic web page could look like this:

```
String String -> S-Expression
; produce a (representation of) a web page with given author
and title
(define (my-first-web-page author title)
  `(html
    (head
      (title ,title)
      (meta ((http-equiv "content-type")
              (content "text-html"))))
    (body
      (h1 ,title)
      (p "I, " ,author ", made this page."))))
```

The function generates the representation of an HTML page in which the transferred parameters are inserted at the desired position. S-expressions and `quasi/antiquote` lead to better readability compared to the variant of the function that assembles the data structure with `cons` and `empty` or `list`. The generated S-expression is not yet HTML, but it can easily be converted to HTML. In Racket, for example, there is the `xexpr->string` and `xexpr->xml` function of the XML library.

```
> (require xml)

> (xexpr->string (my-first-web-page "Klaus Ostermann" "My
Homepage"))
"<html><head><title>My homepage</title><meta
http-equiv=\"content-type\" content=\"text-
html\"/></head><body><h1>My homepage</h1><p>I, Klaus Os-
termann, made this page.</p></body></html>"
```

The string generated by `xexpr->string` is valid HTML and could now be sent to a browser and displayed.

11 DRY: Abstraction everywhere!

This part of the script is based on [HTDP/2e] Part III

We have already learned about the "Don't Repeat Yourself" principle in section §2.6 "DRY: Don't Repeat Yourself!": Good programs should not contain any repetitions and should not be redundant. We have learned that we can eliminate this redundancy through various *abstraction mechanisms*.

In this section, we will first review these already known abstraction mechanisms. However, these mechanisms are not suitable for all types of redundancy. In this chapter, we will therefore get to know other powerful abstraction mechanisms. To do this, we need new language concepts, which you can activate by switching to the "Intermediate with Lambda" language level.

This makes our programming language more complex for the time being. At the end of the chapter, however, we will see that there is such a powerful type of abstraction that we can use it to subsume many other types of abstraction. This will simplify the language conceptually again.

11.1 Abstraction of constants

Probably the simplest type of abstraction is applicable if there are expressions in several places in a program that are constant, i.e. always evaluate to the same value. The abstraction mechanism that we can use in this case is the definition of constants. We have already discussed this possibility in detail in section §2.6 "DRY: Don't Repeat Yourself!".

11.2 Functional abstraction

There is often a situation in a program where expressions occur in many places that are not the same, but only differ in some places in the values used.

Example:

```
(list-of String) -> Boolean
; does l contain "dog"
(define (contains-dog? l)
  (cond
    [(empty? l)
     false] [else
    (or
      (string=? (first l)
        "dog") (contains-dog?
        (rest l))))]))
```

```
(list-of String) -> Boolean
```

```

; does l contain "cat"
(define (contains-cat? l)
  (cond
    [(empty? l) false]
    [else
     (or
      (string=? (first l)
                 "cat") (contains-cat?
                        (rest l))))]))

```

The expressions in the two function bodies are identical except for the string "dog" or "cat" respectively.

Good programmers are too lazy to write many similar expressions and functions. *Functional abstraction* is suitable for eliminating this type of redundancy:

```

; String (list-of String) -> Boolean
; to determine whether l contains the string s
(define (contains? s l)
  (cond
    [(empty? l) false]
    [else (or (string=? (first l) s)
               (contains? s (rest l)))]))

```

If desired, `contains-dog?` and `contains-cat?` can be restored on the basis of this function:

```

(list-of String) -> Boolean
; does l contain "dog"
(define (contains-dog? l)
  (contains? "dog" l))

(list-of String) -> Boolean
; does l contain "cat"
(define (contains-cat? l)
  (contains? "cat" l))

```

Alternatively, the callers of `contains-dog?` and `contains-cat?` can be modified so that `contains?` is used instead.

Once you have defined `contains?`, it will never again be necessary to define a function similar to the first variant of `contains-dog?`

11.3 Functions as function parameters

Consider the following code:

```

; (list-of Number) -> Number

```

```

; adds all numbers in l
(define (add-numbers l)
  (cond
    [(empty? l) 0]
    [else
     (+ (first l)
        (add-numbers (rest l)))]))

```

```

; (list-of Number) -> Number
; multiplies all numbers in
l (define (mult-numbers l)
  (cond
    [(empty? l) 1]
    [else
     (* (first l)
        (mult-numbers (rest l)))]))

```

In this example, we have a similar situation as with `contains-dog?` and `contains-cat?`: Both functions only differ in two places: a) In the `empty?` case, `0` is returned once and `1` is returned once. b) In the other case, one is added and the other multiplied.

Case a) can be solved completely analogously to `contains-dog?` and `contains-cat?` by passing this value as a parameter.

We have a different situation in case b). Here the functions do not differ in a value, but in the name of a called function! For this we need a new type of abstraction, namely the ability to pass functions as parameters to other functions.

If we ignore this problem for a moment and just do the obvious for good luck:

```

(define (op-numbers op z l)
  (cond
    [(empty? l) z]
    [else
     (op (first l)
         (op-numbers (rest l) z))]))

(define (add-numbers l) (op-numbers + 0 l))
(define (mult-numbers l) (op-numbers * 1 l))

```

... we realize that we can actually abstract over functions in exactly the same way as over values! So we can pass functions as parameters and use these parameters in the first position of a function call!

The crucial thing about `op-numbers` is not that `add-numbers` and `mult-numbers` are now one-liners. The crucial point is that we now have a very

powerful abstract function that we can now use universally for many other things.

On the one hand, we can see that `op-numbers` no longer contain any code at all, which is specific to numbers. This indicates that we can also use the function with lists of other values. The name `op-numbers` is therefore misleading and we rename the function:

```
(define (op-elements op z l)
  (cond
    [(empty? l) z]
    [else
     (op (first l)
         (op-elements op z (rest l)))]))
```

Secondly, we can pass not only primitive functions as parameters for `op`, but also any self-defined functions.

Let's take a look at some examples of what you can do with `op-elements`:

We can add up numbers:

```
> (op-elements + 0 (list 5 8 12))
25
```

We can connect strings together:

```
> (op-elements string-append "" (list "ab" "cd"
"ef")) "abcdef"
```

We can compose images:

```
> (op-elements beside empty-image (list (circle 10 "solid" "red")
                                         (rectangle 10 10 "solid"
"blue") (circle 10 "solid"
"green"))))
```



We can copy a list, which is not very useful on its own:

```
> (op-elements cons empty (list 5 8 12 2 9))
'(5 8 12 2 9)
```

...but shows that we can do other interesting things with slight variations of it. For example, we can append two lists together:

```
(define (append-list l1
  l2) (op-elements cons l2
  l1))

> (append-list (list 1 2) (list 3 4))
'(1 2 3 4)
```

We can turn a list of lists into a list:

```
> (op-elements append-list empty (list (list 1 2) (list 3 4) (list 5 6)))  
'(1 2 3 4 5 6)
```

And finally, as a slightly more advanced example, we can use `op-elements` to sort a list of numbers. For this purpose, we need a helper function that inserts a number into a list of sorted numbers:

```
; A (sorted-list-of Number) is a (list-of Number) which is  
sorted by "<"  
  
Number (sorted-list-of Number) -> (sorted-list-of Number)  
; inserts x into a sorted list xs  
(check-expect (insert 5 (list 1 4 9 12)) (list 1 4 5 9 12))  
(define (insert x xs)  
  (cond [(empty? xs) (list x)]  
        [(cons? xs) (if (< x (first xs))  
                        (cons x xs)  
                        (cons (first xs) (insert x (rest  
xs)))))]))
```

And now we can build a sorting function from `insert` and `op-elements`:

```
> (op-elements insert empty (list 5 2 1 69 3 66 55))  
'(1 2 3 5 55 66 69)
```

It is not so important if you do not understand this last example. It is merely intended to demonstrate what you gain when you define reusable functions through abstraction.

Incidentally, the design of functions such as `op-elements` is not just about the reuse of code, but also the "reuse" of correctness arguments. For example, consider the question of whether a program is terminated or perhaps contains an infinite loop. This is one of the most common errors in programs and always a potential problem with loops, as in the examples above. Once we show that `op-elements` terminates, we know that all loops that we create with the help of `op-elements` also terminate (provided that the function that we pass as an argument terminates). If we do not use `op-elements` we would have to make this consideration anew every time.

This sorting function is called *insertion sort* in the algorithm. The function `op-elements` is so useful that it is also provided as a primitive function called `foldr`.

11.4 Abstraction in signatures, types and data definitions

11.4.1 Abstraction in signatures

Redundancy can also occur in signatures, in the sense that there can be many signatures for the same function body.

For example, let's look at the function `second`, which is the second element from of a list of strings:

```
; (list-of String) -> String
(define (second l) (first (rest
l)))
```

Here is a function that calculates the second element from a list of numbers:

```
; (list-of Number) -> Number
(define (second l) (first (rest
l)))
```

The function definitions are identical except for the signature. We could write several signatures for the same function so as not to duplicate the code:

```
; (list-of String) -> String
; (list-of Number) -> Number
(define (second l) (first (rest
l)))
```

Obviously, this is not a very good idea, because we have to extend the list of signatures whenever the function is to be used with a new element type. It is also not possible to add all signatures, because there are an infinite number, e.g. the following infinite sequence of signatures:

```
; (list-of String) -> String
(list-of (list-of String)) -> (list-of String)
; (list-of (list-of (list-of String))) -> (list-of (list-of
String))
; (list-of (list-of (list-of (list-of String)))) -> (list-
of (list-of (list-of String)))
; ...
```

However, we have already learned informally about an abstraction mechanism with which we can eliminate this type of redundancy: Type variables. A signature with type variables implicitly stands for all possible signatures that result when the type variables are replaced by types. We mark names as type variables by placing the name of the type variable in square brackets in front of the signature. We can then use the type variable in the signature. Here is the example from above with type variables:

```
X] (list-of X) -> X
(define (second l) (first (rest l)))
```

The type with which a type variable is replaced may be arbitrary, but it must be the same for all occurrences of the type variable. For example, this signature is not equivalent to the previous one, but false:

```
; [X Y] (list-of X) -> Y
(define (second l) (first (rest l)))
```

The reason is that the signature can also be used for concrete signatures such as

```
(list-of Number) -> String
```


but this is not a valid signature for [second](#).

11.4.2 Signatures for arguments that are functions

In this chapter, we have introduced the possibility of passing functions as parameters to other functions. However, we do not yet know how to describe the signatures of such functions.

We solve this problem by allowing signatures to be used as types. For example, consider a function that combines an image with a circle, but makes the way it is to be combined a parameter of the function:

```
(define (add-circle f img)
  (f img (circle 10 "solid" "red")))
```

For example, this function can be used in this way:

```
> (add-circle beside (rectangle 10 10 "solid" "blue"))
```



or something:

```
> (add-circle above (rectangle 10 10 "solid" "blue"))
```



We can now describe the signature of `add-circle` as follows:

```
; (Image Image -> Image) Image ->
Image (define (add-circle f img)
  (f img (circle 10 "solid" "red")))
```

This signature states that the first parameter must be a function with the signature `Image - Image`. We now use signatures as types or, in other words, we no longer distinguish between signatures and types.

Functions can not only receive functions as arguments, but can also return functions as results. Example:

```
Color -> Image
(define (big-circle color) (circle 20 "solid" color))

Color -> Image
(define (small-circle color) (circle 10 "solid" color))

; String -> (Color -> Image)
(define (get-circle-maker
  size)
  (cond [(string=? size "big") big-circle]
        [(string=? size "small") small-
  circle]))
```

We can now call the `get-circle-maker` function and it returns a function. This means that we can have a call to `get-circle-maker` in the *first* position of a function call. Previously, this position always contained the name of a function or (since we know functions as parameters) the names of function parameters.

Example:

```
> ((get-circle-maker "big") "cyan")
```



Note the brackets in the example: This call is a function call of a function with one parameter. The function we are calling is `(get-circle-maker "big")` and its parameter is `"cyan"`. This is therefore completely different from the similar-looking, but in this example nonsensical expression `(get-circle-maker "big" "cyan")`.

11.4.3 Higher order functions

Function types can be nested as required. For example, let's assume we have other functions with the same signature as `add-circle`, such as `add-rectangle`. A function that we can parameterize with one of these functions is, for example:

```
((Image Image -> Image) Image -> Image) Image Image -> Image
(define (add-two-imgs-with f img1 img2)
  (above (f beside img1) (f beside img2)))
```

We can now parameterize this function with functions such as `add-circle`:

```
> (add-two-imgs-with add-circle (circle 10 "solid" "green") (circle 10 "solid" "blue"))
```



Functions whose signature only contains a single arrow (i.e. functions that have no functions as parameters or return no functions) are also called *first-order functions* in this context. Examples of first-order functions are `circle` or `cons`. Functions that receive functions as parameters or return them as a result are also called *higher-order functions*. Examples of higher-order functions are `add-circle` and `add-two-imgs-with`.

With higher-order functions, a distinction is sometimes made between the maximum nesting depth of the function arrows in the arguments of a function. For example, `add-circle` is called a *second-order function* and `add-two-imgs-with` is called a *third-order function*.

In everyday programming, second-order functions are very common, but functions of third and higher order are rarely used.

11.4.4 Higher order polymorphic functions

Polymorphic functions can also be higher-order functions. We have already seen the `op-elements` function above. Let's take a look at the types with which it is used in the examples.

In the example `(op-elements + 0 (list 5 8 12))` we obviously need the signature for `op-elements`

```
(Number Number -> Number) Number (list-of Number) -> Number
```

while for `(op-elements string-append "" (list "ab" "cd" "ef"))` the signature

```
(String String -> String) String (list-of String) -> String
```

expect. These examples suggest the following generalization by type variables:

```
X] (X X -> X) X (list-of X) -> X
(define (op-elements op z l)
  (cond
    [(empty? l) z]
    [else
     (op (first l)
          (op-elements op z (rest l)))]))
```

However, this signature does not match all the examples above. For example, for the `(op-elements cons empty (list 5 8 12 2 9))` we need the signature

```
(Number (list-of Number) -> (list-of Number)) (list-of Number)
(list-of Number) -> (list-of Number)
```

There is no replacement of `X` with a type in the signature above that creates this signature. If we take a closer look at the function definition, however, we can also find a more general signature, namely this one:

```
X Y] (X Y -> Y) Y (list-of X) -> Y
(define (op-elements op z l)
  (cond
    [(empty? l) z]
    [else
     (op (first l)
          (op-elements op z (rest l)))]))
```

The signature is more general because it stands for more concrete signatures (without type variables) and is sufficient for all the examples we have looked at.

11.4.5 Abstraction in data definitions

In the case of generic data types such as those for lists or trees, we have already seen that in this case you want to write non-redundant data definitions as follows:

```
; a List-of-String is either
; - empty
; - (cons String List-of-String)

; a List-of-Numbers is either
; - empty
; - (cons Number List-of-Number)
```

Similar to function signatures, we abstract over the differences with the help of type variables:

```
; a (list-of X) is either
; - empty
; - (cons X (list-of X))
```

In contrast to function signatures, we do not mark the type variables used separately for data definitions (as with function signatures with square brackets), because with data definitions it is clear from the position of the type variable in the first line of a data definition that it is a type variable.

We can also use data definitions to specify many other properties of the values described by the data definition. For example, we can define the set of non-empty lists:

```
; a (nonempty-list-of X) is: (cons X (list-of X))
```

11.4.6 Grammar of types and signatures

We can describe the types and signatures with which we can now program using a grammar. This grammar reflects the recursive structure of types well. Not all types that can be formed with this grammar are meaningful. For example, `xXy` is a type that is not meaningful. Only types in which all occurring type variables have been bound by a type abstraction of the form `[xXy] xTypy` above them in the derivation tree are meaningful.

```
xType ::= xBasistype
      | xDatentyp
      | ( [ xTypKonstruktory xTypy+ ] )
      | ( [ xTypy+ -> xTypy ] )
      | xXy
      | [ xXy+ ] xTypy
xBasic type ::= Number
```

```

      | String
      | Boolean
      | Image
xDatatype ::= Posn
      | WorldState
      | ...
xTypeConstructor ::= list-of
      | tree-of
      | ...
xXy ::= X
      | Y
      | ...

```

We have not yet said how we should interpret function types. For now, we can interpret it as the set of all functions that satisfy this type. We will specify this later.

11.5 Local abstraction

Constants, functions or structures are often only required *locally*, within a function. In this case, it is possible to define definitions locally using `local` expressions.

11.5.1 Local functions

First, let's look at local functions. Here is an example:

```

; (list-of String) -> String
; appends all strings in l with blank space between
elements (check-expect (append-all-strings-with-
space (list "a" "b" "c")) " a b c ")
(define (append-all-strings-with-space l)
  (local (; String String -> String
          ; juxtapose two strings and prefix with space
          (define (string-append-with-space s t)
            (string-append " " s t)))
    (foldr string-append-with-space
            " "
            l)))

```

Note that `foldr` the name of the built-in function is that our `op-elements` corresponds.

) The `string-append-with-space` function is a *local* function that is only visible within the `append-all-strings-with-space` function. It cannot be called outside of `append-all-strings-with-space`.

Since `local` expressions are expressions, they can appear anywhere where an expression is expected. They are often used as the outermost expression of a function body, but they can also be in any other position.

These expressions are therefore equivalent:

```
> (local [(define (f x) (+ x 1))] (+ (* (f 5) 6)
7)) 43
```

```
> (+ (local [(define (f x) (+ x 1))] (* (f 5) 6))
7) 43
```

An extreme case is to pull the local definition so far inwards that it is positioned directly at the point where it is used.

```
> (+ (* ( (local [(define (f x) (+ x 1))]) f) 5) 6)
7) 43
```

We will see later that there is a better notation for this case.

11.5.2 Local constants

Not only functions can be local, but also constants and structure definitions.

For example, let's look at a function for exponentiating numbers with of potency 8:

```
(define (power8 x)
  (* x (* x (* x (* x (* x (* x (* x (* x x))))))))
```

For example:

```
> (power8 2)
256
```

This function requires eight multiplications to calculate the result. A more efficient method for calculating the power uses the property of the exponential function that $a^{2*b} = ab * ab$.

```
(define (power8-fast x)
  (local
    [(define r1 (* x x))
     (define r2 (* r1 r1))
     (define r3 (* r2 r2))]
    r3))
```

Using a sequence of local constant declarations, we can therefore represent the intermediate results and calculate the result with just 3 multiplications. These constant declarations could not be replaced by (global) constant declarations, because their value depends on the function parameter.

In general, local constants are used for two reasons: 1) to avoid redundancy, or 2) to give intermediate results a name. We will come back to the first point in a moment; here is an example of the second case:

In section §6.5 "Case study: A ball in motion", we programmed a function that adds a motion vector to a position:

Alternatively
in the body
short spelling (*
x x x x x x x
x x x).

```
(define (posn+vel p q)
  (make-posn (+ (posn-x p) (vel-delta-x q))
              (+ (posn-y p) (vel-delta-y q))))
```

Using local constants, we can give the intermediate results for the new x and y coordinates a name and separate their calculation from the construction of the new position:

```
(define (posn+vel p q)
  (local [(define new-x (+ (posn-x p) (vel-delta-x q)))
          (define new-y (+ (posn-y p) (vel-delta-y
                              q)))]
    (make-posn new-x new-y)))
```

Introducing names for intermediate results can help to make code easier to read because the name helps to understand what the intermediate result represents. It can also serve to flatten a very deep nesting of expressions.

The other important motivation for local constants is to avoid redundancy. In fact, we can even avoid two different types of redundancy with local constants: Static redundancy and dynamic redundancy.

Static redundancy refers to our DRY principle: we use local constants to avoid having to repeat ourselves in the program text. This is illustrated by our first example. Let's illustrate this by de-optimizing our `power8-fast` function again by replacing all occurrences of the constants with their definition:

```
(define (power8-fast-slow
  x) (local
      [(define r1 (* x x))
       (define r2 (* (* x x) (* x x)))
       (define r3 (* (* (* x x) (* x x))) (* (* x x) (* x x)))]
    r3))
```

Except for the associativity of the multiplications, this function is equivalent to `power-8` from above. Obviously, however, the DRY principle is violated here because the sub-expressions `(* x x)` and `(* (* x x) (* x x))` occur several times. This redundancy is avoided by using the local constants in `power8-fast`.

The second facet of redundancy that we can avoid by using local constants is dynamic redundancy. This means that we can avoid evaluating an expression multiple times. This is because the value of a local constant is only calculated once when it is defined and only the result that has already been calculated is used in the following. In the `power8-fast` example, we have seen that this enabled us to reduce the number of multiplications from 8 to 3. In general, the definition of a local `let` is "worthwhile" from the point of view of dynamic redundancy if it is evaluated more than once.

There are even programming languages and programming styles in which you have to give *each* intermediate result a name. If you are interested in this, research what *three address code*, *administrative normal form* or *continuation passing style* means.

is.

In summary, local constants have the same purpose as non-local (global) constants, namely the naming of constants or intermediate results and the avoidance of static and dynamic redundancy; however, local constants are more powerful in that they can use the current function parameters and other local definitions.

11.5.3 Intermezzo: Successive Squaring

As an advanced example of the use of local constants, we can consider the generalization of the exponentiation example to arbitrary exponents. This section can be skipped.

To illustrate the effect of avoiding dynamic redundancy even more clearly, let us consider the generalization of `power8` to arbitrary (natural number) exponents. If we understand the exponent as described in Section §9.3.6 "Natural numbers as a recursive data structure" as an instance of a recursive data type, the following definition results:

```
NaturalNumber Number -> Number
(define (exponential n x)
  (if (zero? n)
      1
      (* x (exponential (sub1 n) x)))))
```

The redundancy in this program is not obvious; it only becomes apparent when you look at the "unfolding" of the definition for a fixed exponent, i.e. the version that results when you expand the recursive calls for a fixed `n`. For example, unfolding `exponential` for the case `n`

This "unfolding" is a special case of a more general technique called *partial evaluation*.

[illegible]

A small complication of applying the technique from above to the general case is that the exponent is generally not a power of 2. We deal with this problem by distinguishing between cases where the exponent is an even or an odd number. The overall result is this algorithm, which is also known in the literature as exponentiation by *successive squaring*.

```
(define (exponential-fast x n)
  (if (zero? n)
      1
      (local
        [(define y (exponential-fast x (quotient n 2)))
         (define z (* y y))]
        (if (odd? n) (* x z) z))))
```

In the `power8` example above, we have reduced the number of multiplications required from 8 to 3. In this general version, the difference is even more drastic:

Think about why
exponential-almost
only about $\log_2(n)$
multiplications
required.

The `exponential` function requires n multiplications while `exponential-`requires `almost` only about $\log_2(n)$ multiplications. This is a huge difference. For example, on my computer the evaluation of `(exponential 2 100000)` takes almost a second, while `(exponential-fast 2 100000)` takes less than a millisecond. Try it out for yourself. For example, you can use the `time` function to measure performance.

11.5.4 Local structures

It is also possible to define structures locally with `define-struct`. We will not be using this option for the time being.

11.5.5 Scope of local definitions

The *scope* of a name definition is the area in the program in which a use of the name refers to this definition. Our programming language uses *lexical scoping*, also known as *static scoping*. This means that the scope of a local definition is the sub-expressions of the local expression.

In this example, all uses of `f` and `x` are in the scope of their definitions.

```
(local [(define (f n) (if (zero? n)
                          0
                          (+ x (f (sub1 n)))))
        (define x 5)]
  (+ x (f 2)))
```

In this expression, however, the second use of `x` is not in the scope of the definition:

```
(+ (local [(define x 5)] (+ x 3))
   x)
```

It can happen that there are several definitions of a name (constants, function, structure constructors/selectors) that have an overlapping scope. In this case, the use of the name always refers to the syntactically closest definition. So if you go outwards in the abstract syntax tree of a program, the first definition that you encounter on the way from the occurrence of a name to the root of the tree "wins".

Example: Copy the following program into the definition area of Dr- Racket and click on "Syntax check". Now move the mouse pointer over a definition or use of a function or constant name. The arrows that are now displayed illustrate which definition a name refers to.

```
(add1 (local
```

```

[(define (f y)
  (local [(define x 2)
          (define (g y) (+ y x))]
    (g y)))
 (define (add1 x) (sub1 x))]
(f (add1 2)))

```

11.6 Functions as values: Closures

Consider the following function for numerically calculating the derivative of a function:

```

(Number -> Number) Number -> Number
(define (derivative f x)
  (/ (- (f (+ x 0.001)) (f x))
     0.001))

```

This function gives us the (numerical) derivative of a function at a specific point.

But it would actually be better if this function gave us the derivative itself as a function as a result. For example, if we have a function

```

(Number -> Number) -> Image
(define (plot-function f) ...)

```

we would not be able to draw the derivative of a function with it. Actually, we would like the `derivative` to have the signature

```

(Number -> Number) -> (Number -> Number)
(define (derivative f) ...)

```

has. But how can we implement this version of `derivative`? Obviously, the function we return cannot be a globally defined function, because this function *depends* on the parameter `f`. However, we can define this function *locally* (and also structure it a little better), and then return this function:

```

(Number -> Number) -> (Number -> Number)
(define (derivative f)
  (local
    [(define delta-x 0.001)
     (define (delta-f-x x) (- (f (+ x delta-x)) (f x)))
     (define (g x) (/ (delta-f-x x) delta-x))]
    g))

```

What kind of value is it that is returned by `derivative`? Let's say it's something like the definition of `g`, so `(define (g x)`

`(/ (delta-f-x x) delta-x))`. If this were the case, what happens to the constant `delta-x` and the function `delta-f-x`? These are only bound locally in `derivative`. What happens, for example, when evaluating this expression?

```
(local [(define f (derivative
  exp)) (define delta-x
  10000)]
  (f 5))
```

Is the `delta-x` constant evaluated to about `10000` during the evaluation of `(f 5)`? This would be a violation of lexical scoping and would lead to unpredictable interactions between different parts of the program. Obviously, `delta-x` in the returned function should always refer to the local definition.

If we treat a function as a value, this value is more than just the definition of the function. It is the definition of the function *and* the set of locally defined names (constants, functions, structures, function parameters). This combination of function definition and local *environment* is called *function closure*. The evaluation of a function (not a function call) therefore results in a closure. If this closure is used again at some point later as part of a function call, the saved local environment is reactivated. We will explain the concept of closure in more detail later; for the moment, we note that the evaluation of a function results in the function definition plus its local environment, and this local environment is used when applying the closure to evaluate names in the function body.

11.7 Lambda, the ultimate abstraction

Suppose you want to use the `map` function to duplicate all elements in a list of numbers. You could do this as follows:

```
(local [(define (double x) (* 2 x))]
  (map double lon))
```

This expression is more complicated than it needs to be. For a function as simple as `double`, which is only used locally, it is a waste to assign a name and use a complete extra line of code.

In ISL+ (the language level of HTDP that we currently use), there is therefore the option of defining *anonymous* functions, i.e. functions without a name. Anonymous functions are identified with the keyword `lambda`. This is why such functions are also called *lambda expressions*.

Here is the example from above, but rewritten so that instead of `double` a anonymous function is defined.

```
(map (lambda (x) (* 2 x)) lon)
```

A `lambda` expression generally has the form `(lambda (x-1 ... x-n) exp)` for names `x-1,...,x-n` and an expression `exp`. Its meaning corresponds roughly to a local function definition of the form `(local [(define (f x-1 ... x-n) exp)] f)`. For example, we could have written the expression from above like this:

```
(map (local [(define (double x) (* 2 x))] double) lon)
```

This is not an exact "desugaring". Local functions can be recursive; `lambda` expressions cannot. However, `lambda` **does** contribute to the simplification of the language, because we can now "desugar" function definitions to constant definitions:

```
(define (f x-1 ... x-n) exp)
```

corresponds to

```
(define f (lambda (x-1 ... x-n) exp))
```

The `lambda` expressions therefore make it very clear that functions are "completely normal" values to which we can bind constants. So if we have a function call `(f e-1 ... e-n)` in the above definition of `f`, the `f` in the first position is not a function name, but a constant name that is evaluated to a λ -expression. In general, a function call therefore has the syntax `(exp-0 exp-1 ... exp-n)`, whereby all `exp-i` are arbitrary expressions but `exp-0` must result in a function (more precisely: a closure) during evaluation.

The word `lambda` expression comes from the λ -calculus, which was developed by Alonzo Church in the 1930s. The λ -calculus is a sublanguage of ISL+: In the λ -calculus, there are only λ -expressions and function applications - no numbers, no truth values, no constant definitions, no lists or structs, and so on. Nevertheless, λ -expressions are so powerful that you can recreate all these programming language features within the λ -calculus, for example with the help of so-called *church codes*.

Incidentally, you can also use the Greek letter λ directly in the program text instead of the written-out word `lambda`. So you can also write:

```
(map ( $\lambda$  (x) (* 2 x)) lon)
```

However, it is possible to simulate recursion with `lambda` expressions using so-called *fixed point combinators*.

Take a look at DrRacket under "Insert" to find out which key combination you can use to insert the letter λ into your program.

11.8 Why abstract?

Programs are like books: they are written for people (programmers) and can also be run on a computer. In any case, programs, just like books, should not contain unnecessary repetitions, because nobody wants to read such programs.

Adhering to the DRY principle by creating good abstractions has many advantages. In programs with recurring patterns, there is always a risk of inconsistency, as you have to correctly recreate the pattern at every point where it reappears. We have also seen that not having good abstractions and repeating yourself often can lead to a significant increase in code size.

However, the most important advantage of good abstractions is the following: For each coherent functionality of the program, there is exactly one place where it is implemented. This property makes it much easier to write and maintain a program.

If you have made a mistake, the error is localized in one place and not duplicated many times. If you want to change the functionality, there is one place where you have to change something and you don't have to find all occurrences of a pattern (which can be difficult or practically impossible). You can prove important properties, such as termination or correctness, once for the abstraction and it then automatically applies to all uses of it.

These advantages apply to *all* types of abstraction that we have become familiar with so far: Global and local function and constant definitions, abstract type signatures, abstract data definitions.

For this reason, we formulate the following guideline as a specification of the DRY principle:

Define an abstraction instead of copying a part of a program and then modifying it.

This guideline does not only apply during the initial programming of a program. During the further development and maintenance of programs, you should also always pay attention to whether there are any violations of this principle in your program and eliminate these violations by defining suitable abstractions.

12 Meaning of ISL+

In this chapter, we will precisely define the new language features compared to the last formal language definition from §8 "Meaning of BSL". Our methodology remains the same. We first define the semantics of a core language, which contains all essential language features. Then we define an evaluation rule for programs and definitions as well as a reduction relation with which programs can be evaluated step by step.

12.1 Syntax of Core-ISL+

The two newly added language features in ISL are 1) local definitions and 2) the possibility to consider functions as values. To keep the definitions as simple as possible, we will first remove language features that we have already defined precisely in §8 "Meaning of BSL" and that are not affected by the addition of the new features: Structure definitions (`define-struct`), conditional expressions (`cond`) and logical operators (`and`).

Furthermore, the uniform treatment of functions and values results in even more simplification options. It is not necessary to support constant definitions and function definitions, because every function definition (`define (f x) e`) can be replaced by a constant definition (`define f (lambda (x) e)`) can be expressed. Overall, this results in the following syntax for the Core language:

```
xdefinitiony ::= ( define xnamey xey )
xey          ::= ( xeyxey+ )
              | ( local [ xdefinitiony+ ] xey )
              | xnamey
              | xvy
xvy          ::= ( lambda ( xnamey+ ) xey )
              | xnumbery
              | xbooleany
              | xstringy
              | ximagey
              | x+y
              | x*y
              | x...y
]
```

12.2 Values and environments

In ISL too, the evaluation of programs always takes place in an environment. As there are no longer any structure and function definitions in the core language, an environment is now just a sequence of constant definitions.

```
xenvy        ::= xenv-elementy*
xenv-elementy ::= ( define xnamey xvy )
```

12.3 Importance of programs

The (*PROG*) rule from §8.8 "Meaning of programs" remains almost unchanged for ISL (without the parts for the evaluation of structure and function definitions):

(*PROG*): A program is executed from left to right and starts with the empty environment. If the next program element is an expression, it is evaluated to a value in the current environment according to the rules below. If the next program element is a constant definition (`define x e`), `e` is first evaluated to a value `v` in the current environment, unless `e` is already a value, and then (`define x v`) is added to the current environment (by appending it to the end of the environment).

However, there is an important difference, namely that the (*LOCAL*) rule below can change the rest of the program to be evaluated during the evaluation.

12.4 Evaluation positions and the congruence rule

The evaluation context specifies that function calls are evaluated from left to right, whereby, in contrast to BSL, the function to be called is now also an expression that must be evaluated.

$\text{xEy} ::= []$
 $| (\text{xvy} * \text{xEy} \text{xey} *)$

The standard congruence rule also applies in ISL.

(*KONG*): If $e-1 \rightarrow e-2$, then $E[e-1] \rightarrow E[e-2]$.

12.5 Meaning of expressions

12.5.1 Meaning of function calls

The evaluation of function calls changes in that the function that is called generally only arises during the function evaluation.

(*APP*): $((\text{lambda } (name-1 \dots name-n) e) v-1 \dots v-n) \rightarrow e [name-1 := v-1 \dots name-n := v-n]$

The replacement of the formal arguments by the actual arguments in this rule is more complex than it first appears. In particular, a name such as *name-1* may not be replaced by *v-1* if *name-1* occurs in a context in which there is a lexically closer other binding (by `lambda` or `local`) of *name-1*. Example: $((\text{lambda } (x) (+ x 1)) x)[x := 7] = ((\text{lambda } (x) (+ x 1)) 7)$ and not $((\text{lambda } (x) (+ 7 1)) 7)$. We will refer to this in our discussion on Scope in §12.6 "Scope".

(*PRIM*): If v is a primitive function f and $f(v-1, \dots, v-n) = v$, then $(v v-1 \dots v-n) \rightarrow v$.

Primitive functions can also be the result of calculations; for example, in the expression $((\text{if cond } + *) 3 4)$ the function that is used depends on the truth value of the expression `cond`.

12.5.2 Importance of local definitions

The most complex rule is the one for the meaning of local definitions. It uses an evaluation context E as defined above to turn local definitions into global definitions. To avoid name collisions, local definitions are renamed if necessary. Dependencies of local definitions on the local context (e.g. a function argument) were eliminated by previous substitutions if necessary.

(LOCAL): $E[(\text{local } [(\text{define } \text{name-}l \text{ e-}l) \dots (\text{define } \text{name-}n \text{ e-}n)] e)] (\text{define } \text{name-}l' \text{ e-}l') \dots (\text{define } \text{name-}n' \text{ e-}n') E[e']$ where $\text{name-}l', \dots, \text{name-}n'$ are "fresh" names that do not occur anywhere else in the program and $e', e-l', \dots, e-n'$ are copies of $e, e-l, \dots, e-n$ in which all occurrences of $\text{name-}l, \dots, \text{name-}n$ are replaced by $\text{name-}l', \dots, \text{name-}n'$.

The grammatically incorrect notation $(\text{define } \text{name-}l' \text{ e-}l') \dots (\text{define } \text{name-}n' \text{ e-}n') E[e]$ should mean that $E[(\text{local } \dots e)]$ is replaced by $E[e]$ and at the same time the definitions $(\text{define } \text{name-}l' \text{ e-}l') \dots (\text{define } \text{name-}n' \text{ e-}n')$ are included in the program as the next definition to be evaluated using *(PROG)*.

Example:

```
(define f (lambda (x)
  (+ 2
    (local
      [(define y (+ x 1))]
      (* y 2))))))

(f 2)

Then

(f 2)

(+ 2
  (local
    [(define y (+ 2 1))]
    (* y 2)))

(define y_0 (+ 2 1))
(+ 2 (* y_0 2))
```

In this second step, the *(LOCAL)* was used to turn the local definition into a global definition. The dependency on the local context (i.e. the function argument x) was previously eliminated in the first step by using the *(APP)* rule. The evaluation now *continues* by using the *(PROG)*, i.e. we evaluate the constant definition by $(+ 2 1) 3$, add $(\text{define } y_0 3)$ to the environment, and now evaluate in this environment $(+ 2 (* y_0 2))$ to the result 8.

12.5.3 Meaning of constants

The definition of constants is unchanged compared to BSL.

(CONST): `name v`, if `(define name v)` is the last definition of `name` is in *env*.

12.6 Scope

Taking into account the local definitions, we want to revisit the discussion about scope from §11.5.5 "Scope of local definitions" and discuss how the formal definition guarantees lexical scoping. Lexical scoping is visible in two of the rules above: *(APP)* and *(LOCAL)*.

A renaming takes place in *(LOCAL)*: The name of local constants is renamed and all uses of the name in the sub-expressions of the `local` expression are also renamed. The fact that this renaming is carried out precisely in the sub-expressions ensures lexical scoping. The fact that a "fresh" name is used means that no use of the name outside of these sub-expressions can be bound to the renamed definition.

The same behavior can be found in *(APP)*: By replacing the formal parameters with the current arguments only in the body of the function, lexical scoping is ensured. At the same time, this defines how clouds are represented, namely as function definitions in which the names bound "further out" have already been replaced by values.

Example: The expression `(f 3)` in this program

```
(define (f x)
  (lambda (y) (+ x
y))) (f 3)
```

is reduced to `(lambda (y) (+ 3 y))`; the value for `x` is therefore also saved in the closure.

An important aspect of lexical scoping is *shadowing*. Shadowing is a strategy for dealing with the situation where several definitions of a name are in scope at the same time.

Example:

```
(define x 1)
(define (f x)
  (+ x (local [(define x 2)] (+ x 1))))
```

In this example, there are three *binding* occurrences of `x` and two *bound* occurrences of `x`. The left occurrence of `x` in the last line of the example is in the lexical scope of two of the three definitions; the right occurrence of `x` is even in the lexical scope of all three definitions.

Shadowing means that the lexically "closest" definition always "wins" in such situations. By "closest", we mean the definition that is encountered first,

if you go from the name to the outside in the grammatical structure of the program text. The more internal definitions therefore cover the more external definitions: They cast a shadow in which the outer definition is not visible. Therefore, in the example above, the expression `(f 3)` is evaluated to `6`.

Shadowing is justified by a modularity consideration: The meaning of an expression should be readable as locally as possible. In particular, an expression that contains no unbound names (a so-called "closed" term) should have the same meaning everywhere, no matter where it is used. For example, the expression `(lambda (x) x)` should always be the identity function and not, for example, the constant `3` function only because somewhere further out it says `(define x 3)`. This desired behavior can only be guaranteed by lexical scoping with shadowing.

Programming languages that allow names to be defined locally and use lexical scoping with shadowing are also called programming languages with a *block structure*. Block structure was one of the major innovations in the ALGOL 60 programming language. Most modern programming languages today have block structure.

13 Pattern Matching

Many functions consume data that has a sum type or an algebraic data type (i.e. a mixture of sum and product types (§7 "Data definition by alternatives and decomposition: Algebraic data types") with a sum "at the top".

Frequently (and according to our design recipe), such functions look like this: first of all, a distinction is made as to which alternative is currently available, and then the components of the product type present in the alternative are accessed (possibly in auxiliary functions).

For example, functions that process lists usually have this structure.

```
(define (f l)
  (cond [(cons? l) (... (first l) ... (f (rest l))...)]
        [(empty? l) ...]))
```

With *pattern matching*, such functions can be defined with significantly less effort. Pattern matching has two facets: 1) It implicitly defines a condition, analogous to the conditions in the `cond` clauses above. 2) It defines names that can be used instead of the projections (`(first l)` and `(rest l)` in the example).

Pattern matching can be used on all types of sum types. In particular, it is not restricted to recursive types such as lists or trees.

13.1 Pattern matching using an example

To get support for pattern matching, we use the teachpack 2htdp/abstraction. Therefore, insert this instruction at the beginning of your program:

```
(require 2htdp/abstraction)
```

The following example shows how the `match` construct can be used.

```
(define (f x)
  (match
    x [7
      8]
      ["hey" "joe"]
      [(list 1 y 3) y]
      [(cons a (list 5 6)) (add1 a)]
      [(posn 5 5) 42]
      [(posn y y) y]
      [(posn y z) (+ y
                    z)]
      [(cons (posn 1 z) y) z]
      [(? cons?) "non-empty list"]]))
```

Here are some examples that illustrate the behavior of the `match` construct.

```

> (f 7)
8
> (f "hey")
"joe"
> (f (list 1 2
      3))
2
> (f (list 4 5
      6))
5
> (f (make-posn 5 5))
42
> (f (make-posn 6 6))
6
> (f (make-posn 5 6))
11
> (f (list (make-posn 1 6) 7))
6
> (f (list 99 88))
"non-empty list"
> (f 42)
match: no matching clause for 42

```

Each clause in a `match` expression begins with a pattern. A pattern can be a literal, as in the first two clauses (`7` and `"hey"`). In this case, the pattern is merely an implicit condition: If the value being matched (`x` in the example) is equal to the literal, then the value of the overall expression is that of the right-hand side of the clause (analogous to `cond`).

Pattern matching is interesting because it is also possible to "match" lists and other algebraic data types. Names may occur in the patterns (such as the `y` in `(list 1 y 3)`); unlike structure names or literals, these variables are not conditions, but are used to bind the names to the corresponding part of the structure.

However, names can become a condition if they occur more than once in the pattern. In the example above, this is the case in the pattern `(posn y y)`. This pattern only matches if `x` is a `posn` and both components have the same value.

If several patterns match at the same time, the first pattern that matches always "wins" (similarly, as with `cond`, the first clause whose condition results in `true` always "wins"). Therefore, for example, `(f (make-posn 5 5))` in the example gives the result `42` and not `5` or `10`.

The penultimate pattern, `(cons (posn 1 z) y)`, illustrates that patterns can be arbitrarily can be deeply nested.

In the last pattern, `(? list?)`, we see that predicate functions of lists and structures can also be used to check what kind of value we currently have. This type of pattern is useful if you just want to know whether the value we are matching on is a list or a `posn`, for example. In many cases, pattern matching is a useful alternative to the use of

cond expressions. For example, we can use pattern matching to call the function

```
(define (person-has-ancestor p a)
  (cond [(person? p)
        (or
         (string=? (person-name p) a)
         (person-has-ancestor (person-father p) a)
         (person-has-ancestor (person-mother p)
                               a))]
        [else #false])))
```

from §9.1 "Recursive data types":

```
(define (person-has-ancestor p a)
  (match p
    [(person name father
              mother) (or
                       (string=? name a)
                       (person-has-ancestor father a)
                       (person-has-ancestor mother
                                             a))]
    [else #false])))
```

13.2 Pattern matching in general

We look at the syntax, meaning and reduction of pattern matching.

13.2.1 Syntax of pattern matching

To define the syntactic structure of the match expressions, we extend the grammar for expressions from §8.3 "Syntax of BSL" as follows:

```
key ::= ...
      | (match key { [xpatterny key ]+ } )
xpatterny ::= xliteral-constanty
            | xnamey
            | ( xnamey xpatterny* )
            | ( ? xnamey? )
xliteral-constanty ::= xnumbery
                   | xbooleany
                   | xstringy
```

13.2.2 Meaning of pattern matching

If you have an expression of the form (match v [p-1 e-1] ... [p-n e-n]), you can understand pattern matching as the task of finding a minimal i so that p-i "matches" v. But what exactly does this mean? But what exactly does that mean?

We can define matching as a function that receives a pattern and a value as input

and returns either "no match" or a *substitution*. A substitution is a mapping $rx_1 : v_1, \dots, x_n : v_n$ s from names to values. A

Substitution can be applied to an expression. This means that all names in the expression that are mapped to a value in the substitution are replaced by this value. If e is an expression and σ is a substitution, we write $e\sigma$ for the application of σ to e . For example, for $\sigma \text{ rx} : 1, y : 2$ s and $e \text{ p x y zq}$, is

$$e\sigma \text{ p x y zq rx} : 1, y : 2 \text{ s p 1 2 zq}$$

The matching of a value to a pattern is now defined as follows:

matchpv, vq rs
 $\text{matchppname } p_1 \dots p_n q, \text{ make name } v_1 \dots v_n q \text{ matchpp}_1, v_1 q \dots \text{ matchpp}_n, v_n q_n$
 $\text{matchppcons } p_1 p_2 q, \text{ cons } v_1 v_2 q \text{ matchpp}_1, v_1 q \text{ matchpp}_2, v_2 q_2$
 $\text{matchpp? name? q, make name } \dots q \text{ rs}$
 $\text{matchpx, vq rx} : \text{vs}$
 $\text{matchp. } \dots \text{ q no match in all other cases}$

Here \cdot is an operator that combines substitutions. The result of $\sigma_1 \cdot \sigma_2$ is "no match" if σ_1 or σ_2 are "no match" or σ_1 and σ_2 both define a mapping for the same name but these are mapped to different values.

$\text{rx}_1 : v_1, \dots, x_k : v_k \text{ s } \cdot \text{rx}_{k+1} : v_{k+1}, \dots, x_n : v_n \text{ s rx}_1 : v_1, \dots, x_n : v_n \text{ s}$
 if the following applies for all $i, j: x_i x_j v_i v_j$.

Examples:

$\text{matchppmake posn x yq, make posn 3 4 q rx} : 3, y : 4 \text{ s}$

$\text{matchppmake posn 3 yq, make posn 3 4 q ry} : 4 \text{ s}$

$\text{matchpx, make posn 3 4 q rx} : \text{make posn 3 4 s}$

$\text{matchppcons pmake posn x 3q yq, cons make posn 3 3 empty q rx} : 3, y : \text{emptys}$

$\text{matchppcons pmake posn x xq yq, cons make posn 3 4 empty q no match}$

13.2.3 Reduction of pattern matching

We extend the grammar of the evaluation context so that the expression on which the match is made can be reduced to a value. All other sub-expressions of the match expression are not evaluated.

$xEy ::= \dots$
 $| (\text{match } xEy \{ [xpatterny \text{ key}]^+ \})$

In the reduction relation, we now use the *match* function from above to decide whether a pattern matches and, if necessary, to replace the names bound by the pattern in the corresponding expression with the corresponding values. (*MATCH-YES*): If the following applies in an expression $(\text{match } v [p-1 \ e-1] \dots [p-n \ e-n])$: $\text{match}(p-1, v) = \sigma$ and $e-1 \ \sigma = e$, then $(\text{match } v [p-1 \ e-1] \dots [p-n \ e-n]) \ e$.

(*MATCH-NO*): If, however, $\text{match}(p-1, v) = \text{"no match"}$, the following applies: $(\text{match } v [p-1 \ e-1] \dots [p-n \ e-n]) (\text{match } v [p-2 \ e-2] \dots [p-n \ e-n])$.

If there are no more patterns left to match, the evaluation is aborted with a runtime error, as illustrated above in the (f 42) example. This is modeled in the reduction semantics by the fact that the evaluation "gets stuck", i.e. can no longer be further reduced although the expression is not yet a value.

13.2.4 Pattern matching pitfalls

There are some peculiarities regarding the behavior of pattern matching when using literals that may lead to confusion.

Here are a few examples:

```
> (match true [false false] [else
42]) #true
> (match true [#false false] [else
42]) 42
> (match (list 1 2 3) [empty empty] [else 42])
'(1 2 3)
> (match (list 1 2 3) [(list) empty] [else 42])
42
```

The first two examples illustrate that it is important to write the Boolean constants as *#true* and *#false* when they occur in patterns. If you write *false* or *true* instead, these are interpreted as names that are bound by the pattern matching.

The last two examples show that the same phenomenon occurs with list literals. Write *(list)* and not *empty* if you want to match on the empty list.

14 Generative recursion

The structure of functions is often based on the structure of the data on which the functions operate. For example, our design recipe for algebraic data types is to define an auxiliary function for each alternative of the data type. If we have a recursive data type, our design recipe provides for the use of structural recursion.

In some cases, however, it is necessary to deviate from this parallelism of data and functions: The structure of the data does not match the way in which the problem is to be divided into sub-problems.

14.1 Where does the ball collide?

As an example, consider the case study on the ball in motion in §6.5 "Case study: A ball in motion". Suppose we want a function that calculates for a ball at which position the ball has a collision with the wall for the first time. If we have a ball `ball`, we can simulate the movement of the ball by calling `(move-ball ball)`, `(move-ball (move-ball ball))` and so on. How long do we want to run this simulation? Until there is a collision, i.e. until `(collision current-ball)` is not `"none"`.

This justifies the following definition:

```
; Ball -> Posn
; computes the position where the first collision of the
; ball occurs
(define (first-collision ball)
  (cond [(string=? (collision (ball-loc ball))
                  "none") (first-collision (move-ball ball))]
        [else (ball-loc ball)]))
```

If we look at this definition, we notice two special features: 1) The case distinction in the body of the function has nothing to do with the structure of the input. 2) The argument that we pass in the recursive function call is not part of the original input. Instead, `(move-ball ball)` generates a completely new ball, namely the ball that has moved one step forward. Obviously, it is not possible to generate a function of this type with our previous design recipe.

14.2 Fast sorting

Consider the problem of sorting a list of numbers. Using our design recipe results in the following template:

```
; (listof number) -> (listof number)
; to create a list of numbers with the same numbers as
; alon sorted in ascending order
(define (sort l)
```

```

(match l
  [empty
   ...]
  [(cons x xs) ... x ... (sort xs) ...]))

```

In the basic case, completing the template is trivial. In the recursive case, we obviously have to *insert* *x* into the already sorted list *(sort xs)*. To do this, we can use the *insert* function already defined in §11.3 "Functions as function parameters". This results in the following definition:

```

; (listof number) -> (listof number)
; to create a list of numbers with the same numbers as
; alon sorted in ascending order
(define (sort l)
  (match l
    [empty
     empty]
    [(cons x xs) (insert x (sort xs))]))

```

This algorithm, which is also called *insertion sort*, is inevitable if you want to sort a list using structural recursion. However, this algorithm is not very efficient. If we want to sort a list *(list x-1*

... x-n), the expansion of *(sort (list x-1 ... x-n))* results in the expression *(insert x-1 (insert x-2 ... (insert x-n empty) ...))*. In the worst case (e.g. a backwards sorted list), *insert* requires as many calculation steps as the list is long. Since $n + (n-1) + \dots + 1 = n*(n+1)/2$, the result is that the number of calculation steps of the sorting algorithm grows quadratically with the length of the input in the worst case.

A better algorithm results if we break the problem down into sub-problems more cleverly than the structure of the data suggests. A common algorithm is *quick sort*. In this algorithm, we select one element in each step, for example the first list element. This element is called the pivot element. Then we divide the rest of the list into list elements that are smaller (or equal) and those that are larger than the pivot element. If we sort these newly generated lists recursively, we can sort the entire list by appending the two sorted lists with the pivot element in the middle.

Overall, this results in the following definition:

```

; (listof number) -> (listof number)
; to create a list of numbers with the same numbers as
; alon sorted in ascending order
(define (qsort l)
  (match l
    [empty
     empty]
    [(cons x xs)
     (append
      (qsort (smaller-or-equal-than x xs))
      (list x)
      (qsort (larger-than x xs)))]))

```

```
(qsort (greater-than x xs))))))
```

```

; Number (listof Number) -> (listof Number)
; computes a list of all elements of xs that are
; smaller or equal than x
(define (smaller-or-equal-than x xs)
  (filter (lambda (y) (<= y x)) xs))

; Number (listof Number) -> (listof Number)
; computes a list of all elements of xs that are
; greater than x
(define (greater-than x xs)
  (filter (lambda (y) (> y x))
    xs))

```

The recursion structure in this algorithm also differs significantly from the familiar pattern of structural recursion. Although the input of the recursive call is in a sense part of the input (in the sense that the list elements in the recursive calls form a subset of the list elements of the original input), they are not part of the input in the sense of the structure of the input. The case distinction in this example is the same as in structural recursion, but instead of one recursive call as in structural recursion on lists, we have two recursive calls.

It is not easy to see that quick sort is usually much faster than insertion sort (and not the subject of this course), but you can see that if the two lists (`smaller-or-equal-than x xs`) and (`greater-than x xs`) always produce lists of approximately the same size, the recursion-depth is only logarithmic in the length of the list. It can be shown that the number of calculation steps required to sort a list of length n is on average proportional to $n \cdot \log(n)$.

14.3 Design of generative recursive functions

We call recursion structures that do not (necessarily) correspond to the pattern of structural recursion *generative recursion*. A generative recursive recursion has a structure that looks something like the following:

```

(define (generative-recursive-fun problem)
  (cond
    [(trivially-solvable? problem)
     (determine-solution problem)]
    [else
     (combine-solutions
      ... problem ...
      (generative-recursive-fun (generate-problem-
1 problem))
      ...
      (generative-recursive-fun (generate-problem-
n problem))))))

```

This template should make it clear that we need to answer the following five questions when designing a generative recursive function:

1. What is a trivially solvable problem?
2. What is the solution to a trivially solvable problem?
3. How do we generate new problems that are easier to solve than the original problem? How many new problems should we generate?
4. How do we calculate the solution of the original problem from the solutions of the generated problems? Do we need the original problem (or a part of it) again?
5. Why does the algorithm terminate?

We will deal with the last question separately in the next section. The answer to the first four questions for the first example above is: 1) A ball that is already colliding. 2) The current position of the ball. 3) By taking a movement step of the ball (bringing it closer to the collision point). 4) The solution of the generated problem is the solution of the original problem; no further computation is necessary.

The answer to the first four questions for the second example is: 1) Sorting an empty list. 2) The empty list. 3) By selecting a pivot element and generating two new problems: Sorting the list of all elements of the original problem that are smaller (or equal) than the pivot element, and sorting the list of all elements of the original problem that are larger than the pivot element. 4) By concatenating the two sorted lists with the pivot element in the middle.

A generative recursive function should be considered in the following situations:

1) The input has a recursive structure, but it is not possible or too complicated to solve the problem using structural recursion (for example, because the result of the recursive call does not help to solve the original problem). 2) There is a structurally recursive function that solves the problem, but it is not efficient enough. 3) The input is not recursive, but the number of computational steps to solve the problem is not proportional to the size of the input.

If you want to solve a problem using generative recursion, you should first answer the four questions above and then use the template above in the template step of the design recipe (adapted to the answers to questions 1 and 3). You can then use the answers to questions 2 and 4 to complete the implementation of the template. The tests for the function should in any case contain examples for trivial problems as well as for the generative recursive case.

An important difference between structural recursion and generative recursion is that the design of generative recursive functions requires more creativity. In particular, a special mental insight is required as to how meaningful smaller sub-problems can be generated from the problem. With structural recursion, on the other hand, the definition of the function often follows almost automatically from the template.

14.4 Scheduling

An important property of structurally recursive functions is that they always terminate: Since the input data has a finite size and the input becomes genuinely smaller in each recursive call, a non-recursive base case must be reached at some point.

This is different with generative recursion: We have to explicitly consider why a generative recursive function terminates.

Consider a variant of the `qsort` algorithm from above, in which we replace the expression `(smaller-or-equal-than x xs)` with `(smaller-or-equal-than x l)`. So instead of just picking out the smaller-or-equal elements from the remaining list (without the pivot element), we look for the smaller-or-equal elements in the list that also contains the pivot element.

otelement still contains:

```
(define (qsort
  l) (match l
      [empty
       empty]
      [(cons x xs)
       (append
        (qsort (smaller-or-equal-than x l))
        (list x)
        (qsort (greater-than x xs))))))
```

Let us now consider a call such as `(qsort (list 5))`. Since `(smaller-or-equal-than 5 (list 5))` results in the list `(list 5)`, this call will lead to a recursive call of the form `(qsort (list 5))`. We have therefore produced an infinite loop.

How can we avoid this mistake? The termination of a generative recurrent function can be shown by two steps:

1. We define a mapping that maps the set of function arguments to a natural number. This mapping effectively measures the size of the input, where "size" does not necessarily describe the physical size of the data in memory but the size of the problem from the algorithm's point of view.
2. We show that the size of the input with respect to the mapping from the first step becomes strictly smaller in all recursive function calls.

If the size of the original input (with regard to the defined mapping) is therefore n , it is ensured that the maximum recursion depth is also n .

In the case of quick sort, we can use the length of the list `l` as the mapping in the first step. If the length of `(cons x xs)` is n , then the length of `xs` is $n-1$ and therefore `(smaller-or-equal-than x xs)` and `(greater-than x xs)` are not greater than $n-1$. Therefore, the size of the input is strictly smaller in all recursive calls.

In the case of `first-collision`, it is much more complicated to show the termination. Consider how the size of the input can be measured in this case so that

the condition from the second step applies. Hint: In fact

`first-collision` does not always terminate. Use the search for a termination proof to find and correct this error.

15 Accumulation of knowledge

The result of a function call only depends on the function parameters, but not on the context in which a function is called. On the one hand, this property makes functions very flexible, as they can be called easily in any context. On the other hand, however, some problems are such that the problem definition requires a certain dependency on the context.

This is particularly important when designing recursive functions. Here are two examples.

15.1 Example: Relative distance between points

Suppose we are given a list of distances between points, for example `(list 0 50 40 70 30 30)`. The first point is 0 away from the origin, the second point is 50 away from the origin, the third point is 40 away from the second point, so $50+40=90$ away from the origin.

Let's assume we want to transform this list into a list with absolute distances to the origin. Based on this information and the example, we can start implementing this function:

```
(list-of Number) -> (list-of Number)
; converts a list of relative distances to a list of
absolute distances
(check-expect (relative-2-absolute (list 0 50 40 70 30 30))
              (list 0 50 90 160 190 220))
(define (relative-2-absolute alon) ...)
```

According to our design recipe, we can try to solve this problem using structural recursion:

```
(list-of Number) -> (list-of Number)
; converts a list of relative distances to a list of
absolute distances
(check-expect (relative-2-absolute (list 0 50 40 70 30 30))
              (list 0 50 90 160 190 220))
(define (relative-2-absolute alon)
  (match alon
    [empty ...]
    [(cons x xs) ... x ... (relative-2-absolute xs) ...]))
```

The case for the empty list is trivial, but how can we calculate the total result from the result of `(relative-2-absolute xs)` and `x`? The first element of the result is `x`, but obviously we have to add `x` to each element of `(relative-2-absolute xs)`. With this knowledge, we can complete the function definition:

```
(list-of Number) -> (list-of Number)
```

```

; converts a list of relative distances to a list of
absolute distances
(check-expect (relative-2-absolute (list 0 50 40 70 30 30))
              (list 0 50 90 160 190 220))
(define (relative-2-absolute alon)
  (match alon
    [empty empty]
    [(cons x xs) (cons
                  x
                  (map
                   (lambda (y) (+ y x))
                   (relative-2-absolute xs))))))

```

Although this function works as desired, it is problematic. For example, if you use the `time` function to test how long the function takes to process lists of different lengths, you can see that the time required increases quadratically with the length of the list.

This phenomenon can also be seen directly in the function definition, because the complete list calculated so far is processed again in each recursion step by calling `map`.

If we do this calculation by hand, we proceed differently. We only go through the list once from left to right and memorize the sum of the numbers we have seen so far.

However, this form of bookkeeping does not match our previous scheme for recursion. If we have two lists `(cons x1 xs)` and `(cons x2 xs)`, the recursive call has the form `(f xs)` in both cases. What `f` does with `xs` can therefore not depend on `x1` or `x2`.

To solve this problem, we introduce an additional parameter: `accu-dist`. This parameter represents the accumulated distance of the points seen so far. When we start the calculation, `accu-dist` is initialized to `0`. During the calculation, we can convert the first relative distance into an absolute distance by adding `accu-dist`; for the calculation in the recursive call, we have to adapt the value of the accumulator to the new context.

This results in the following code:

```

; (list-of Number) Number -> (list-of Number)
(define (relative-2-absolute-with-acc alon accu-dist)
  (match alon
    [empty empty]
    [(cons x xs)
     (local [(define x-absolute (+ accu-dist x))]
       (cons x-absolute (relative-2-absolute-with-acc xs x-absolute))))))

```

This definition is not yet fully equivalent to `relative-2-absolute`, but we can easily reconstruct the original signature:

```

(list-of Number) -> (list-of Number)

```

```

; converts a list of relative distances to a list of
absolute distances

(check-expect (relative-2-absolute-2 (list 0 50 40 70 30 30))
              (list 0 50 90 160 190 220))

(define (relative-2-absolute-2 alon)
  (local
    [(list-of Number) Number -> (list-of Number)
     [(define (relative-2-absolute-with-acc alon accu-dist)
        (match alon
          [empty
           empty]
          [(cons x xs)
           (local [(define x-absolute (+ accu-dist x))]
             (cons x-absolute
                   (relative-2-absolute-with-acc xs x-
absolute))))))]
      (relative-2-absolute-with-acc alon 0))])

```

Some experiments with `time` confirm that `relative-2-absolute-2` is much more efficient than `relative-2-absolute` and shows only a linear growth of the runtime instead of a quadratic one.

15.2 Example: Search in a graph

As a second example, let's look at the problem of finding a path between two nodes in a graph (if it exists).

A directed graph is a set of nodes and a set of directed edges between the nodes. There are different ways to represent graphs. We opt for a representation in which the edges emanating from each node are stored:

A Node is a symbol

A Node-with-neighbors is a (list Node (list-of Node))

; A Graph is a (list-of Node-with-neighbors)

Here is an example of a graph that corresponds to this data definition:

```

(define
  graph1 '((A
    (B E))
    (B (E F))
    (C (D))
    (D ()))
    (E (C F))
    (F (D G))
    (G ())))

```

We are therefore looking for a function whose task we can define as follows:

```
Node Node Graph -> (list-of Node) or false
; to create a path from origination to destination in G
; if there is no path, the function produces false
(check-member-of (find-route 'A 'G graph1) '(A E F G) '(A B E F G) '(A B F
G)) (define (find-route origination destination G) ...)
```

First of all, we can see that we cannot solve this problem with structural recursion. For example, if we are looking for a route from B to E in `graph1`, the information as to whether such a route exists in `(rest graph1)` is of little use to us, because it could be that the route goes through A.

To solve this problem with generative recursion, we need to answer the five questions described in §14.3 "Design of generative recursive functions":

1. A trivially solvable problem is the question of a route from a node `n` to yourself.
2. The solution in the trivial case is `(list n)`.
3. If the problem is non-trivial, we can generate the problem of finding a route from the neighbor to the destination for each neighbor of the node.
4. If one of these problems is solved successfully, the overall result is the route to the neighbor followed by the route found from the neighbor.
5. We will postpone the scheduling argument until later.

This results in the following program:

```
Node Node Graph -> (list-of Node) or false
; to create a path from origination to destination in G
; if there is no path, the function produces false
(check-member-of (find-route 'A 'G graph1) '(A E F G) '(A B E F G) '(A B F
G)) (define (find-route origination destination G)
  (cond
    [(symbol=? origination destination) (list destination)]
    [else (local ((define possible-route
                    (find-route/list (neighbors origination G) destination G)))
              (cond
                [(boolean? possible-route) false]
                [else (cons origination possible-route)]))]))

(list-of Node) Node Graph -> (list-of Node) or false
; to create a path from some node on lo-0s to D
; if there is no path, the function produces false
(check-member-of (find-route/list '(E F) 'G graph1) '(F G) '(E F G))
```

```

(define (find-route/list lon D G)
  (match lon
    [empty
     false]
    [(cons n ns)
     (local ((define possible-route (find-route n D
                                                G))) (cond
              [(boolean? possible-route) (find-route/list ns D G)]
              [else possible-route]))))

; Node Graph -> (list-of Node)
; computes the set of neighbors of node n in graph
g (check-expect (neighbors 'B graph1) '(E F))
(define (neighbors n
  g) (match g
      [(cons (list m m-neighbors) rest)
       (if (symbol=? m n)
           m-neighbors
           (neighbors n
                     rest))]
      [empty (error "node not found")]))

```

Algorithms like `find-route` are also called *backtracking algorithms* because they systematically try out alternatives and, when the locally visible alternatives are exhausted, jump back to the next point at which not all alternatives have been exhausted.

Let us now return to the question of whether the algorithm always terminates. How do we know that we are really "closer to the goal" when we move from one node to its neighbor? "Closer to the goal" in this case means that we are either closer to the target node or that we have reduced the number of alternatives that still need to be tried.

However, it is relatively easy to see that this algorithm does not always terminate, namely when the graph contains cycles. For example, if we use the graph

```

(define
  graph2 '((A
    (B E))
    (B (A E F))
    (C (D))
    (D (C))
    (E (C F))
    (F (D G))
    (G (D))))

```

and the expression

```
(find-route 'A 'G graph2)
```

we find that the evaluation does not terminate because we run in circles along the route '(A B A B A B ...)' and make no progress at all:

A call from

```
(find-route 'A 'G graph2)
```

causes a call of

```
(find-route/list '(B E) 'G graph2)
```

which in turn draws a call from

```
(find-route 'A 'G graph2)
```

as a result.

The problem is, just like in the `relative-2-absolute` example above, we don't know anything about the context in which `find-route` is called. In this example, we want to know if `find-route` is called with a start node from which we are already trying to find a route to the destination.

The accumulation parameter that we need in this example represents the list of nodes that we have already visited on the path that is being constructed. Such a parameter is easy to add. Since `find-route` and `find-route/list` are mutually recursive, the additional parameter is passed through both functions. If `find-route` is called within `find-route/list`, the invariant that `visited` always represents the set of already visited nodes is ensured by the expression `(cons origination visited)`.

```
(define (find-route origination destination G visited)
  (cond
    [(symbol=? origination destination) (list destination)]
    [else (local ((define possible-route
                    (find-route/list (neighbors origination G)
                                     destination
                                     G
                                     (cons origination visited))))
            (cond
              [(boolean? possible-route) false]
              [else (cons origination possible-route)])))]))

(define (find-route/list lon D G visited)
  (match lon
    [empty false]
    [(cons n ns)
     (local ((define possible-route (find-
route n D G visited)))
       (cond
```

```

    [(boolean? possible-route)
     (find-route/list ns D G visited)]
    [else possible-route]])))))

```

However, simply calculating and passing the accumulation parameter does not fundamentally change the program behavior. In particular, the algorithm still does not terminate with cyclic graphs.

But it is now easy to change the program so that already visited nodes are not visited again, namely by subtracting the nodes that are already contained in `visited` from the list of neighbors (`neighbors origination G`) in `find-route`.

We define a small auxiliary function for this purpose:

```

; [X] (list-of X) (list-of X) -> (list-of X)
; removes all occurrences of members of l1 from l2
(check-expect (remove-all '(a b) '(c a d b e a b
g))
              '(c d e a b
g)) (define (remove-all l1
l2)
  (match l1
    [empty
     l2]
    [(cons x xs) (remove-all xs (remove x l2))]))

```

and can use it to change `find-route` as described above:

```

Node Node Graph (list-of Node) -> (list-of Node) or false
; to create a path from origination to destination in G
; that does not use nodes in visited.
; if there is no path, the function produces false
(check-member-of (find-route 'A 'G graph1) '(A E F G) '(A B E F G) '(A B F
G)) (check-member-of (find-route 'A 'G graph2) '(A E F G) '(A B E F G) '(A B
F G)) (define (find-route origination destination G visited)
  (cond
    [(symbol=? origination destination) (list destination)]
    [else (local
             [(define possible-route
                  (find-route/list
                     (remove-all visited (neighbors origination G))
                     destination
                     G
                     (cons origination visited))])]
             (cond
              [(boolean? possible-route) false]
              [else (cons origination possible-route)])))))

```

The test with `graph2` suggests that the algorithm now also terminates on graphs with cycles. What is the argument for the general case?

According to the methodology described in §14.4 "Termination", we need to define a mapping that maps the function arguments of `find-route` to a natural

number. Let n be the number of nodes in G and m the number of nodes in `visited`. Then we define the size of the input of `find-route` as $n-m$, i.e. the number of nodes that have not yet been visited.

The difference $n-m$ is always a natural number, because by removing the already visited nodes from the neighbors (`(remove-all visited (neighbors origination G))`) in the call `(cons origination visited)`, only nodes are added from the graph that were not previously contained in `visited`. Therefore, `visited` is always a subset of the nodes of the graph; no node occurs more than once in `visited`.

Regarding the second step from §14.4 "Termination", we can state that the size $n-m$ always becomes strictly smaller by adding a new node in `(cons origination visited)`.

However, the recursive call in `find-route` is indirect via `find-route/list`. This function is structurally recursive and therefore always terminates. Furthermore, we can state that `find-route/list` always passes both G and `visited` unchanged. If `find-route/list` therefore calls `find-route`, it does so with unchanged values for G and `visited`.

Overall, we can conclude that we have shown that the recurrence depth is limited by the number of nodes of the graph and thus the algorithm terminates for all graphs.

15.3 Design of functions with accumulators

Now that we have seen two examples where an accumulator is useful, let's discuss when and how to design functions with accumulators in general.

First of all, you should only consider designing a function with an accumulator if the attempt to design the function using the standard design recipe has failed or leads to code that is too complicated or too slow. The key activities when designing a function with an accumulator are:

- 1) recognize that the function requires (or would benefit from) an accumulator, and 2) understand what exactly the accumulator represents (the *accumulator invariant*).

15.3.1 When does a function need an accumulator

We have seen two reasons why functions need an accumulator:

1. If a function is structurally recursive and the result of the recursive call is processed again by a recursive auxiliary function. The functionality of the recursive auxiliary function can often be incorporated into the main function using an accumulator and instead of nested iterations (which often lead to quadratic runtimes), a simple iteration is often sufficient.

Here is another standard example:

```

; invert : (listof X) -> (listof X)
; to construct the reverse of alox
structural recursion
(define (invert alox)
  (cond
    [(empty? alox) empty]
    [else (make-last-item (first alox) (invert (rest alox)))]))

; make-last-item : X (listof X) -> (listof X)
; to add an-x to the end of alox
structural recursion
(define (make-last-item an-x
  alox) (cond
    [(empty? alox) (list an-x)]
    [else (cons (first alox) (make-last-item
an- x (rest alox)))]))

```

2. If we have a generative recursive function, then it can be difficult to design this function so that it calculates a correct output for all inputs. In the example above, we saw that we had to remember which nodes we had already visited to ensure termination on cycles in the graph. In general, with an accumulator we can accumulate and use any knowledge about the current iteration. So if there is knowledge that is not locally available but can only be accumulated in the course of the iteration, an accumulator is the right tool.

These two options are not the only ones, but they are very common.

15.3.2 Template for functions with accumulators

If we have decided to equip a function with an accumulator, it makes sense to create a template for the function definition. This looks like this: we turn the function with accumulator into a local function defined with `local` of the function actually to be defined and then call this function in the body of the `local` expression.

In the example above, this template looks like this:

```

; invert : (listof X) -> (listof X)
; to construct the reverse of alox
(define (invert alox0)
  (local (; accumulator ...
    (define (rev alox accumulator)
      (cond
        [(empty? alox) ...]
        [else
         ... (rev (rest alox) ... ( first alox) ... accumulator)]

```

```

        ...]))))
(rev alox0 ...)))

```

15.3.3 The accumulator variant

Next, it makes sense to formulate the accumulator invariant. The accumulator invariant says what the accumulator represents in each iteration step.

To this end, we need to consider what data we want to accumulate so that the accumulator helps us to implement the function.

In the case of `invert`, it would help us if the accumulator had the list elements that we have seen so far (in reverse order), because then we can return them in the `empty` case instead of calling `make-last-item`.

We should write this accumulator invariant into the code:

```

; invert : (listof X) -> (listof X)
; to construct the reverse of alox
(define (invert alox0)
  (local (; ;; accumulator is the reversed list of all those
        items
        on alox0 that precede alox (define
          (rev alox accumulator)
            (cond
              [(empty? alox) ...]
              [else
               ... (rev (rest alox) ... ( first alox) ... accumulator)
               ...]))))
    (rev alox0 ...)))

```

15.3.4 Implementation of the accumulator variant

The next step is to ensure that the accumulator invariant is actually maintained. This means that we have to think about the initial value for the accumulator and the calculation of how to obtain the new accumulator in the recursive call. In our example, the initial accumulator is obviously `empty` and the new accumulator in the recursive call (`cons (first alox) ac- cumulator`):

```

; invert : (listof X) -> (listof X)
; to construct the reverse of alox
(define (invert alox0)
  (local (; accumulator is the reversed list of all those
        items
        on alox0 that precede alox (define
          (rev alox accumulator)
            (cond
              [(empty? alox) ...]

```

```

      [else
        ... (rev (rest alox) (cons (first alox) accumulator))
        ...]]))
    (rev alox0 empty)))

```

15.3.5 Use of the accumulator

Implementing the accumulator invariant does not change the external behavior of the function. The point of the whole operation is that we can now use the accumulator to implement the function. If we have formulated the accumulator invariant precisely, this step is typically simple:

```

; invert : (listof X) -> (listof X)
; to construct the reverse of alox
(define (invert alox0)
  (local (; accumulator is the reversed list of all those
    items
    on alox0 that precede alox (define
      (rev alox accumulator)
      (cond
        [(empty? alox)
         accumulator] [else
          (rev (rest alox) (cons (first alox) accumulator))]))
    (rev alox0 empty)))

```

16 Language support for data definitions and signatures

In a certain sense, it doesn't matter which programming language you use: Every calculation and every algorithm can be expressed in every programming language. You can simulate any programming language and any programming language concept in any other programming language.

The design techniques that we teach you can be used in all programming languages. However, programming languages differ in how well the design techniques are supported by the language. For example, if you are programming in assembler programming languages, there is nothing that corresponds to our `define-struct`; an assembler programmer must therefore work out for himself how to combine several values and arrange them in a linear memory.

On the other hand, there are also programming languages in which our design techniques are better supported than in BSL.

In this chapter, we want to talk about how languages can support one of our core concepts, namely that of data definitions and signatures. Data definitions are used to structure and classify the data in our program. Signatures are used to describe a function's expectations of its arguments and to describe the guarantees regarding the result.

In order to evaluate languages and language features in terms of their support for data definitions and signatures, we need criteria that qualitatively measure this support. We will consider the following criteria:

- Is it ensured that all values and functions are used in a way that matches the information/calculation that this value/function represents?
- At what point are errors found? In general, you want errors to occur as early as possible; preferably before the program starts, but at least at the time when the part of the program responsible for the error is executed. The worst thing is if the program itself never reports an error and instead may produce incorrect results unnoticed.
- How modular are the error messages? If an error occurs, you want to know which part of the program is "to blame". This so-called "blame assignment" is extremely important in order to be able to effectively localize and rectify errors. When errors occur, there should always be a clearly identifiable part of the program that caused the error.
- How expressive is the signature language? Can the conditions for inputs and outputs be precisely expressed in it?
- Are there any useful programs that can no longer be executed?

This applies at least to all so-called "Turing-complete" languages. Almost all common programming languages are Turing-complete.

- Is the maintenance of consistency between data definitions/signatures and program behavior supported?

In the following, we will describe the most important classes of programming languages, which differ in important points with regard to the criteria listed above.

16.1 Untypical languages

Untyped languages are characterized by the fact that any operation can be applied to all types of data, regardless of whether it makes sense or not. For example, it does not make sense to add a string and a number together.

Assembler languages are typically untyped. All types of data are represented as (32 or 64 bit) numbers. Strings, boolean values and all other data are also represented by such numerical values. If you now add two values, the numerical values are added, regardless of whether this makes sense from the point of view of what these values represent.

With regard to the first point from the list above, untyped languages therefore offer no support; it is entirely the responsibility of the programmer to ensure this property.

Errors, i.e. violations of this property, are therefore found very late in assembler programs, because the program simply continues to run, even if the data that has just been calculated is completely nonsensical.

Since there are no signatures or data definitions supported by the language, there are no restrictions on the strength of expression of the signatures/data definitions and there are no restrictions such that certain programs cannot be executed. However, there is also no support for maintaining consistency; this is the sole responsibility of the programmer.

16.2 Dynamically typed languages

In dynamically typed languages, each value is assigned a type and the runtime system represents values in such a way that the type of a value can be queried at any time during execution. A type is therefore a kind of marker for values that provides information about what kind of value it is. Typically, there are built-in ("primitive") types (which vary depending on the language) and types that can be defined by the user. The Beginning Student Language, in which we have programmed so far, is a dynamically typed language. Fixed types are, for example, Boolean (`boolean?`), Number (`number?`), String (`string?`) and Symbol (`symbol?`). New types can be defined using `define-struct`.

The dynamic types are used to ensure that only those primitive operations are applied to the values that are also defined for these values. For example, if we evaluate `(+ x y)`, the runtime system checks that `x` and `y` are actually numbers. If `x`, on the other hand, is a

is a boolean value, it is not simply interpreted as a number, unlike untyped languages.

However, this property does not apply to functions defined by the programmer himself. It makes a lot of sense to provide each function definition with a signature, as we have done, but it is not checked whether the signature is also adhered to by the function and the callers of the function.

However, as the dynamic types do not include all the information that we record in data definitions, values can still be misused. For example, it makes no sense to dedicate a temperature and a length. However, if both are represented by the type `Number`, the runtime system cannot detect this error.

Let's take a look at some examples of how and when type errors occur in dynamic type systems. Consider the following function:

```
Number (list-of Number) -> (list-of Number)
; returns the remainder of xs after first occurrence of x, or
empty otherwise
(define (rest-after x xs)
  (if (empty? xs)
      empty
      (if (= x (first xs))
          (rest xs)
          (rest-after x (rest xs)))))
```

The function returns the rest of the list after the first occurrence of the element `x` (and `empty` if the element does not occur). Here are two examples:

```
> (rest-after 5 (list 1 2 3
4)) '()
> (rest-after 2 (list 1 2 3 4))
'(3 4)
```

But what happens if we violate the signature?

```
> (rest-after 2 (list "one" "two" "three"))
=: expects a number as 2nd argument, given "one"
```

We can see that we receive a runtime error in this case. However, a runtime error does not occur (immediately) every time the signature is violated:

```
> (rest-after 2 (list 1 2 "three"
"four")) '("three" "four")
```

In this example, a list is passed that does not only contain numbers, but since the trailing elements of the list are not used, there is no runtime error.

Let us now consider the case where it is not the caller of the function but the function itself that violates the signature. In this example, the function returns a string instead of an empty list if `x` does not occur in the list.


```

Number (list-of Number) -> (list-of Number)
; returns the remainder of xs after first occurrence of x, or
empty otherwise
(define (rest-after x xs)
  (if (empty? xs)
      "not a list"
      (if (= x (first xs))
          (rest xs)
          (rest-after x (rest xs)))))

```

These examples illustrate that there is no error if we call the function with an `x` that is contained in the list.

```

> (rest-after 2 (list 1 2 3 4))
'(3 4)

```

Even if we select an `x` that is not contained in the list, the execution does not result in a runtime error.

```

> (rest-after 5 (list 1 2 3
4)) "not a list"

```

Only when we use the result and calculate with it does a runtime error occur.

```

> (cons 6 (rest-after 5 (list 1 2 3 4)))
cons: second argument must be a list, but received 6 and "not a list"

```

However, it is generally very difficult to find out who is "to blame" for this error, because the place where the error occurs may be far away from the place that causes the error. If you look at the error message, you will also see nothing that indicates that the cause of the error is to be found in the implementation of the `rest-after` function. Therefore, error messages in dynamically typed languages are not very modular.

16.3 Dynamically verified signatures and contracts

To find errors earlier and make error messages more modular, signatures and data definitions can also be defined as programs. These programs can then be used to check signatures while the program is running.

A data definition such as:

```

; a list-of-numbers is either:
; - empty
; - (cons Number list-of numbers)

```

can be represented by the following program, for example:

```

X] (list-of X) -> Boolean
; checks whether xs contains only numbers
(define (list-of-numbers? xs)
  (if (empty?
      xs) true
      (and (number? (first xs))
            (list-of-numbers? (rest xs)))))

```

These "executable" data definitions can then be used, together with the predefined predicates such as `number?`, to define a dynamically tested variant of the `rest-after` function:

```

Number (list-of Number) -> (list-of Number)
; dynamically checked version of rest-after
(define (rest-after/checked x xs)
  (if (number? x)
      (if (and (list? xs)
                (list-of-numbers? xs))
          (if (list-of-numbers? (rest-after x
                                             xs)) (rest-after x xs)
              (error "function must return list-of-numbers"))
          (error "second arg must be list-of-numbers"))
      (error "first arg must be a number")))

```

This function behaves exactly like `rest-after` as long as the function and its callers adhere to the signature:

```

> (rest-after/checked 2 (list 1 2 3 4))
'(3 4)

```

In the event of an error, however, an error message is issued much earlier and this error message is modular (it occurs at the location that is also the cause of the error).

```

> (rest-after/checked "x" (list 1 2 3 4))
first arg must be a number

```

However, programs are now also aborted with an error message which, as we have seen above, previously ran without errors:

```

> (rest-after/checked 2 (list 1 2 "three" 4))
second arg must be list-of-numbers

```

Nevertheless, it makes sense to abort these programs with an error, because in general a (then no longer modular) error will occur sooner or later. In any case, a violation of the signature is an indication of a programming error, regardless of whether it would actually lead to an error in the end.

As we can see, however, it is relatively laborious and error-prone from a programmer's point of view to check signatures and data definitions in this way. For this reason, there are some languages that support the dynamic verification of signatures and data definitions directly and conveniently.

This applies, for example, to the Racket language, which is also supported by the DrRacket environment and which, apart from minor deviations, supports the Beginning Student Language as a sub-language. In Racket, dynamic signature checks - also known as *contracts* in this context - can be defined at the boundary between modules. Modules are self-contained program units that are usually associated with files in Racket, i.e. each file is a module.

Here you can see the definition of a module that implements the `rest-after` function from above. In the `provide` clause of the module, this function is assigned a *contract*, i.e. an executable signature. We save the module in a file "heinz.rkt" to illustrate that perhaps the developer Heinz programmed this module. As you can see, Heinz has built the same error into the implementation that we looked at above.

"heinz.rkt"

```
#lang racket

(provide
  (contract-out
    [rest-after (-> number? (listof number?) (listof number?))]))
(define (rest-after x xs)
  (if (empty? xs)
      "not a list"
      (if (= x (first xs))
          (rest xs)
          (rest-after x (rest xs)))))
```

Now consider a module from the developer Elke, who wants to use this function and therefore "imports" Heinz's module via a `require` clause. This clause makes it clear that Elke's module depends on the Heinz module and wants to use its functions.

"elke.rkt"

```
#lang racket

(require "heinz.rkt")

(rest-after "x" (list 1 2 3 4))
```

However, Elke has violated the contract when calling the function and has passed a string as the first argument. If we try to execute this program, we get the following error message:

```
rest-after: contract
violation expected: number?
```

```

given: "x"
in: the 1st argument
  of (->
    number?
    (listof number?)
    (listof number?))
contract from: /Users/klaus/heinz.rkt
blaming: /Users/klaus/elke.rkt
(assuming the contract is correct)
at: /Users/klaus/heinz.rkt:3.24

```

You can see that not only the call of the function was directly recognized as faulty. The error message also clearly states who is to blame for this error, namely Elke.

So Elke corrects her mistake. However, the error that Heinz has built into the function now comes into play. However, this error is found immediately and Heinz is correctly blamed for it.

"elke.rkt"

```

#lang racket

(require "heinz.rkt")

(rest-after 5 (list 1 2 3 4))

rest-after: broke its contract
promised: "list?"
produced: "not a list"
in: the range of
  (->
    number?
    (listof number?)
    (listof number?))
contract from: /Users/klaus/heinz.rkt
blaming: /Users/klaus/heinz.rkt
(assuming the contract is correct)
at: /Users/klaus/heinz.rkt:3.24

```

As these examples illustrate, the main advantage of dynamically verified signatures and contracts is that errors are found earlier, the error messages are modular and there is a clearly identifiable "culprit" in the event of violations. Although errors are found earlier as a result, the errors are still only found during program execution. As there are generally an infinite number of different program executions for a program, you can never be sure that contract violations will not occur at runtime.

An important disadvantage of contracts is that you can only express contracts that can actually be calculated. A signature like

```
X] (list-of X) -> (list-of X)
```

requires, for example, that a predicate that checks whether a list element is an X is needed to check this contract. This predicate may have to be passed in the program and, if necessary, "looped" through the program over large "distances".

In addition, contracts can obviously only check restrictions for which the relevant information is also available as data. A data definition such as

```
A temperature is a number that is larger than -273.15.  
; interp. temperature in degrees Celsius
```

cannot be checked because we cannot look at a number to see whether it represents a temperature. However, we can use a structure definition to define a suitable tag, which can then also be checked at runtime:

```
(define-struct temperature (d))  
A temperature is: (make-temperature Number) where the number  
is larger than -273.15  
; interp. a temperature in degrees celsius
```

A pragmatic disadvantage of dynamic checks is that they can have a strong negative impact on the runtime of a program. For this reason, some languages offer the option of switching off the dynamic check.

16.4 Statically typed languages

The last variant to support signatures and data definitions by the language is the idea of a static type system. In a static type system, each program part is assigned a type *before execution* in such a way that the type of a composite program part depends only on the types of its components (so-called *compositionality*).

For example, the expression `(+ e-1 e-2)` can be assigned the type `Number` provided that `e-1` and `e-2` also have this type.

Static type systems are characterized by two important properties: 1) If a program runs through the type checker ("is well-typed"), no type error will occur in all (generally infinitely many) possible program executions. In this sense, static types are much more powerful than tests, because they can only ever check a small number of different program executions. 2) There are always programs that are rejected by the type checker, even though they could actually be executed without a type error occurring. This property is a direct consequence of the so-called "Rice's theorem", which states that non-trivial properties of the behavior of programs cannot be decided.

DrRacket supports a typed variant of Racket, Typed Racket. Here is our example from above in Typed Racket:

```
(: rest-after (-> Integer (Listof Integer) (Listof
Integer))) (define (rest-after x xs)
  (if (empty?
      xs) empty
      (if (= x (first xs))
          (rest xs)
          (rest-after x (rest xs)))))
```

As we can see, Typed Racket has a formal syntax for the signature of functions. The type system of Typed Racket is designed in such a way that the consistency of the function to the specified signature can be checked without executing the program or a test. It can therefore be checked once, "once and for all", that `rest-after` will comply with the specified signature for all parameters that satisfy the specified types.

This function can now be called as in the Beginning Student Language:

```
> (rest-after 2 (list 1 2 3 4))
- : (Listof Integer)
'(3 4)
```

However, there is one important difference: the function call is also checked for consistency with the function signature before the call is made:

```
> (rest-after "x" (list 1 2 3 4))
eval:5:0: Type Checker: type mismatch
  expected: Integer
  given: String
  in: 4
```

```
> (rest-after 2 (list 1 2 "three" 4))
eval:6:0: Type Checker: type mismatch
  expected: (Listof Integer)
  given: (List One Positive-Byte String Positive-Byte)
  in: 4
```

The fact that this check takes place before the call can be recognized by the fact that the type check of a call is successful even if the actual call would generate a runtime error.

```
> (:print-type (rest-after (/ 1 0) (list 1 2 3
4))) (listof integer)
```

The function itself is also tested without executing the function. Any violation of the specified signature will be displayed immediately.

```
(: rest-after (-> Integer (Listof Integer) (Listof Integer)))
```

```

(define (rest-after x xs)
  (if (empty? xs)
      "not a list"
      (if (= x (first xs))
          (rest xs)
          (rest-after x (rest xs)))))

```

```

eval:9:0: Type Checker: type mismatch
  expected: (Listof Integer)
  given: String
  in: xs

```

The great advantage of static type checking is that it is found before the program is executed (e.g. by the developer and not by the customer) and a well-typed program will never generate type errors. In theory, this property is often formalized in such a way that the reduction semantics for the typed language always preserves well-typedness, i.e. if a program is well-typed before the reduction, it is also well-typed after the reduction (so-called "Preservation" or "Subject Reduction" theorem) and well-typed programs that are not values can always be reduced (so-called "Progress" theorem).

The biggest disadvantage of static type checking is that there are always programs that are rejected by the type checker, even though their execution would not result in an error.

Here is a small program which is executed in BSL without type errors:

```

> (+ 1 (if (> 5 2) 1 "a"))
2

```

The same program is rejected in Typed Racket:

```

> (+ 1 (if (> 5 2) 1 "a"))
eval:10:0: Type Checker: type mismatch
  expected: Number
  given: (U One String)
  in: "a"

```

The design of type systems that can be used to check as many programs as possible is a very active branch of research in computer science.

17 Language support for algebraic data types

As we saw in §7 "Data definition by alternatives and decomposition: Algebraic Data Types" and §9.1 "Recursive Data Types", algebraic data types are essential for structuring complex data. Similar to how signatures and data definitions can be supported differently by language means (§16 "Language support for data definitions and signatures"), there are also differences in how well algebraic data types are supported by the programming language.

We will look at four different ways of expressing algebraic data types and then evaluate them.

For this purpose, we consider the following data definitions for arithmetic expressions:

Example:

```
An Expression is one of:  
; - (make-literal Number)  
; - (make-addition Expression Expression)  
; interp. abstract syntax of arithmetic expressions
```

As an "interface" for the data type, we want to consider the following set of constructors, selectors and predicates:

```
Number -> Expression  
constructs a literal expression  
(define (make-literal value) ...)
```

```
Expression -> Number  
; returns the number of a literal  
throws an error if lit is not a literal  
(define (literal-value lit) ...)
```

```
X] X -> Bool  
; returns true iff x is a literal  
(define (literal? x) ...)
```

```
Expression Expression -> Expression  
constructs an addition expression  
(define (make-addition lhs rhs) ...)
```

```
X] X -> Bool  
; returns true iff x is an addition expression  
(define (addition? x) ...)
```

```
Expression -> Expression  
; returns left hand side of an addition expression  
; throws an error if e is not an addition expression
```



```

(define (addition-lhs e) ...)

Expression -> Expression
; returns right hand side of an addition expression
; throws an error if e is not an addition expression
(define (addition-rhs e) ...)

```

We will now look at different ways in which we can represent this data type.

17.1 ADTs with lists and S-expressions

As discussed in §10.4 "S-Expressions", nested lists with numbers, strings etc. - i.e. S-Expressions - can be used as a universal data structure. Here is a realization of the functions from above based on S-expressions:

```

(define (make-literal n)
  (list 'literal n))

(define (literal-value l)
  (if (literal? l)
      (second l)
      (error 'not-a-literal)))

(define (literal? l)
  (and
   (cons? l)
   (symbol? (first l))
   (symbol=? (first l) 'literal)))

(define (make-addition e1
  e2) (list 'addition e1
  e2))

(define (addition? e)
  (and
   (cons? e)
   (symbol? (first e))
   (symbol=? (first e) 'addition)))

(define (addition-lhs e)
  (if (addition? e)
      (second e)
      (error 'not-an-addition)))

(define (addition-rhs e)
  (if (addition? e)

```

```
(third e)
(error 'not-an-addition)))
```

Functions can now be defined on the basis of this interface, such as an interpreter for the expressions:

```
(define (calc e)
  (cond [(addition? e) (+ (calc (addition-lhs e))
                          (calc (addition-rhs e)))]
        [(literal? e) (literal-value e)]))

> (make-addition
   (make-addition (make-literal 0) (make-literal 1))
   (make-literal 2))
'(addition (addition (literal 0) (literal 1)) (literal 2))

> (calc (make-addition
         (make-addition (make-literal 0) (make-literal 1))
         (make-literal 2)))
3
```

Note that the `calc` code does not differ in any way from the representation of the expressions but was merely defined on the basis of the interface.

17.2 ADTs with structure definitions

In this variant, we use structure definitions to implement the above interface. We have chosen the names so that they match those bound by `define-struct`, so we can implement the entire interface in two lines:

```
(define-struct literal (value))
(define-struct addition (lhs
                          rhs))
```

In this variant, too, we can now implement functions based on the interface. The `calc` function from the previous section works unchanged with the `define-struct` representation of algebraic data types:

```
(define (calc e)
  (cond [(addition? e) (+ (calc (addition-lhs e))
                          (calc (addition-rhs e)))]
        [(literal? e) (literal-value e)]))
```

However, with `define-struct` we now have a new, more convenient way of processing algebraic data types, namely pattern matching (§13 "Pattern Matching"):

```

(define (calc e)
  (match e
    [(addition e1 e2) (+ (calc e1) (calc e2))]
    [(literal x) x]))

> (make-addition
   (make-addition (make-literal 0) (make-literal 1))
   (make-literal 2))
(addition (addition (literal 0) (literal 1)) (literal 2))

> (calc (make-addition
         (make-addition (make-literal 0) (make-literal 1))
         (make-literal 2)))
3

```

17.3 ADTs with define-type

The 2http/abstraction teachpack offers a new way of representing algebraic data types with the `define-type` construct. In contrast to `define-struct`, `define-type` offers direct support for sum types, meaning that the sum type `expression` with its various alternatives can be defined directly. Another important difference to `define-type` is that a predicate function is specified for each field of an alternative, which defines which values are permitted for this field. These predicate functions are a form of dynamically checked contracts (see §16.3 "Dynamically checked signatures and contracts").

Switch to the
"Intermediate
Student Language"
to use `define-`
`type`

```

(define-type Expression
  (literal (value number?)))
  (addition (left Expression?) (right Expression?)))

```

Analogous to `define-struct` (§6.2 "Structure definitions"), `define-type` defines constructor, selector and predicate functions for each alternative, so that we can define values of this type in the same way:

```

> (make-addition
   (make-addition (make-literal 0) (make-literal 1))
   (make-literal 2))
(addition (addition (literal 0) (literal 1)) (literal 2))

```

However, when the constructors are called, the system checks whether the fields fulfill the associated predicate function. The following call, which could be executed with the `define-struct` variant, now fails at runtime:

```

> (make-addition 0 1)

```

*make-addition: expects an undefined or an Expression, given
 0 in: the 1st argument of*

```
(-
  (or/c      undefined?
    Expression?)      (or/c
    undefined?      Expression?)
  addition?)
```

*contract from: make-
 addition blaming: use
 (assuming the contract is correct)
 at: eval:13.0*

In addition to `define-type`, there is also an extension of `match`, namely `type-case`. With the help of `type-case`, the `calc` function can now be defined as follows:

```
(define (calc e)
  (type-case Expression e
    [literal (value) value]
    [addition (e1 e2) (+ (calc e1) (calc e2))]))
```

```
> (calc (make-addition
          (make-addition (make-literal 0) (make-literal 1))
          (make-literal 2)))
3
```

The most important difference between `match` and `type-case` is that the type `expression`, on which we want to make a case distinction, explicitly with is specified. This enables the DrRacket environment to check whether all cases have been covered *before* the program is executed. For example, the following definition is rejected before execution with a corresponding error message.

```
> (define (calc2 e)
  (type-case expression e
    [addition (e1 e2) (- (calc2 e1) (calc2 e2))]))
```

type-case: syntax error; probable cause: you did not include a case for the literal variant, or no else-branch was present

The price for this completeness check is that `type-case` only allows very limited pattern matching. For example, it is not permitted to use literals, nested patterns or non-linear pattern matching. In general, the clauses of `type-case` always have the form `((variant (name-1 ... name-n) body-expression))`, whereby the names `name-1` to `name-n` can be used in `body-expression`.