Defining functions

Functions can be thought of as microservices written in the rule language and are defined like this:

```
<name>(<param>, ..., <param>) =
<expr>
```

Example:

```
square(*n) = *n * *n
```

Variables in functions: The let expression As function definitions are based on expressions rather than action sequences, we cannot put an assignment directly inside an expression. For example, the following is not a valid function definition:

```
quad(*n) = *t = *n * *n; *t *
*t
```

To solve this problem, the let expression provides scoped values in an expression. The general syntax for the let expression is:

```
let <assignment> in <expr>
```

For example:

```
quad(*n) = let *t = *n * *n in
*t * *t
```

The variable on the left hand side of the assignment in the let expression is a let-bound variable. The scope of such a variable is within the let expression. A let bound variable should not be reassigned inside the let expression.

Defining Rules

Define rules with nontrivial rule conditions like this:

```
<name>(<param>, ..., <param>) {
 on(<condition>) {
   <actions>
 }
}
```

The rule condition can be skipped for rules with trivial or non-existent conditions:

```
<name>(<param>, ..., <param>) {
<actions>
}
```

A rule can have multiple conditional expressions:

```
<name>(<param>, ..., <param>) {
 on(<condition>) { <actions> } ...
 on(<condition>) { <actions> }
}
```

Generating and Capturing Errors

In a rule, we can also prevent the rule from failing when a microservice fails:

```
errorcode(msi)
```

The errormsg microservice captures the error message, allows further processing of the error message, and avoids the default logging of the error message, like so:

```
errormsg(msi, *msg)
```

In a rule, the **fail**() and **failmsg**() microservices can be used to generate errors. **fail**(errorcode) generates an error with an error code.

Example:

```
fail(-1)
```

failmsg(<errorcode>, <errormsg>) generates an error with an error code and an error message.

Example:

```
failmsg(-1, "this is an error message")
```

The msiExit microservice is similar to failmsg:

```
msiExit("-1", "msi")
```

Inductive Data Types

The features discussed in this section are currently under development!

An inductive data type is a data type for values that can be defined inductively, i.e. more complex values can be constructed from simpler values using constructors. General syntax:

```
data <name> [ ( <type param list> ) ] =
  | : <data constructor type>
  ...
  | <data constructor name> :
    <data constructor type>
```

For example, a data type that represents the natural numbers can be defined as

```
data nat =
 | zero : nat
 | succ : nat -> nat
```

Here the type name defined is `nat`. The type parameter list is empty. If the type parameter list is empty, we may omit it. There are two data constructors. The first constructor "zero" has type "nat," which means that "zero" is a nullary constructor of nat. We use "zero" to represent "0". The second constructor "succ" has type "nat -> nat" which means that "succ" is unary constructor of nat. We use "succ" to represent the successor. With these two constructors we can represent all natural numbers: `zero`, `succ(zero)`, `succ(succ(zero))`.

Pattern matching

If a data type has more than one data structure, then the "match" expression is useful:

```
match <expr> with
 | <pattern> => <expr>
 ...
 | <pattern> => <expr>
```

For example, given the nat data type we defined earlier, we can define the following function using the match expression:

```
add(*x, *y) =
 match *x with
 | zero => *y
 | succ(*z) => succ(add(*z, *y))
```

For another example, given the "tree" data type we defined earlier, we can define the following function

```
size(*t) =
 match *t with
 | empty => 0
 | node(*v, *l, *r) => 1 +
size(*l) + size(*r)
```