Program structure and declarations

F language program is actually a ***sequence of declarations***. Semicolons separate declarations from each other.

```
Program
    Declaration { ; Declaration }
```

Each declaration introduces ***an entity***.

```
Declaration
    Identifier [ : Type ] is Expression
```

Typically, an entity is characterized by the following properties (attributes):

- ***Name*** which is actually an identifier; the name is used to refer to the entity from other program parts.

- ***Type*** that indicates 1) the set of possible values that the entity can get, and 2) the set of possible operations on values of the type. The F language has a variety of possible types, see below.

- ***Value*** which entity obtains by its declaration (initial value). Later, while program execution, the initial value can be changed (for example, by assignment). Syntactically, the initial value is specified as `Expression`.

Notice that in most cases, it is not necessary to specify the type of the entity explicitly. The reason is that the type can be *deduced* by the compiler automatically, by the type of the initial value (which is mandatory by the syntax rule). Perhaps the only case where explicit type specification is necessary, is declarations of function parameters.

Here are some declaration examples:

```
a is 5; // a has integer type and the initial value of 5
b is 7.8;   // b is of real type with the value of 7.8
c is "John Lord"; // c is string "John Lord"

d: integer is 123;
          // d has integer type specified explicitly
```

Notice that there is no notion of “main program” or “entry point” in F language programs. The rule is that any function can play a role of the starting function. The name of the starting function should be specified as one of the compiler option, see later.

Below is the example of a simple F program:

```
a is 123
b is 45.6

max is func(p1:integer, p2:integer) => if p1>=p2 then p1
else p2 end
min is func(p1:integer, p2:integer) => if p1<=p2 then p1
else p2 end

main1 is func() do print "max = ",max(a,round(b)),"\n" end
main2 is func() do print "min = ",max(a,round(b)),"\n" end
```

The program can be compiled with either `main1` or `main2` as main functions. The details of the example that are not clear yet are to be explained later.

Entity types

The F language contains rich set of entity types:

- Boolean type
- Integer type
- Real type
- Rational type
- Complex type
- Tuple type
- Array type
- Map type
- Function type

The type is assigned to an entity by its declaration and cannot be changed while program execution. This rule is known as ***strong typing*** principle.

The formal grammar production for the notion of type is presented below:

```
Type
     BooleanType
     IntegerType
     RealType
     RationalType
     ComplexType
     StringType
     FunctionType
     TupleType
     ArrayType
     MapType
```

Boolean, integer, real, rational, complex, and string types are specified in programs by means of the corresponding keywords:

```
BooleanType
     boolean
IntegerType
     integer
RealType
     real
RationalType
     rational
ComplexType
     complex
StringType
     string
```

Some examples:

```
a: integer is 777
b: string is "This is a string\n"
```

Atomic types

The first three types (boolean, integer, and real) are treated as ***atomic***; this means that there are no way to get access to any part of their values. Rational and complex types are actually compound types: they consist of pairs of values (see details later), and there are standard functions that get access of the parts (real and imaginary parts of complex numbers, and numerator and denominator of rational numbers).

Integer literals (constant) are represented as a sequence of digits.

Real values are represented as two sequences of digits separated by the dot character (`.`). The first sequence represents integer part of the real value, and the second part represent mantissa. Notice that both sequences should present. In other words, notation like `5.` or `.7` are not considered as valid real values. Notice also that exponential form of reals (like `5.0E-3`) is not supported.

Rational numbers are represented to sequences of digits separated by backslash character (`\`). The first sequence is for numerator, and the second one – for denominator.

Complex numbers is the special mathematical abstraction that is usually defined using the following formula: $c = a+i*b$, where a and b are real numbers, and $i = \sqrt{-1}$. Complex numbers are represented in the F language as two integer or real literals separated by the `i` character. The first literal represents real part of the complex number, the second literal – the imaginary part.

Finally, there are two boolean literals representing possible truth values: **`true`** and **`false`**. Notice that these literals are syntactically represented as keywords (reserved identifiers).

Examples:

```
a is 1\2       // a is of type rational
b is num(a)    // b is of type integer, and equals to 1

c is 1.2i3.4   // c is of type complex
d is im(c)     // d is of type real, and equals to 3.4
e is re(c)
```

In addition, there are standard functions creating complex and rational values:

```
x1 is 1.2
x2 is 3.4
x is compl(x1, x1+x2)

y1 is 777
y is rat(y1, round(x2))  // 777\3
```

The second language principle (in addition to strong typing) is that all kinds of ***implicit conversions are strongly prohibited***; any type conversion should be specified explicitly. The best way to explain the principle is to show some examples:

```
a is 123.456   // real type
b is a
   // Error: cannot implicitly convert real to integer
c is round(a)  // Correct

x is 33\55
x1:integer is x
```

```
        // Error: cannot convert rational to integer
x2 is denom(x)    // Correct: x2 gets the value of 55

t is true         // type boolean
t1: integer is t
        // Error: cannot convert boolean to integer
t2 is if t then 1 else 0 end  // Correct
```

The only exception from this rule is that it's legal to convert integers to reals. The idea behind the exception is that such a conversion doesn't lead to any data loss:

```
p is 777
p1: real is p+1   // Correct: the value of p1 is real
778.0.
```

Below are tables specifying all valid types of operands for operators on atomic types:

Binary infix operators + (addition), – (subtraction), and * (multiplication)

Left operand	Right operand	Result
integer	integer	integer
integer	real	real
real	integer	real
real	real	real
integer	rational	rational
rational	integer	rational
rational	rational	rational
integer	complex	complex
real	complex	complex
complex	integer	complex
complex	real	complex
complex	complex	complex

Binary infix operator / (division)

Left operand	Right operand	Result
integer	integer	real
integer	real	real
real	integer	real
real	real	real
integer	rational	rational
rational	integer	rational
rational	rational	rational
complex	integer	complex
complex	real	complex
complex	complex	complex

Binary infix relational operators: <, <=, >, >=, = /=

Left operand	Right operand	Result
integer	integer	boolean
integer	real	boolean
real	integer	boolean

real	real	boolean
integer	rational	boolean
rational	integer	boolean
rational	rational	boolean
complex	complex	boolean

Binary infix boolean operators: & (logical and), | (logical or), and ^ (exclusive or)

Left operand	Right operand	Result
boolean	boolean	boolean

All other combinations of operands are illegal.

Below is the list of the standard type conversion functions:

Function name	Argument type(s)	Resulting type	Comment
round	real	integer	Rounding of the real argument
round	rational	real	Creates the new real number converting the rational number from the argument.
re	complex	real	Returns the imaginary part of the complex number
im	complex	real	Returns the real part of the complex number
num	rational	integer	Returns the numerator of the rational number
denom	rational	integer	Returns the denominator of the rational number
compl	integer real real, real real, integer integer, real integer, integer	complex	Creates the new complex number from two components provided as arguments. For the case when only one argument, the second one is assumed to be equal to real 0.0.
rat	integer integer, integer	rational	Creates the new rational number from two components provided as arguments. If only one argument is provided, the second one is assumed to be equal to integer 1.
norm	rational	rational	Normalizes the rational number given as argument.

String types and string entities

```
hello is "Hello, world!\n"
sentence is ["this", "is", "array", "of", "strings"]
```

Functions and function types

Functions and function types are perhaps the central and most important part of the language. In the F language, functions are treated as “first-class entities”. This means that functions behave exactly as all other kinds of entities. For example, it’s possible to specify functions as initial values of entities (making those entities as ones having “functional types”). Functions can be passed as arguments to other functions, and even

they can serve as result values of other functions. In addition, of course, the most important way of using functions is to “call” them (i.e., to activate their algorithms).

Let’s consider a few code snippets. Below is the simplest example of function:

```
inc is func(p: integer) => p+1
```

Here, we declare a new entity, `inc`, and give it the initial value which is *function*. The function declaration starts with the **`func`** keyword following parameters (or just empty parentheses if there are no parameters). The body of the function (single expression in this case) starts after `=>` token. It’s clear that the function just increases the value of its parameter and returns the resulting value.

Notice that there is no need to specify the type of the value returned by the function: the compiler is able to deduce this type from the expression that specifies the value to be returned. Nevertheless, it’s legal (and sometimes is desirable for better readability) to specify the return type explicitly:

```
inc is func(p: integer): integer => p+1
```

That’s the important point: what’s the type of the `inc` entity after this declaration? By definition, `inc` has *function type*. This type can be written as **`func(integer):integer`**. Actually, this obvious notation defines *all functions* with one integer parameter returning a value of integer type. This means, among other things, that later `inc` entity can get another function (e.g., by assignment) with the same type (that is, with the same signature), for example:

```
inc := func(p: integer) => if p>=0 then p else -p end
```

The functional approach to programming assumes that the whole program is built as a set of (simple) functions calling each other to get the result. However, sometimes function should perform more complex actions than is shown on the previous examples. In that case, the function body will contain several statements. Below is the example:

```
average is func(array:[integer]): real do
    avg is 0.0;
    for i in array do
        avg := avg + i
    end;
    return avg/length(array)
end
```

Here, the algorithm of the function is specified by the *block*: a sequence of statements enclosed by **`do`** and **`end`** keywords. Function parameter is an *array* (of an arbitrary length) consisting of integer elements. The notation for compound types (arrays, tuples and maps) is introduced below. The standard function `length`, being applied to array, returns the current number of array elements.

The main operation on functions is *call* where the algorithm of the function is activated:

```
x1 is inc(777)
        // x1 becomes the result of the function call: 778
x2 is average([1,2,3,4,5,6])   // x2 equals to 3.5 (real)
```

In both examples, types of `x1` and `x2` get deduced from types of initializing functions.

Functions provide great power and flexibility in programming. In particular, a function can accept another function as parameter:

```
apply is func(array: [integer], conversion:
func(integer):integer) do
    newarr is []; // newarr is initially empty
    for elem in array loop
        newarr := newarr + conversion(elem)
    end
end
```

Here the function conversion gets passed to apply as parameter and is used inside it for creating a new array from another one which is also passed to the function.

This example illustrates how a function can be returned as the result of some other function:

```
incGenerator is func(p:real) do
    f is func(p1:real) => p+p1;
    return f
end

foo1 is incGenerator(7.7)    // foo1 is a function...
...foo1(8.7)...              // ...returning 16.4
foo2 is incGenerator(5)      // foo2 is a function...
...foo2(10)...               // ...returning 15
```

Below are formal grammar productions describing notions of function and function type:

```
FunctionType
     func ( [ Type { , Type } ] ) : Type

Function
     func ( [ Parameter { , Parameter } ] ) [ : Type ] Body

Parameters
     Identifier : Type

Body
     do Statements end
     => Expression
```

Compound types

Compound types are those whose values consist of some number of elements, and it's possible to get access to the elements independently.

There are following compound types in the F language:

- Arrays
- Maps, of associative arrays
- Tuples

Arrays

Arrays in the F language are defined in the similar way as arrays in many other programming languages: this is a compound type consisting of elements of the same type. Elements are enumerated by integer indices, starting from 0 (that is, the first array element has the index 0). There is one special feature in F arrays: they are *completely*

dynamic. This means that if an array was defined, say, with three elements, then it can be later increased. Below are some examples.

```
arr is [1, 2, 3]  // arr is array of three elements of type
integer
arr : = arr + 7   // now arr has four elements: 1, 2, 3,
and 7

vector:[real] is []
```

The declaration of `arr` introduces the array consisting of integer values. The type of `arr` is not specified explicitly; it is deduced by the compiler from the type of the elements from the initializer.

The array type definition is syntactically defined as `[Type]`. So, the full declaration of `arr` from the example above will look like as follows:

```
arr: [integer] is [1, 2, 3]
```

The declaration of `vector` introduces the array of reals. Initially it is empty, and the compiler cannot deduce element type; therefore, it's necessary to specify the type explicitly.

Access to array elements is performed by the usual construct with square brackets:

```
elem is arr[i+1]
```

where the array name is written in front of brackets, and the index of the element (actually, in form if expression) is written within the brackets.

The special standard function `length()` is used to get the current array length (number of elements):

```
l is length(arr)
```

In case where it's necessary to specify array type explicitly (e.g., when an array is passed to a function as parameter), the following notation is used:

```
foo is func(arr: [boolean]) ...
```

By definition, function foo accepts any array consisting of boolean values, of any length.

The formal grammar productions specifying array types and arrays, is below:

```
ArrayType
     [ Type ]

Array
     [ Expression { , Expression } ]
```

The table below defines all possible operators on arrays.

Left operand	**Binary infix operator**	**Right operand**	**Result**	**Comment**
Array of type `T`	+	Entity of type `T`	Array of type `T`	New array element is appended to the array.
Array of type `T`	+	Array of type `T`	Array of type `T`	Array concatenation: two arrays compose the

				single result array where elements of the second array go after element of the first array.

Maps, or associative arrays

```
telephone_book is { "John Lord": 1000125,
                    "Ian Gillan": 2200330 }


MapType
     { Type : Type }

Map
     { Pair { , Pair } }

Pair
     Expression : Expression
```

Tuples

Example:

```
info is (name is "John", age is 21, 0.325) // tuple
declaration
```

Here is the example of the tuple declaration. The entity `info` is the tuple: a composite entity consisting of three elements: `name`, which is a string, `age` of integer type, and the *unnamed* element of type `real`.

```
n is info.name     // ‘n’ gets the value "John"
info.age := info.age+1   // access to named tuple element
info.3 := 0.7            // access to unnamed tuple element

empty is ()  // empty tuple


printTuple is func(t:(string,integer,real)) do
    for e in t loop
        print(e)
    end
    print   // prints "\n"
end

printTuple(info)  // invokes the function above for the
tuple info


// The function calculates the roots of a square equation,
// defined by its three coefficients.
sqRoots is func(a: real, b: real, c: real) : (real, real)
is
```

```
    d is sqrt(b*b - 4*a*c)
    return ((-b+d)/(2*a),(-b-d)/(2*a))
end

roots is sqRoots(1,2,3)
root1 is roots.1
root2 is roots.2
```

The function `sqRoots()` returns the tuple with two unnamed components, both of type `real`. After calling the function, its result is assigned to the `roots` variable, and later two components (i.e., two roots) can be taken separately.

Below are formal grammar productions defining the syntactic structure of type types and entities of type tuple.

```
TupleType
     ( Type { , Type } )

Tuple
     ( [ TupleElement { , TupleElement } ] )

TupleElement
     [ Identifier is ] Expression
```

Expressions

Expression are constructs that are used for calculating new values. In the F language, expressions follow usual syntactic rules that are described by following formal grammar productions:

```
Elementary
     false
     true
     INTEGER_LITERAL      // 1234
     REAL_LITERAL         // 12.34
     RATIONAL_LITERAL     // 7\8
     COMPLEX_LITERAL      // 1.2i3.4
     STRING_LITERAL       // "string"
     Identifier           // entity reference

Primary
     Elementary
     Conditional
     Function
     Array
     Tuple
     Map
     ( Expression )

Conditional
     if Expression then Expression else Expression end

Secondary
     Primary
     Secondary ( [ Expression { , Expression } ] ) // call
     Secondary [ Expression ]           // map/array element
     Secondary . IDENTIFIER             // named tuple
element
```

```
        Secondary . INTEGER_LITERAL       // unnamed tuple
    element

    Expression
        Secondary
        Expression OperatorSign Expression

    OperatorSign
        One of
        +  -  *  /  <  >  <=  >=  =  /=  &  |  ^
```

Statements

```
    Statement
        Assignment
        FunctionCall
        IfStatement
        LoopStatement
        ReturnStatement
        BreakStatement
        PrintStatement
        Declaration

    FunctionCall
        Secondary ( [ Expression { , Expression } ] )

    Assignment
        Secondary := Expression

    IfStatement
        if Expression then Statements [ else Statements ] end

    LoopStatement
        [ LoopHeader ] loop [ Statements ] end

    LoopHeader
        for [ Identifier in ] Expression [ .. Expression ]
        while Expression

    ReturnStatement
        return [ Expression ]

    BreakStatement
        break

    PrintStatement
        print [ Expression { , Expression } ]
```

```
a is 5              // integer
b is 7.8            // real
c is “string”       // string
d is [1, 2, 3]                                      // array


a1 is 1\2           // rational
b1 is 1i2           // complex
b2 is 1.2i3.4       // complex


a := expression


d[i] := expression
e1 := e + "one"     // append
e2 := e1 + "more"   // append
e3 is e1 + e2       // list concatenation
for l in e3 loop
   ...
end


f1 is func(p:Integer) => p+1
f2 is func() return func()=>1 end
```