DRAKON-Erlang: Visual Functional Programming

By Stepan Mitkin

Abstract

Functional programming is based on useful and practical ideas. Unfortunately, there is a problem with how functional programs look. They are often hard to read.

Improving the visual appearance of functional programs can make them easier to understand. One way of doing it is to combine some existing functional language with a graphic notation.

DRAKON-Erlang is an attempt to do so. This hybrid language can be described as Erlang that uses DRAKON for flow control.

Each of these two technologies have been successful on their own.

DRAKON was developed within the Russian space program. It was used in Buran, Sea Launch and other space projects. The strong point of DRAKON is that it makes algorithms easier to understand by relying on ergonomics.

Erlang is one of the most widely used functional languages. It started its path in telecom and later spread out to many other industries. Erlang is well known for its simplicity, reliability and built-in support for concurrent programming.

DRAKON Editor provides an implementation of DRAKON-Erlang.

Table of Contents

Introduction

A close look at functional programming

Functional programming is gaining popularity nowadays. It is being more and more widely accepted as "mainstream". Why? Because it feels "esoteric" or "mathematical"? Maybe. But apparently, the functional programming style provides certain practical benefits. Here are a few of them:

- **A better separation of concerns.** Function composition and closures are an easy way to draw healthy boundaries within a program. They glue together different algorithms that can be developed and tested separately. They help the programmer keep different ideas about the program apart.
- **No need to keep track of changing values of variables.** In conventional languages, the same variable may hold different values at different moments of time. When looking at any given part of a program, the programmer must deduce the expected value of the variable from the context. And it must be done for all occurrences of a variable, because its value can change. Tracking the changes requires significant mental effort. This effort is not needed with functional languages. They guarantee that there is only one immutable value behind each symbol.
- **Each algorithm has an explicit input and output.** Functions take arguments by value. Returning the result of computation is the only way how a function can change the outside world. As a result, a function can be seen as a system that has a clearly defined input and output. This separation of the input and output has an extremely positive effect on ergonomics. All effect of a function invocation is recorded in a very visual way. There is no "something" that get changed "somewhere". Visualizing this "something" is hard work with conventional languages.

In spite of the above, functional programming is still not the primary way of how people write programs. Let us look at some of the reasons for that and find possible ways to mitigate those reasons.

- **Recursion is not immediately visible.** In conventional languages, loops are implemented with special keywords. These keywords are easy to spot. The functional way to do loops is recursion. The problem with recursion is that recursion takes some effort to recognize it in source code. The user must compare the name of the current function to the name of the function being called. It gets even worse with indirect recursion. How to fix that? Let us add a new rule to the programming style: each recursive call must be marked with a special

comment "recursion". Of course, a good IDE should detect recursion automatically and highlight it.

- **Recursion allows for only one loop per algorithm.** This is an inherent disadvantage of functional programming. In contrast, algorithms in conventional languages may have several loops. Luckily, many loops can be represented as calls to standard looping functions like **filter**, **map**, **fold** and others.
- **Tail recursion is a trick.** Some recursive algorithms must be implemented with tail recursion. Otherwise they get too slow. But tail recursion is not intuitive. It requires very intensive thinking to do it right and adds complexity to the original algorithm. Tail recursion is an evil remnant of ancient compilers. Hopefully, developments in compiler technology will take care of that.
- **Non-trivial syntax.** This issue is not as easy as it seems. There are a lot of people who think that it is not a problem at all. They find the syntax of popular functional languages convenient. But there are also many people who consider functional languages "cryptic". Let us admit: it is way too easy to make a "write-only" functional program. A program that would be really difficult to read and understand.

Good idea, bad presentation

The ideas behind functional programming are simple, powerful and practically useful. But the presentation layer is of a very low quality.

It is the visible appearance of functional programs that makes them not attractive.

But isn't the essence more important than the form? Who cares about the wrapping if the gift inside is good?

With information technology, we cannot take the gift and throw out the wrapping. All kinds of information, no matter how abstract, must be represented in a specific form. The choice of form is extremely important because it greatly affects productivity. Productivity in software projects is closely tied to the amount of effort required for understanding. The harder it is to understand a program, the more effort is needed to develop and maintain it.

This is especially relevant to projects that have more than one developer. Any given piece of code gets written once, but is read many times by several people. Whatever makes understanding easier boosts productivity.

In addition, it takes less time to find bugs in a program that is easy to understand. That is why easy understanding also improves quality.

Representing algorithms

Any non-trivial program consists of 3 parts:

1. algorithm;
2. data structure;
3. glue code.

Algorithm is the core of software. It is the algorithm that produces the desired output of a program. Therefore, **if the algorithm is easy to understand, the whole program is easy to maintain, extend and document.**

Unfortunately, the traditional way of recording algorithms is wrong. This is a major problem of

programming in general, not only functional programming. Algorithms in most of the modern programs are written using text. This text is called "the source code".

Usually, source code is indented in a special way and highlighted with different colors in the editor. But it is plain text. Text is bad because humans are not very good at understanding text. Their eyes and brains have been optimized for seeing images during millions of years. Text is a recent invention. The human biological hardware is not natively compatible with text.

If the algorithm is presented as an picture instead of text, it can be easier to understand.

There are a lot of ways to draw an algorithm. Choosing the right one is critically important. A bad picture can be worse than text. A good picture will leverage the ability of the human being to perceive information simultaneously.

Flowcharts are a popular graphical notation for representing algorithms. Many developers do not like flowcharts. The reason is that complex algorithms tend to end up in badly cluttered flowcharts. Those can be harder to figure out than text-based programs. As a result, graphical programming is often dismissed as an unsuccessful experiment.

But the problem is not in graphical programming itself. The problem is that flowcharts as a graphical language are not good enough. The good news is, it is possible to improve this language.

DRAKON provides such improvement. It offers several simple, but effective rules that cardinally increase readability of a diagram. DRAKON is a visual language that can be described as flowcharts optimized for ergonomics. The creators of DRAKON paid great attention to human visual habits. Every little detail of DRAKON is aimed at ensuring fast and easy understanding.

That is why DRAKON is an excellent candidate for visualization of a functional programming language. The clarity of DRAKON is something that functional programming can greatly benefit from.

DRAKON-Erlang

Let us consider the hybrid language **DRAKON-Erlang**.

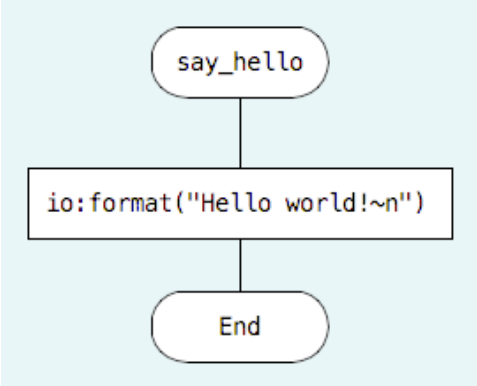
- In DRAKON-Erlang, the flow control is performed by the graphical icons of DRAKON.
- Erlang expressions and function calls are placed inside icons.

The choice of Erlang is not arbitrary. It is not only an excellent functional programming language. The main advantage of Erlang is that it was designed from the beginning to make concurrent programming easy. And nowadays, all programming is concurrent. Single-core processors become increasingly hard to find. In addition, Erlang makes distributed computing and clustering a breeze.

The combination of a solid functional programming foundation, built-in concurrency and visual clarity make DRAKON-Erlang a very promising technology.

Sequence of Actions

The simplest form of algorithm is a sequence of actions. Let us start with the traditional "Hello world!" program. It is a sequence that contains only one action. This action outputs a pre-defined string.

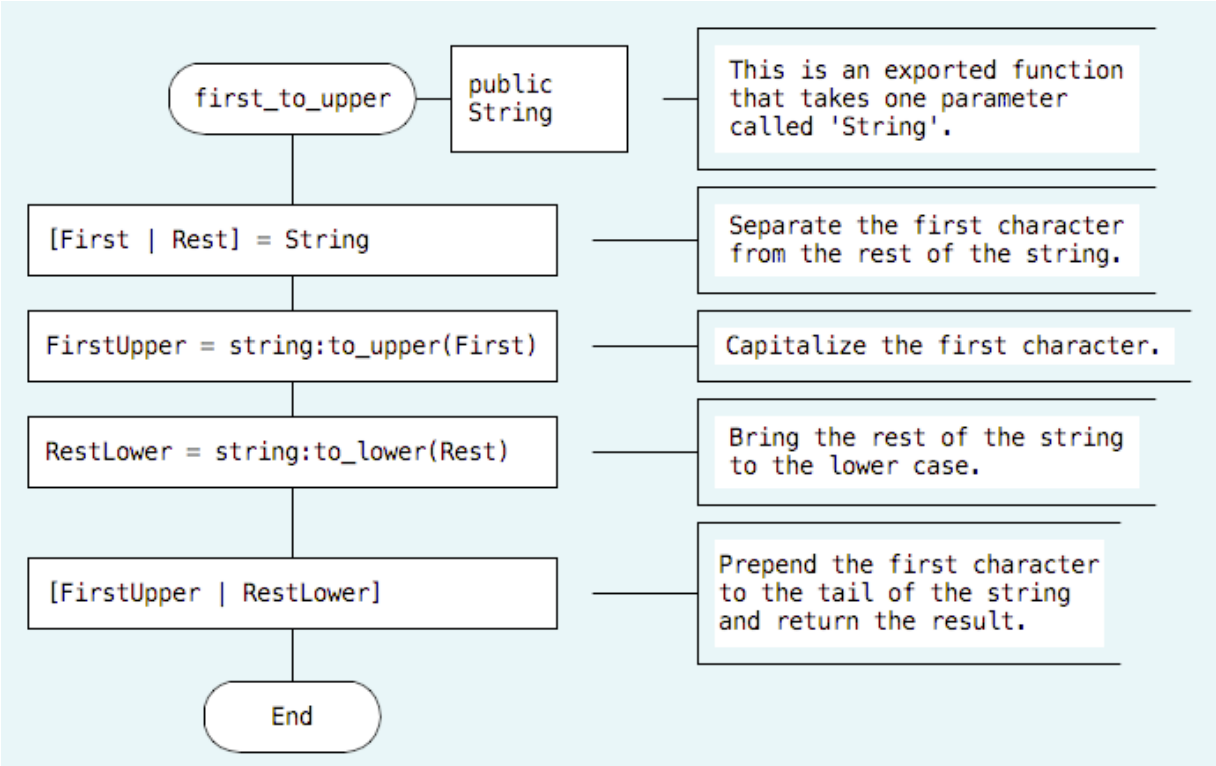


This diagram consists of three icons:

- The name of the function is placed at the top in the "Header" icon.
- The rectangle is an "Action" icon. An "Action" icon is an order to do something.
- The diagram ends with an "End" icon.

There must be only one "End" icon in any diagram. The text inside the "End" diagram must be exactly "End".

More interesting algorithms have more actions. Here is an algorithm that makes the first character of a string a capital letter.



This trivial diagram illustrates a few important principles of DRAKON.

- **Clarity is above all.** Making an algorithm appear shorter must not sacrifice readability.

  Some may think that this function should be written in a shorter way. For example:

  ```
  first_to_upper([First | Rest]) ->
    [string:to_upper(First) | string:to_lower(Rest)].
  ```

  Indeed, this form is more concise. But it requires the reader to unpack the code in his head. The goal of DRAKON is to eliminate any unnecessary mental work. This is why each operation deserves its own line on the diagram.

- **The next icon is always below the current one.** In DRAKON, consecutive icons are placed on a vertical line. The order of processing the icons is always from top to bottom. This order is not arbitrary:
  - Moving down is very natural in our material world with gravity. Dropping things does not require any effort.
  - The descending order is very familiar because it is used in text. The next line goes beneath the current one.
- **The vertical line must always be straight and not have any bends and turns.**

  Having such a convention reaches two goals at once.
  - It develops a habit for the reader. Diagrams that follow the reader's habit are predictable and fast to read.
  - Arrows can be replaced with plain lines. Arrow heads create visual noise. They clutter the diagram with redundant visual details.

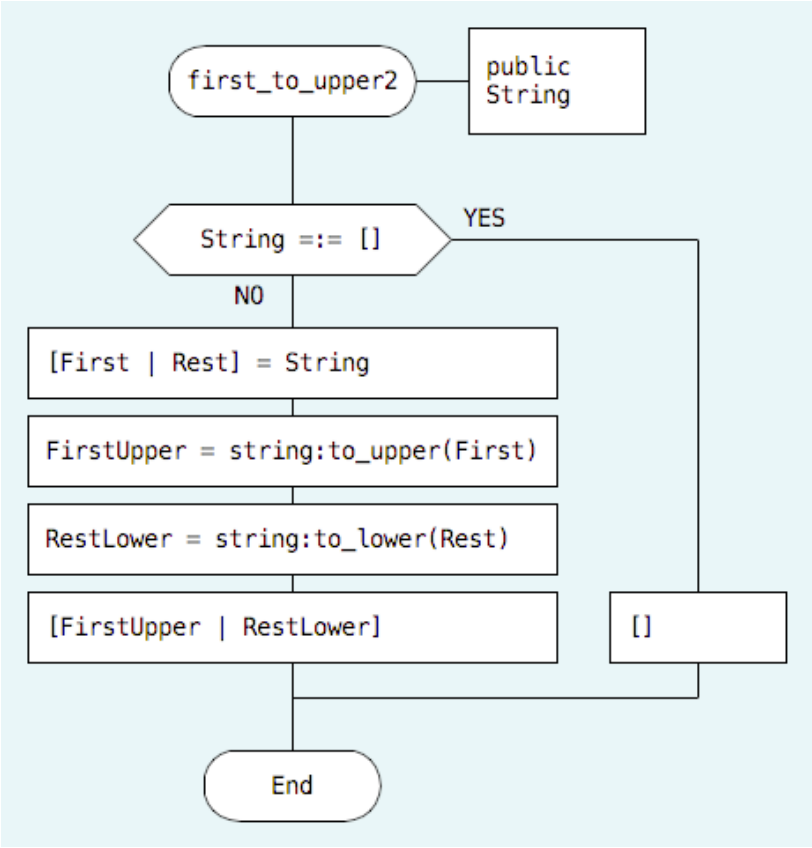
  DRAKON gets rid of anything that does not have an immediate practical value. Lines always lead down, so there is no point in specifying direction. There is only one usage of arrows in DRAKON. An arrow always goes up and describes a loop. But there are no loops in Erlang, so we will not discuss them here.

Sequential programming is the main building block of algorithms. But it does not reveal the full power of DRAKON.

Branching

There is a bug in the *first_to_upper* function mentioned before. If the input is an empty string, that function will crash.

Lets us add a check for that special case.



The "If" icon chooses the code path based on a question which can be answered **yes** or **no**. The "If" icon has two exits. The first exit comes out of the bottom and goes down. The second one comes out of the right side of the icon and goes to the right.

Placing an exit on the left side is not allowed. This is a very strict rule. The point of having this rule is again to make use of human habits. When the exits are at the expected locations, the user does not need to look around and search for what happens next. He just follows the usual pattern. The user can concentrate on the algorithm itself, instead of the diagram that represents it.

The exits are marked with labels **YES** and **NO**:

- The labels can be swapped.
- If **YES** goes to the right, **NO** goes down.
- If **NO** goes to the right, **YES** goes down.

Note that the words **yes** and **no** are used instead of **TRUE** and **FALSE**. **TRUE** and **FALSE** sound scientific, but they are not convenient. **Yes** and **no** are much more natural, because these are the words that children learn very early.

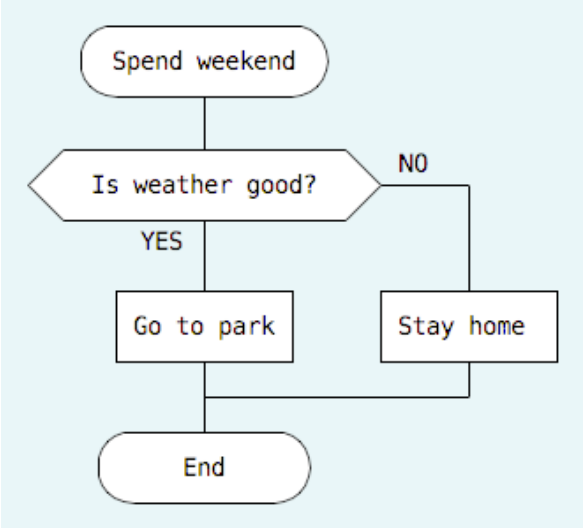
The shape of the "If" icon has also been optimized for better ergonomics. On traditional flowcharts, "If" is drawn as a diamond. DRAKON cuts off the top and the bottom vertexes of the diamond. This way, the "If" icon occupies less space and can fit more text inside.

The exits of an "If" icon are not equal. The less desired outcome of the "If" icon should go to the

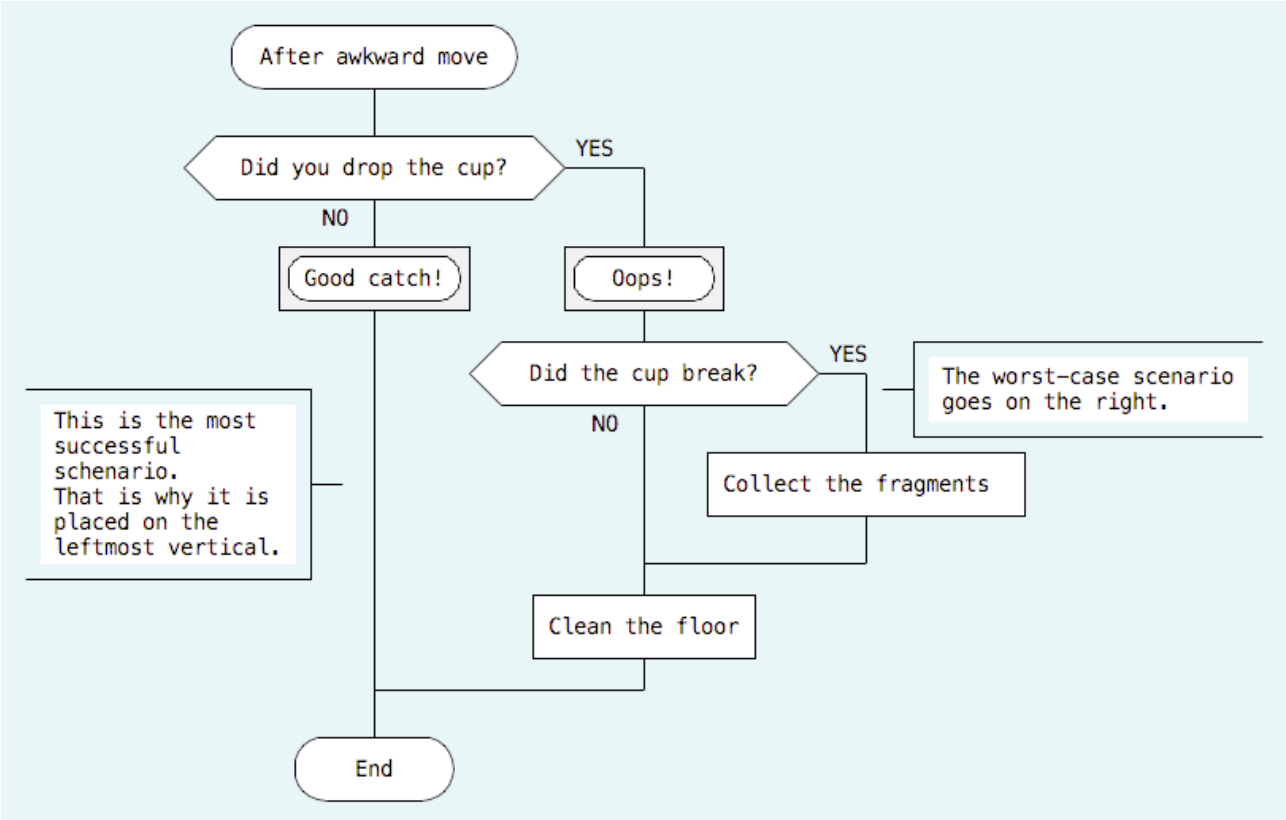
right. The exit that leads to a greater success should go down.

Because of this, **the happy path through the algorithm remains on the leftmost vertical**.

If there are several "If" icons, there can be more than two verticals. The happy path should still be on the leftmost vertical. Unpleasant situations and error handling should be located further to the right on the diagram.



Rule: The further to the right — the worse.



Let us look at the modified version of the *first_to_upper* function again. An empty string is a special case. This is why this situation is handled on the right vertical.

The ability to visually distinguish between good paths and not so good ones is a unique feature of DRAKON. It adds an additional dimension to the diagram. The reader can immediately see how badly things have gone in a particular icon depending on the position of that icon on the diagram.

Another advantage is that if the reader is not interested in error handling, he can just follow the happy path and skip everything else.

There is another important rule related to horizontal lines produced by "If" icons.

No icons are allowed on horizontal lines.

This rule ensures that:

- The next icon is always below the current one.
- Each icon has exactly one entry.
- The entry is always at the top of the icon.

As a result, diagrams receive a consistent look.

Advanced Branching with Switch

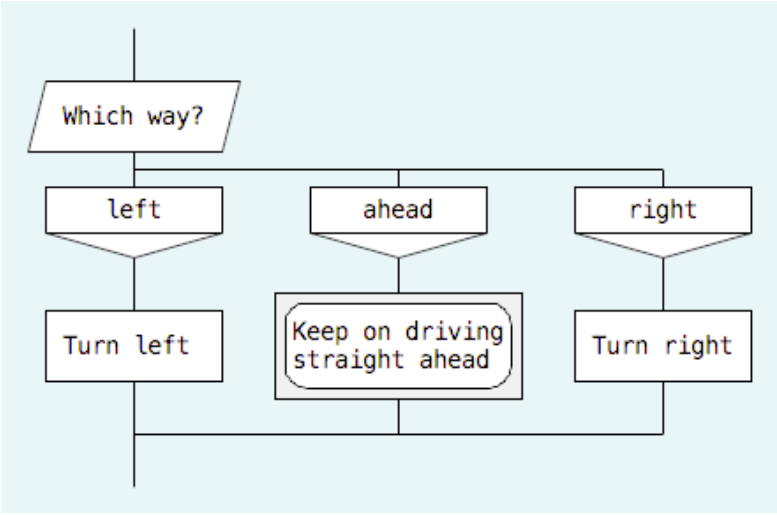
The "If" icon is good for questions that can be answered "yes" or "no". For example:

- Is X greater than Y?
- Did you party last night?
- Is it Tuesday today?

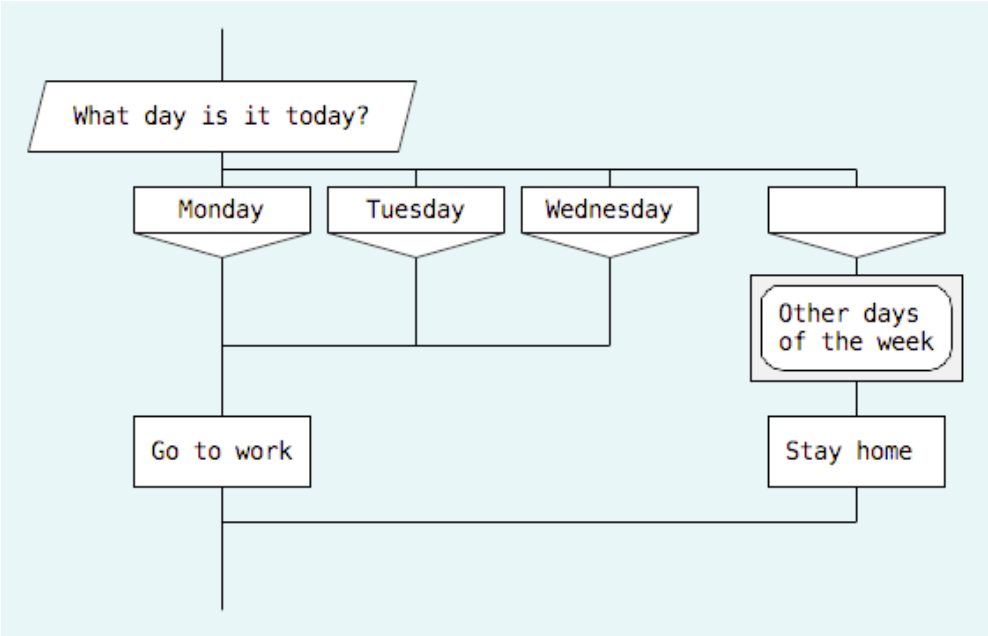
But not all questions are so simple. There are questions that may have multiple answers. For example:

- What is the value of X? 10? 20? 30? Greater than 30?
- When did you party last? Last night? Two days ago? A week ago? Ages ago?
- Which day of the week is it today? Monday? Tuesday? Wednesday? Some other day?

The common thing about such questions is that they split one algorithm path into several. DRAKON handles multiple choice questions with the "Switch" construct. The "Switch" construct consists of one "Switch" icons and several "Case" icons.



The rightmost "Case" icon can be empty. An empty "Case" icon means "all other values". It is similar to the "default" clause of "switch" statements in procedural languages.



"The further to the right — the worse" rule also applies to the "Switch" construct.

If some branches are worse than others, the "Case" icons should be sorted left-to-right.

The "Case" icon that corresponds to the luckiest scenario should be at the left.

The worst "Case" will be the rightmost one. This is similar to how the "If" exits are arranged.

Regardless of the way of branching, the happy path goes on the main vertical and error handling is placed on the right side.

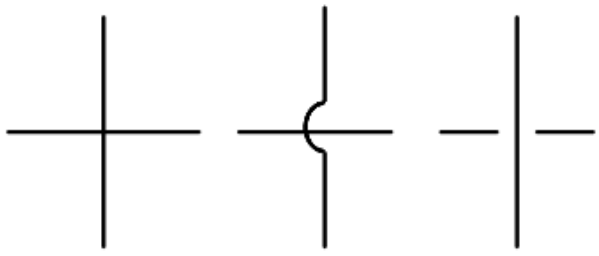
Sometimes all branches are equally good. In this situation the "Case" icons should be sorted by the ascending order. Then the rule becomes "the further to the right — the greater". Depending on the problem, other rules could be devised.

The key thing to remember is to sort the "Case" icons in some monotonous way.

No Intersections

Line intersections kill readability. They quickly turn a diagram into an entangled clutter. A diagram with intersections takes much more time to understand.

There have been attempts to make intersections look less ugly.



All these attempts failed.

The truth behind intersections is that they produce diagrams that are not planar. Those non-planar diagrams cannot be drawn on a sheet of paper without introducing the third dimension. In that case the reader must visualize the depth of the diagram. That leads to additional work imposed on the brain.

This is why **DRAKON forbids intersections**.

Silhouette

Primitive and silhouette

DRAKON has two types of diagrams:

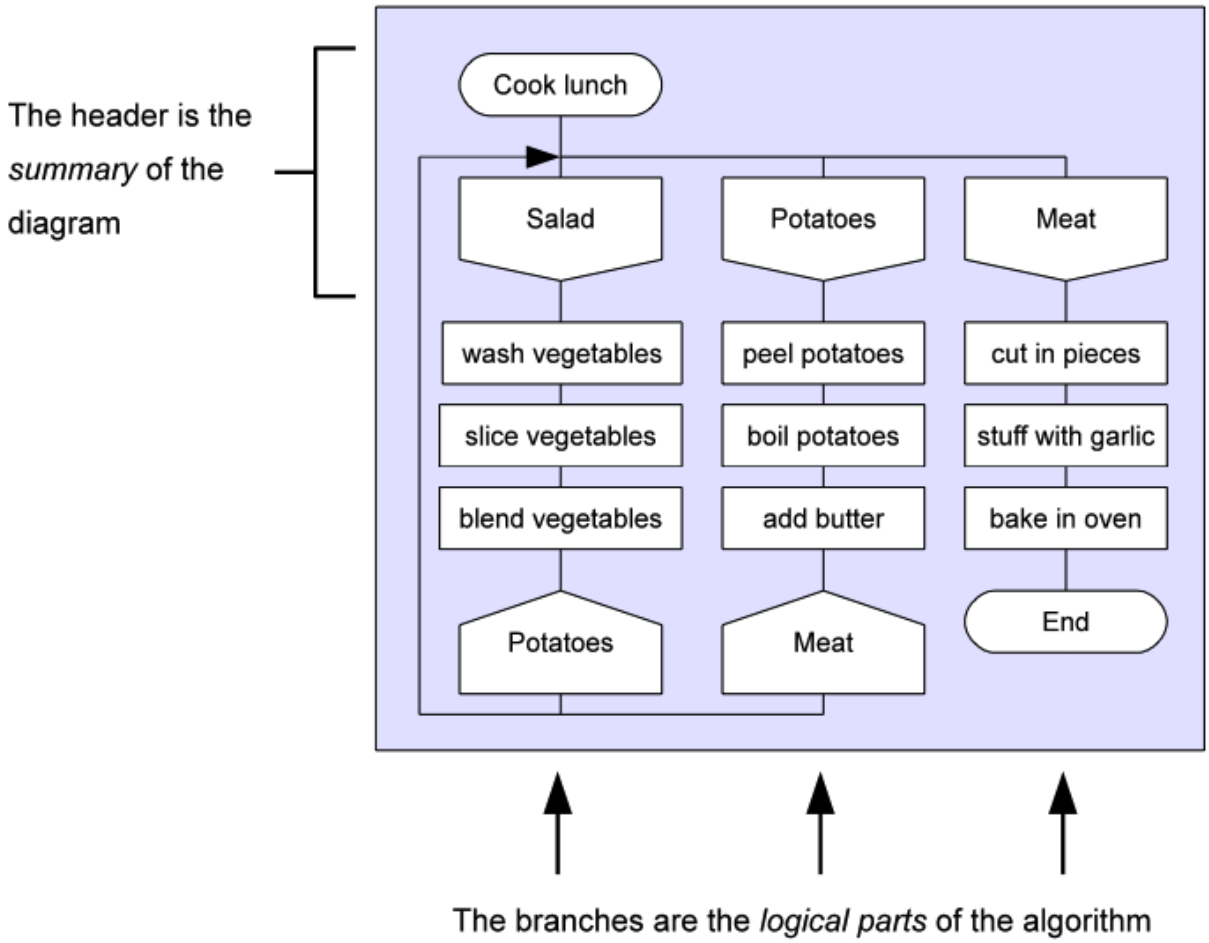
1. The diagrams that have been introduced before are called **primitives**. Primitives are good for very basic algorithms and for educational purposes.
2. The full power of DRAKON, however, comes with another type of diagrams. That type is called **silhouette**.

Silhouette is the recommended way of drawing diagrams with DRAKON.

Branches

Silhouette combines several simple diagrams to build a complex one. The goal is to partition a big algorithm into logical parts. These parts are called "branches".

Each branch has a name. A branch ends with an "Address" icon that points to the next branch. The rightmost branch has an "End" icon instead of an "Address" at the bottom. There can be only one "End" in an algorithm.



The header of the diagram

Silhouette answers the three questions of the king:

1. What is the name of the task?
2. Which parts does the task consist of?
3. What are the names of the parts?

The "Header" icon contains the name of the diagram and answers **the first question**.

The branches answer **the second question**.

The headers of the branches give the answer **to the third question**.

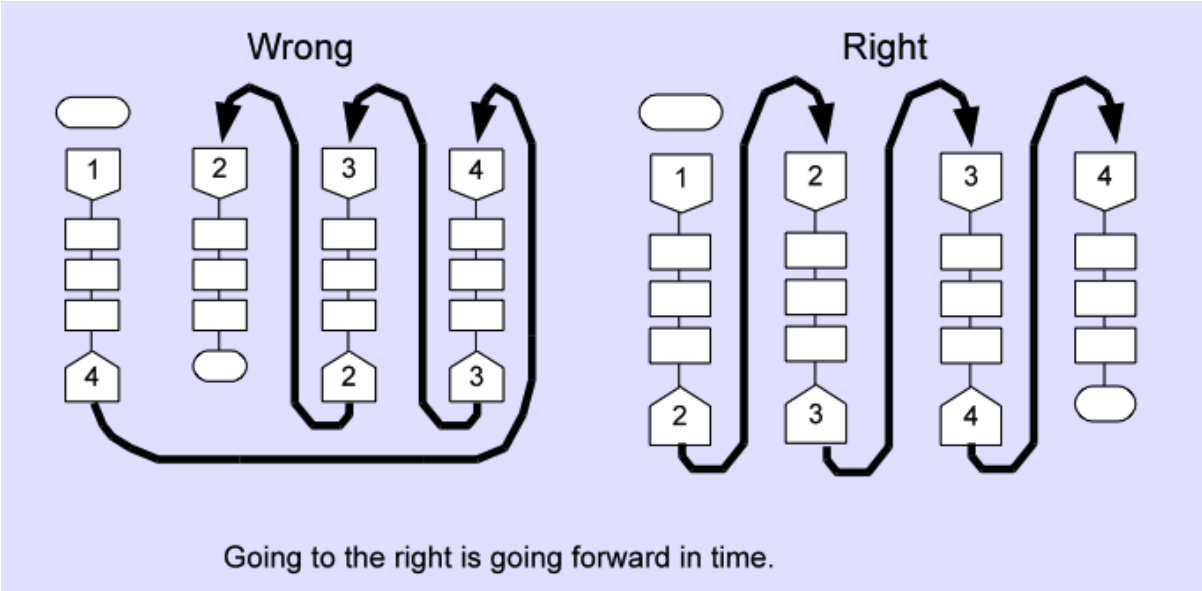
The diagram name is always located in the upper-left corner of the diagram. The branch headers are just under the diagram name. They are aligned to form a horizontal line.

This convention allows the reader to grasp the general ideas of the algorithm without delving into the details.

The order of branches

Theoretically, the branches could follow each other in any order. The next branch to execute is determined by the name inside the "Address" icon of the previous branch.

Nevertheless, DRAKON rules recommend to arrange the branches left-to-right. The leftmost branch should be the first one. The rightmost branch should be the last one. The branches in between should be sorted in such a way that the next branch should always be to the right from the current one. Some branches can be skipped.



In procedural languages, it is possible to jump back to a branch which is to the left from the current branch. This is especially convenient for programming of finite automata.

DRAKON's silhouette is a very good way of representing a state machine:

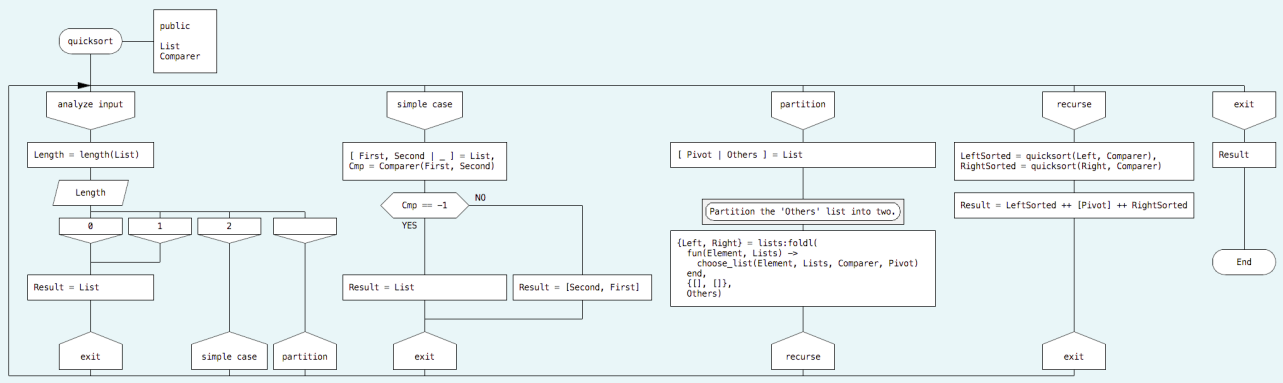
- Each branch becomes a state.
- The "Address" icons define the allowed state transitions.

But functional languages usually do not have loops. Therefore jumping left on a DRAKON-Erlang silhouette is not allowed.

Sorting "Address" icons

When a branch has several "Address" icons, it is recommended to sort them so that they appear in the same order as the branches they point to. This guideline should not apply when such ordering breaks "The further to the right — the worse" rule. Routing the happy path through the main vertical and moving the error handling to the right is more important.

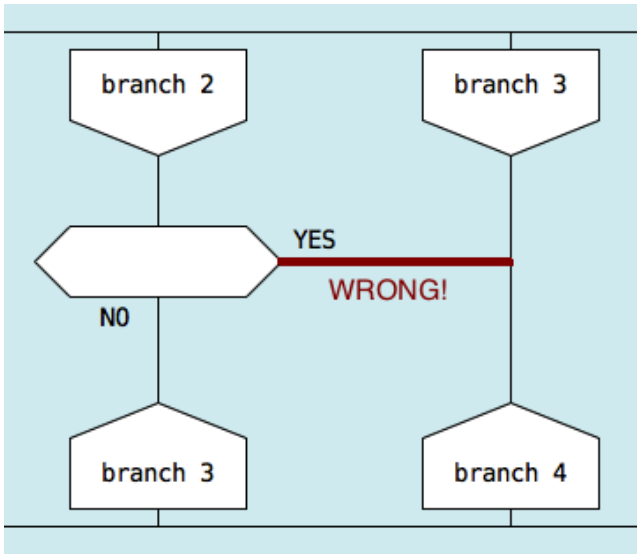
A silhouette that implements the quicksort algorithm:



The main verticals of the branches constitute the happy path of the whole silhouette. The reader may follow only the main verticals if he is not interested in the processing of error situations.

Do not connect the branches

Rule: never connect two branches of a silhouette.



The connecting lines

The lines and the arrow that surround the branches in a silhouette are technically not necessary. But their presence gives the diagram a connected and finished look. The diagram is perceived not as a disjoint set of parts, but as a whole picture that has a meaning.

Boolean Expressions

There are three ways to express boolean logic in Erlang:

1. **Boolean operators: and, or, xor, not.**

```
more_than_10_times(Left, Right) ->
    if (Right =/= 0) and (Left / Right > 10)
        -> true
    ; true
        -> false
    end.
```

2. **Short-circuit boolean operators: andalso, orelse.** These operators evaluate their second operand only when necessary.

```
more_than_10_times_ex(Left, Right) ->
    if (Right =/= 0) andalso (Left / Right > 10)
        -> true
    ; true
        -> false
    end.
```

3. **Guards.** Guards may contain comma (,) and semicolon (;) operators. The comma operator is somewhat similar to **andalso**. Semicolon resembles the **orelse** operator.

```
more_than_10_times_g(Left, Right) when Right =/= 0, Left / Right > 10
    -> true;
more_than_10_times_g(_, _)
    -> false.
```

There is one common problem about all of these forms. They get totally unreadable when the logic expression gets even moderately complex. They require a lot of hard thinking to figure them out.

Another issue is short-circuit evaluation. Although efficient, the short-circuit operators introduce hidden paths into the algorithm. Depending on the data being processed, guards and short-circuit operators may skip the second operand. The burden of determining which operand is skipped and when fully lies on the reader of the program.

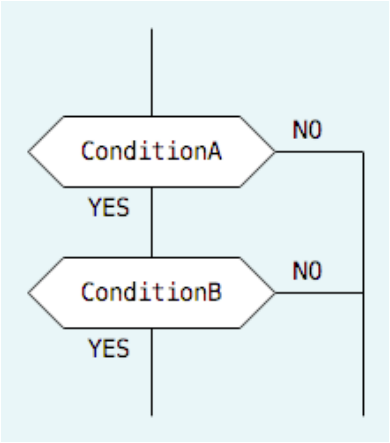
DRAKON has a way to deal with these problems. The standard DRAKON's "If" icons can be arranged to form **visual logic formulas**. Visual logic formulas can represent any logic expression in an easily comprehensible way.

Here is the visual formula for the short-circuit **AND** expression.

Text formula for **AND**:

```
ConditionA andalso ConditionB
```

Visual formula for **AND**:

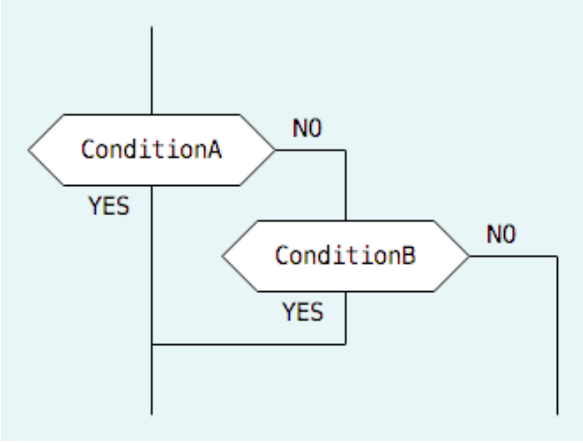


Here is the visual formula for the short-circuit **OR** expression.

Text formula for **OR**:

```
ConditionA orelse ConditionB
```

Visual formula for **OR**:



As you can see, all 4 possible combinations of the operands are present on the diagram. The outcome for each of them can be traced with a finger.

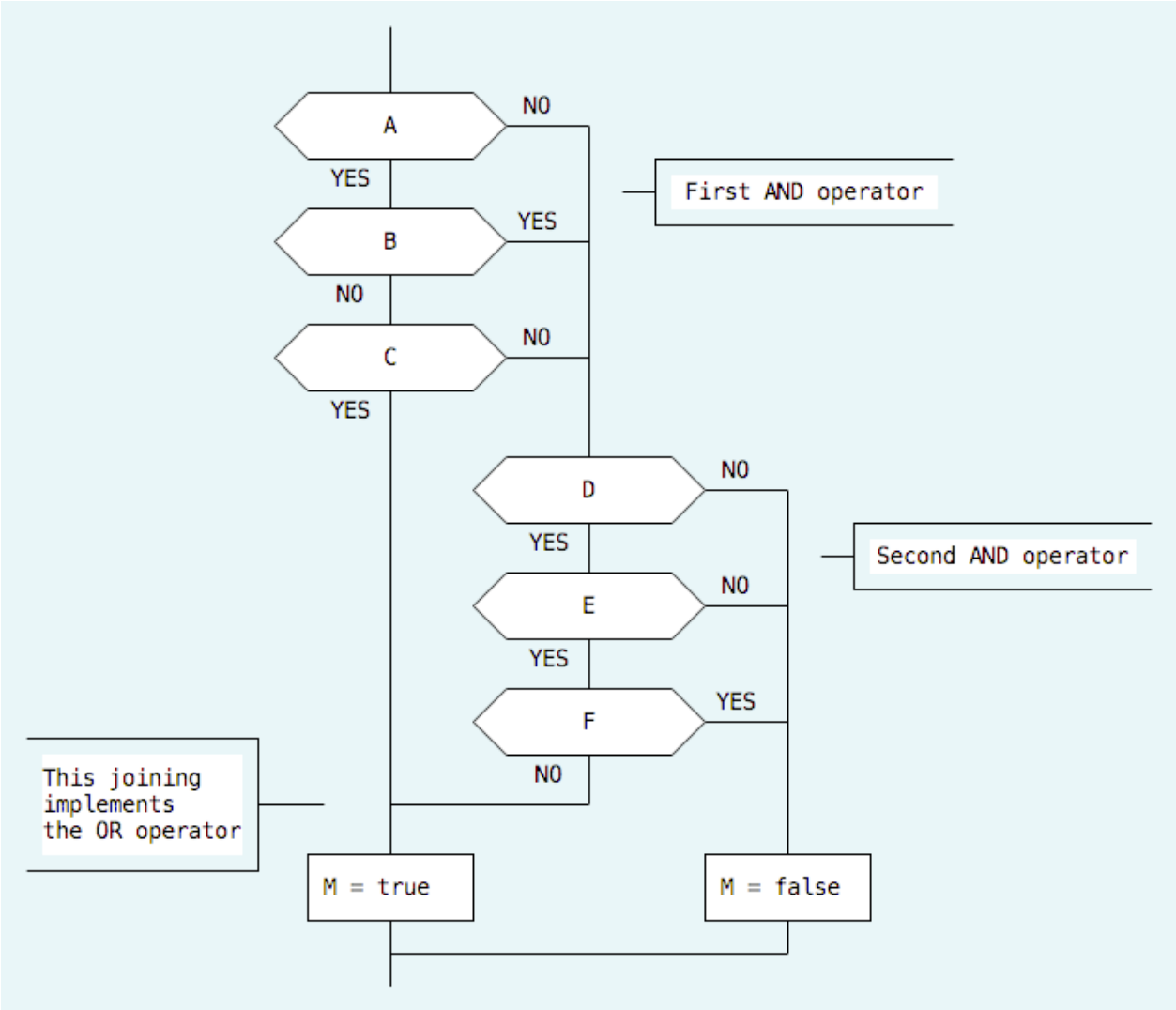
If we want to evaluate both of the operands before invoking the AND operator, it should be done in an explicit way.

Now let us compare a text-based and a visual formula for a more sophisticated logic expression.

Text formula:

```
M = (A and not B and C) or (D and E and not F)
```

Visual formula:



The text-based formula definitely occupies less space. But it is the reader's job to decompress it. The complexity is still there, but it is hidden.

The key point of DRAKON is that it does not hide anything. All quirks of the algorithm are visible. Hence the term 'visual programming'.

Note that **no additional construct is required to express negation**. DRAKON does not have a NOT operator because it is not needed. If you want to invert the outcome of an "If" icon, just swap the YES and NO labels.

With DRAKON, there is no need to remember which logic operators force evaluation of their arguments and which do not. Every path through the algorithm is clearly visible. **The recommended way of doing logic with DRAKON is visual formulas.**

Pattern Matching

Pattern matching is considered a very useful feature in programming languages. What are the main uses of pattern matching in Erlang?

1. Taking to pieces a complex value.
2. Asserting assumptions.
3. Branching.

Taking to pieces a complex value. Pattern matching is the primary way to get the elements of lists and tuples in Erlang. The following line puts the head of *SomeList* into *First*. The tail of *SomeList* goes to *Rest*.

```
[First | Rest] = SomeList.
```

Another example. Let us assume that the *Person* variable has this value:

```
Person = {person, "John", "Malkovich", {date, 1953, 12, 9}}.
```

The following statement extracts the first name, the second name and the year of birth from *Person*.

```
{_, FirstName, SecondName, {_, Year, _, _}} = Person.
```

FirstName becomes "John". *SecondName* becomes "Malkovich". *Year* becomes 1953.

Asserting assumptions. An expression with pattern matching will throw an exception if the expected value is not equal to the actual one. Erlang is very good at dealing with exceptions and recovery. That is why asserting is especially helpful in Erlang.

```
{person, FirstName, "Malkovich", {date, Year, _, _}} = Person.
```

The statement above has two effects:

- *FirstName* and *Year* variables get bound with values.
- The structure and the content of the *Person* variable is verified: the record type must be **person**, the second name must be "Malkovich", the record type for the birth date must be **date**.

Branching. Pattern matching can work as an **if** or **switch** statement in disguise.

```
feed({person, FirstName, SecondName, _}) ->
    give_money(FirstName, SecondName);
feed({animal, Name}) ->
    give_food(Name).
```

Here, the *feed* function checks whether its argument is a person or an animal and chooses what to do next based on that.

There is one more way to use pattern matching for branching.

```
feed(Creature) ->
    case Creature of
        {person, FirstName, SecondName, _} ->
            give_money(FirstName, SecondName);
        {animal, Name} ->
            give_food(Name)
    end.
```

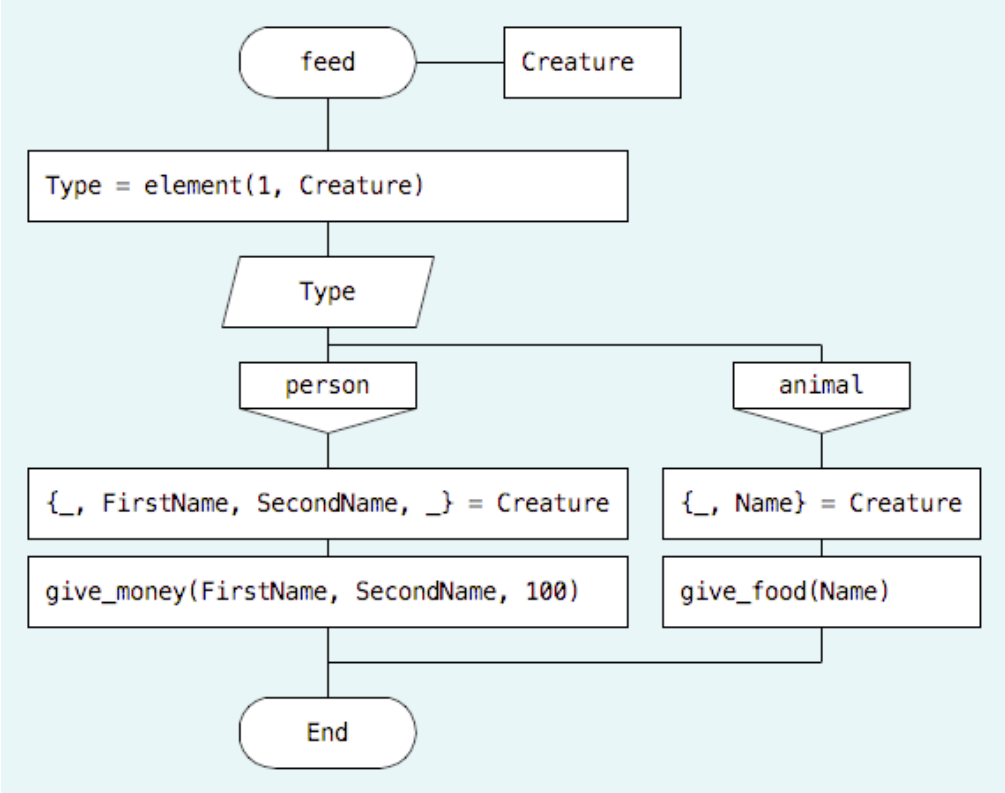
How does pattern matching work in DRAKON-Erlang?

The first two application of this technique work exactly as they do in the standard Erlang.

- Pattern matching disassembles tuples and lists as usual.
- The structure and content of data items is checked exactly like in text-based Erlang.

But branching is different. **DRAKON-Erlang does not use pattern matching for branching.** Selecting an algorithm path based on a condition is done either with an "If" or a "Switch" icon. Choosing "If" or "Switch" depends on the nature of the condition. If the condition can be either **true** or **false**, then the "If" icon is used. If the condition is a multiple-choice question, there goes the "Switch" icon.

In the following example pattern matching is used only for element extraction and for assertion. Branching is done explicitly with a "Switch" construct.



Why is this limitation present in DRAKON-Erlang?

- **A picture is better then text.** Branching is central to algorithms. It is also the main source of complexity. DRAKON always strives to make important and complex things immediately visible. This is why branching is represented with visual icons. The traditional text-based forms of branching may look shorter. But they are harder to read. DRAKON's philosophy is against making branching implicit and disguised.
- **Simplicity.** DRAKON-Erlang is a simpler language than the traditional Erlang. Erlang has many ways to do branching:
  - **if** expression;
  - **case** expression;
  - patterns inside function arguments;
  - guards in function headers.

  DRAKON has only two: "If" and "Switch". Twice as simple.
- **Pattern matching can get hard to read.** In some non-trivial cases it is hard to compare

patterns. Especially if they have several elements which are similar.

```
[foo, bar, X, Y, _, Z]
[foo, dar, X, _, m, _]
```

Calculating the difference between patterns can become hard mental work. DRAKON believes that all unnecessary work must be eliminated.

One important exception is the **receive** statement. Since this statement is very special in Erlang, it should be written in the text form as usual. But if an application is based on OTP there should not be any **receive** statements anyway.

Branching is making decisions. Decisions are important in programming. That is why DRAKON focuses on representing decisions in a clear and consistent way.

Summary

Functional programming and DRAKON

The Böhm-Jacopini theorem states that any imaginable algorithm can be expressed with only 3 basic building blocks:

1. A sequence of actions.
2. Branching based on a boolean expression.
3. A loop construct.

Functional programming throws away loops. Doing so makes programming conceptually easier:

- All values can be made read only. Everything is passed by value.
- Function composition becomes safe.
- An algorithm receives an explicit input and output.

But functional programming has a problem. This problem is the tradition to use hard-to-read text-based notation. DRAKON-Erlang alleviates this issue by using graphics instead of text for flow control.

DRAKON-Erlang integrates two state-of-the-art technologies:

- Erlang, a functional programming language with built-in support for concurrency.
- DRAKON, an ergonomically aware graphical language for algorithms.

The rules of DRAKON

Line and icon rules:

- No slanting or curved lines. Only straight lines with right angles.
- No line intersections.
- Icons are placed only on vertical lines.
- Main verticals must not have turns.
- Do not connect branches in a silhouette.
- There must be exactly one "End" icon on a diagram.

Usability recommendations:

- The further to the right — the worse.
- Arrange "Case" icons in a consistent manner.

- Use graphic formulas instead of text formulas to express boolean logic.

Silhouette recommendations:

- Prefer silhouette over primitive when it makes sense.
- The order of branches in a silhouette should be left-to-right.
- If that does not break other rules, sort the "Address" icons of the same branch according to the order of the branches they point to.

Erlang-specific recommendations:

- Don't use pattern matching for branching. Do use pattern matching for other purposes.
- Mark recursive calls.

The principles of DRAKON

- Clarity is above all. It is better to sacrifice space than readability.
- No hidden paths. DRAKON makes all important things visible.
- No redundant visual details.
- An algorithm must look good.

Benefits of DRAKON-Erlang

- DRAKON provides a switch from text to ergonomically optimized graphics.
- Rule "The further to the right — the worse" adds an additional dimension to the algorithm.
- The silhouette construct introduces hierarchy into the diagram.
- Boolean expressions are not cryptic any more.
- Branching is done in a simple, visible and consistent way.

Sources

- http://learnyousomeerlang.com/
- Erlang and OTP in Action. Martin Logan, Eric Merritt, and Richard Carlsson. Manning, 2010. ISBN: 1933988789.
- Учись писать, читать и понимать алгоритмы. Пароджанов В.Д. ВМК Пресс, 2012. ISBN: 978-5-94074-800-7
- DRAKON: The Human Revolution in Understanding Programs. Stepan Mitkin. 2011.
- DRAKON Editor: http://drakon-editor.sourceforge.net/

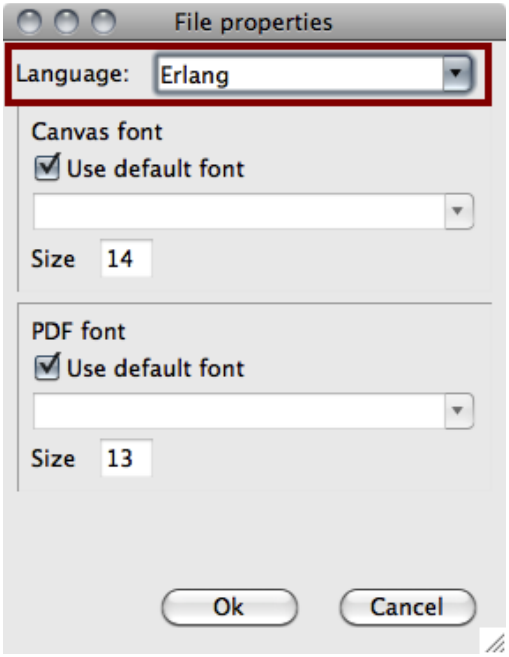
Appendix: Implementation of DRAKON-Erlang in DRAKON Editor 1.14

Generating Erlang source code

How to create a DRAKON-Erlang file:

1. Create a new .drn file with DRAKON Editor.
2. Choose **Erlang** as the language of the file.

   Main menu / File / File Properties...



How to generate Erlang source code from the .drn file:

Main menu / DRAKON / Generate code

Or Ctrl+B (Command+B on Mac).

An Erlang source file will be created near the .drn file.

There is also a command line tool for generating code from .drn files (**drakon_gen.tcl**). Using this tool, you can specify the output directory for the generated files.

drakon_gen.tcl is not available in the Mac version of DRAKON Editor. But you can download the cross-platform package and run the tool.

The module name

DRAKON Editor puts a **module** statement at the beginning of the source file. The module name is taken from the file name.

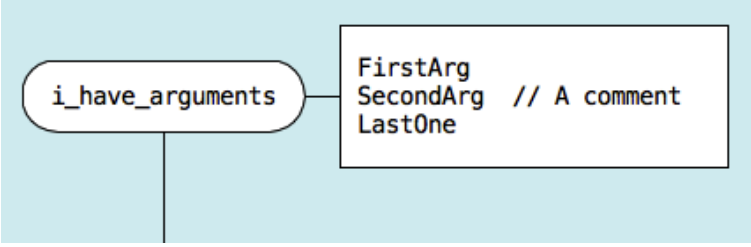
Here is the autogenerated **module** statement for the file called **demo.drn**:

```
-module(erldemo).
```

Function arguments

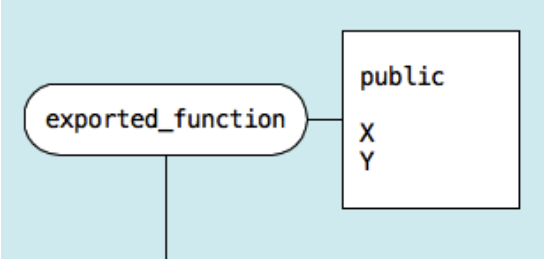
The function arguments should be placed in the "Formal parameters" icons. It's just an "Action" icon placed to the right from the diagram header.

Each argument occupies one line.



Exported functions

In order to make a function exported, put the **public** keyword on the first line inside the "Formal parameters" icon.



DRAKON Editor will add this function to the **export** statement at the beginning of the file.

```
-export([exported_function/2]).
```

Commas, dots and semicolons

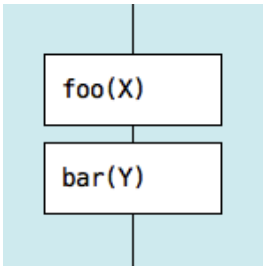
The standard Erlang has quite complex punctuation rules. Statements must end with a different symbol depending on the context:

- comma;
- dot;
- semicolon;
- nothing.

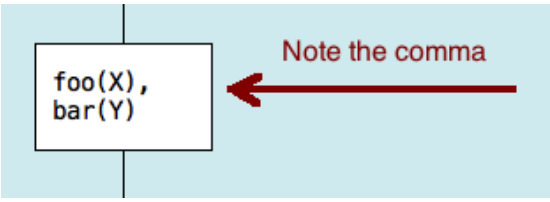
DRAKON-Erlang removes this complexity.

The rule is:

- Put no punctuation to icons that have only one statement.



- Put a comma at the end of the statement when there are many of those in the icon.



Exceptions

Exceptions introduce hidden paths into the algorithm. DRAKON as a language aims to expose hidden paths. This is why DRAKON does not have a visual construct to support exceptions.

Yet exceptions are implemented very well in Erlang. They are also relatively harmless in this language. At least when the code that throws them does not have any side effects.

In order to catch exceptions in DRAKON-Erlang, put **try** and **catch** statements inside "Action" icons.

Arbitrary source code

You can instruct DRAKON Editor to insert some free text into the generated source file. This is done by adding special sections to the file description:

- **header**. The text that goes before the functions generated by the editor.
- **footer**. The text that follows the functions.

Here is a typical file description (main menu / File / File description...):

```
The text that goes before the section
is ignored.
It is the DESCRIPTION of the file.

=== header ===
some_hand_made_function() -> true.

=== footer ===
another_hand_made_function() -> false.
```

Limitations on the content of "If" icons

The expression inside the "If" icon cannot contain calls to any user-defined function. Only a small set of built-in-functions (BIFs) can be used.

This constraint is the same as the one for Erlang guards.

stipan.mitkin@gmail.com